

Matrox Imaging Library

Version 9.0 with Processing Pack 1

User Guide

Manual no. Y10513-301-0910

July 29, 2009

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

Microsoft®, MSDN®, Visual Basic®, Visual C++®, Visual C#®, Visual Studio®, and Windows® are registered trademarks of Microsoft Corporation.

PCI-X® and PCI Express® are registered trademark of PCI-SIG.

PCIe™ is a trademark of PCI-SIG.

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

Camera Link® is a registered trademark of the Automated Imaging Association (AIA).

GigE Vision™ is a trademark of the Automated Imaging Association (AIA).

Intel®, MMX®, Celeron®, and Pentium® are registered trademarks of Intel Corporation.

Texas Instruments® is a registered trademark of Texas Instruments Incorporated.

PowerPC® is a registered trademark of International Business Machines Corporation, used under license by Freescale Semiconductor Inc.

Freescale™ is a trademark of Freescale Semiconductor Incorporated.

Linux® is a registered trademark of Linus Torvalds.

Red Hat® is a registered trademark of Red Hat, Inc.

SUSE® is a registered trademark of Novell, Inc.

Ubuntu® is a registered trademark of Canonical, Ltd.

Portions of this software are copyright ©2007 The FreeType Project.

Parts of the MIL Driver for GigE Vision™ are copyrighted © and are the properties of StorageCraft, Inc.

Some scripts in this help file are implemented using the jQuery library version 1.2.6 copyright ©2008 John Resig.

The ColorSpace tool is copyrighted © and is the property of Philippe Colantoni (www.couleur.org).

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

Protected by U.S. Patents 7,027,651; 7,319,791; 7,327,888; U.S. Patents Pending.

© Copyright Matrox Electronic Systems Ltd., 1992-2009.

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 might 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: Introduction. 39

Introduction to MIL and MIL-Lite.	40
MIL and MIL-Lite modules.	41
Functionality available in both MIL and MIL-Lite.	41
Functionality available only in MIL.	42
MIL and the Intel MMX/SSE technologies.	46
MMX.	46
SSE.	47
Requirements to run MIL.	47
Under Windows.	47
Under Linux.	48
Before you start.	50
Installation.	51
Under Windows.	51
Under Linux.	52
General installation notes.	54
MilConfig.	55
Testing the installation.	55
Communicating properly.	56
Compiling and linking.	57
The MIL User Guide.	59
Examples.	59

Chapter 2: Building an application. 61

Organization of the MIL modules.	62
Similar MIL functions in different modules.	62
Interactive functionality.	64
Allocating an application and its systems.	65
Mandatory allocations for an application.	65
Board-type versus Host-type systems.	66
Multiple systems.	67
Allocating and displaying an image buffer.	69
Allocating an image buffer.	69
Displaying an image buffer.	70
Grabbing images.	71
Continuous grabbing and adjusting your camera.	72
Sequential grabbing.	72
An example.	72
Dealing with color.	74
Grabbing.	74
Displaying.	75
Managing color images.	75
Using the defaults.	76
Using MilConfig to change your default settings.	76
Using your defaults.	76
Error reporting.	77
Multiple systems.	77
MIL custom data types, extensions, and portability functions.	78
MIL custom data types.	78
M_MIL_USE_SAFE_TYPE extension.	80
Portability functions.	81

Porting a 32-bit MIL application to MIL 64-bit.	82
Modified functions.	82
Void pointers.	84
Project processor definitions.	85
MIL under Linux.	86
How to create a portable application.	86

Chapter 3: Image processing. 87

Image processing overview.	88
MIL and image processing.	88
Steps to performing a typical application.	89
A typical application.	90
How to encode these steps.	91
Image quality.	93
Techniques to improve images.	94
Averaging an input sequence.	95
Denoising.	96
Low-pass spatial linear filters.	96
Rank filters.	98
Area open and area close.	98
Erosion and dilation.	101
Basic erosion.	102
Basic dilation.	102
An example.	103
Opening and closing.	105
Basic geometric transforms.	106
Image manipulation in general.	107

Image statistics.	107
Generating a histogram.	108
Calculating general statistics.	110
Finding the image extremes.	111
Locating events.	111
Counting image differences.	112
Projecting an image to one dimension.	112
Thresholding your images.	113
Binarizing.	113
Clipping.	115
Histogram equalization.	115
Enhancing and detecting edges.	116
Edge enhancement.	118
Edge detection.	119
Arithmetic with images.	122
Combining images.	122
Mapping an image.	124
Distance transform.	125
City Block transform.	125
Chessboard transform.	126
Chamfer 3-4 transform.	126
Labeling.	127
Peak intensity detection and range images.	129
Creating depth maps (range images).	132

Chapter 4: Advanced image processing..... 135

Advanced image processing in general.	136
Custom spatial filters.	136
Finite Impulse Response (FIR) filters.	136
Infinite Impulse Response (IIR) filters.	140
A summary.	142
An example.	143
Custom morphological operations.	145
Defining your own structuring element.	145
Erosion and dilation.	146
Thinning and thickening.	151
Matching.	153
Searching for hits or misses.	153
Connectivity mapping.	154
Fast Fourier Transform.	157
Magnitude and phase.	158
Filtering an image.	160
Watershed transformations.	164
Using watersheds to separate touching objects.	165
Using watersheds to separate objects from their background.	167
Minimum grayscale variation of a catchment basin.	168
Using marker images.	169
Style of the watershed lines.	171
Exact versus straight.	172
Skipping the last level.	172
Filling the source.	173
Polar-to-rectangular and rectangular-to-polar transforms.	174

Warping.	176
First-order polynomial warpings.	183
Perspective polynomial warpings.	184
Custom warpings.	185
Interpolation modes.	185
Points outside the source buffer.	186
Discrete Cosine Transform.	187
Deinterlacing.	188
Discard algorithm on the entire image.	189
Averaging algorithm on the entire image.	190
Bob algorithm on the entire image.	192
Adaptive algorithms.	193

Chapter 5: Camera calibration. 197

Camera calibration - overview.	198
Types of distortions.	199
Calibration mechanism.	200
Steps to performing a calibration.	201
Basic concepts.	203
Coordinate systems.	205
Absolute world coordinate system.	206
Relative world coordinate system.	206
Tool coordinate system.	208
Camera coordinate system.	209
Robot-base coordinate system.	209
Image coordinate system.	210

Calibrating your camera setup.	210
Real-world grid.	210
Image of the grid.	211
Specifying the top-left corner of your grid.	213
List of coordinates.	216
Calibration modes.	217
Piecewise linear interpolation mode.	218
Perspective transformation mode.	219
Tsai-based mode.	219
Robotics mode.	221
Inquiring about your calibration.	222
Calibration status and errors.	222
Calibration accuracy.	222
Position and orientation of coordinate systems.	223
Pinhole camera.	225
Getting results in real-world units.	226
Transforming coordinates or results.	226
Physically correcting an image.	226
Automatically getting results in real-world units.	230
Changing your camera setup.	233
Types of transformation.	233
Transforming coordinate systems.	236
Special considerations concerning the tool and camera coordinate systems.	239

Calibrating a camera setup that analyzes large objects.	241
Single camera fixed on a movable tool (manipulator): tool coordinate system example.	242
Single camera and movable object: relative coordinate system example.	243
Several fixed cameras and fixed object: grid offset example.	244
Processing calibrated images.	245
Processing calibrated images.	245
Saving and reloading a calibrated image.	246
Saving and reloading the child buffer of a calibrated image.	247

Chapter 6: Blob analysis. 249

Blob analysis overview.	250
MIL and blob analysis.	250
Steps to performing blob analysis.	251
Identifying blobs.	253
Segmenting the blob image.	254
Preprocessing.	254
Adjusting blob analysis processing controls.	255
Controlling the image lattice.	255
The pixel aspect ratio.	256
Setting the blob identification mode.	257
Selecting blobs.	259
Making feature extractions.	261
The area and perimeter.	263
Dimensions.	266
Determining the shape.	268

Finding the blob location.	269
Blob points.	270
Chained pixels.	271
Moments.	273
Location, length and number of runs.	273
Blob reconstruction.	273
Merging results.	278
A simple merge.	280
Border blobs.	281
Inclusion state.	283
Other remarks.	285
Blob analysis example.	286

Chapter 7: Pattern matching. 291

Pattern matching - in general.	292
Steps to performing a pattern search.	293
Defining a model.	295
Rotation.	296
Setting the angle of search.	298
Determining the rotation tolerance of a model.	299
Masking the model.	300
Search constraints.	302
Specifying the number of matches.	302
Setting the acceptance level.	303
Setting the certainty level.	303
Redefining the model's reference position.	304
Selecting the search region.	305
Positional accuracy.	306
Setting the speed parameter.	306

Preprocess the search model.	307
Speeding up the search.	307
Choose the appropriate model.	308
Adjust the search speed setting.	308
Effectively choose the search region and search angle.	309
Searching for multiple models at the same time.	309
The pattern matching algorithm (for advanced users).	310
Normalized correlation.	310
Hierarchical search.	313
Search region.	315
Search heuristics.	315
Subpixel accuracy.	316
Pattern matching examples.	317

Chapter 8: Geometric Model Finder. 331

Geometric Model Finder module.	332
Steps to performing a model search.	333
Basic concepts.	334
Types of Model Finder contexts.	336
General geometric contexts.	336
Controlled geometric contexts.	336

Defining and adding models to your Model Finder context.	337
Image-type models.	337
Synthetic models.	342
Models defined from result buffers.	344
Models defined from two other models.	346
Preprocessing.	346
Model origin.	346
Model indices and labels.	347
Drawing and inquiring the model's active edges.	347
Guidelines for choosing models.	350
Make sure your images have enough contrast.	350
Avoid poor geometric models.	350
Be aware of ambiguous models.	351
Nearly ambiguous models.	353
Masking your model.	354
Search targets.	359
Finding models in an image.	359
Finding models in an Edge Finder result buffer.	360
Determining what is a match.	361
Score and target score.	361
Model and target coverage.	362
Fit error.	363
Interpreting results.	364
Position, angle, and scale.	366
Enabling calculations specific to searching within a range.	366
Search position and position range.	367
Angle and angular range.	369
Scale and scale range.	370

Customizing search settings.	371
Acceptance levels.	372
Certainty levels.	373
Expected number of occurrences.	373
Reference axis.	375
Advanced search settings.	380
Polarity.	380
Separation.	381
Shared edges.	385
Fit error weighting factor.	385
First and last levels.	386
Global context settings.	387
Setting the search speed	387
Accuracy.	387
Timing out your search.	388
Speeding up the search.	388
Adjust the search speed of the algorithm.	388
Disable calculations specific to range search strategies.	389
Define several models with different expected angles.	389
Limit the position range.	389
Limit the search scale.	389
Specify the exact number of expected occurrences.	390
Clean up your model and your target.	390
Changing the search levels.	390
Retrieving and analyzing results.	391
Possible results.	391
Drawing results.	392

Calibration.	393
Requirements.	393
Models and their calibration object.	395
Setting the aspect ratio control.	395
Using transformation coefficients with calibrated images.	395
Geometric Model Finder example.	397

Chapter 9: Edge Finder. 407

MIL Edge Finder module.	408
Edges and Edge Finder.	409
Steps to extract and analyze edges.	410
Basic concepts.	411
Extracting the edges.	412
The basics.	412
Object contours versus line crests.	414
How edges are calculated.	416
Customizing the edge extraction settings.	419
Filter type.	420
Filter mode.	423
Smoothing.	425
Thresholding.	426
Edgel accuracy.	428
Magnitude type.	429
Filling the edge gaps.	430
Edge features.	433
Dimension features.	434
Location features.	437
Advanced features.	438
Grouped features.	440

Calculating and retrieving results.	441
Sorting keys.	442
Retrieving the results.	442
Selecting the results.	444
Internal processing buffers.	449
Post-calculation.	452
Annotating the results.	454
Advanced edge extraction.	456
Approximating the edges.	456
Masking the edges.	457
Cropping the edges.	458
Advanced thresholding.	460
Providing the image's derivatives.	460
Putting data into an Edge Finder result buffer.	463
Finding the closest edgels to a list of points.	464
Optimizing edge extractions.	471
Specify a region of interest (ROI).	471
Perform post-calculation.	471
Optimize your control type settings.	472
Retrieving calibrated results.	474
Interfacing with Geometric Model Finder.	475
MIL Edge Finder example.	477

Chapter 10: Registration..... 481

MIL Registration module.	482
Steps to performing registration.	484
Basic concepts.	485
The registration process.	486
Correlation contexts.	487

Registration elements and images.	487
Setting the rough location of your images.	490
Selecting an image's reference coordinate system.	490
Specifying the type of transformation and its settings.	493
Copying the rough location.	494
Precision of the rough location and its effect on speed.	497
Customizing your registration settings.	497
Selecting the transformation type.	497
Specifying the maximum allowable displacement.	499
Skipping the optimization step.	499
Specifying the minimum overlap between images.	501
Accuracy.	502
Setting the origin of an image's coordinate system.	502
Retrieving and analyzing results.	503
Possible results.	503
Using the results.	504
Drawing results.	504
Mosaicing.	505
Positioning and scaling your mosaic in the destination image buffer.	506
Mosaic composition in the overlapping regions.	509
Mosaic composition using super-resolution.	510
Registration example.	513

Chapter 11: Optical character recognition. 517

The MIL OCR module.	518
Steps to reading or verifying a string in an image.	519
Basic concepts.	521

Guidelines for choosing context types.	522
General OCR font context type.	522
Constrained OCR font context type.	523
Switching between the two.	524
Deciding which OCR font context type to use.	525
OCR font.	527
User-defined MIL OCR fonts.	528
Existing MIL OCR fonts.	532
Semi fonts.	532
Quality and scale are important.	533
Visualizing.	534
Erasing characters.	534
Defining the target strings.	535
Calibrating your font.	535
Setting appropriate processing controls.	537
Specifying other string information.	538
Constraints.	541
Locating your text.	542
Determining what is a match.	543
Acceptance levels.	543
Unrecognized characters.	544
Retrieving and analyzing the results.	544
A character.	545
A string.	545
A text.	546
Understanding odd results.	546
Hooking functions.	549
Improving search speed.	549
Optical character recognition examples.	551

Chapter 12: String Reader..... 555

MIL String Reader module.	556
Steps to reading a string in an image.	557
Basic concepts.	558
Creating and customizing the fonts for a font-based context.	560
Adding and deleting fonts to the context.	561
Adding and deleting characters to the font.	562
Character representation.	563
Source image foreground.	565
Normalize characters.	565
Space size.	566
Baseline.	566
Sorting characters.	568
Using a fontless context.	569
Enabling and disabling characters.	570
Customizing a fontless context.	570
Adding and deleting string models to the context.	571
Adding and deleting the string models.	572
Preprocess and read.	572
Target image foreground.	573
Space.	573
Space width.	574
Maximum number of consecutive spaces.	575
Number of strings to read.	576

Degrees of freedom.	577
String angle and character angle.	577
String scale and character scale.	579
String aspect ratio and character aspect ratio.	581
Character's maximum baseline deviation.	583
Skew angle.	585
Rules for character placement.	586
String size.	586
Character constraints.	586
Grammar rules.	588
Global context settings.	591
Minimum contrast.	592
Speed.	592
Timeout.	593
Encoding.	593
Scores and acceptances.	594
String acceptance, certainty, and score.	595
Character acceptance and score.	596
String target acceptance, certainty, and score.	598
Retrieving results and annotation.	599
Annotation.	600
Transformation coefficients.	602
Formatted and non-formatted results.	602
Fixing read problems using String Expert.	603
An example.	607

Chapter 13: Codes..... 613

MIL code module.	614
Steps to reading, writing, or verifying a code in an image.	616
Basic concepts.	618
Technical code information.	623
Supported code types.	624
1D code types.	625
2D code types	627
Composite codes.	627
Predefined SEMI code contexts.	628
Supported encoding schemes, sub-types, and error correction schemes by code type.	628
Supported encoding schemes and sub-types.	628
Supported error correction schemes.	631
Customizing read and verify operation settings.	633
Multiple occurrences.	633
Child buffers.	634
Presearching.	634
Foreground color.	634
Setting the search speed.	635
Timing out your search.	635
Cell size and number.	635
Angular search range.	636
Dot spacing.	637
String size.	638
Thresholding.	639

Customizing write operation settings.	641
Destination buffer size.	641
Special characters.	641
Foreground color.	642
Cell size and number.	642
Bearer bars.	643
Verifying your code.	643
Attributes that can be verified.	643
Controls that must be set.	645
Setting the aperture.	645
Retrieving results.	647
Drawing results.	649
Results by code type.	650

Chapter 14: Measurements. 655

The Measurement module.	656
Markers.	657
A multiple marker.	658
Steps to finding and obtaining measurements of markers.	658
Allocating or restoring a marker.	658
Setting the marker's measurement box and other marker characteristics.	659
Viewing the marker.	659
Specifying the measurement control settings.	660
Acquiring and preprocessing a target image.	660
Finding the marker and taking measurements.	660
Reading results.	660
Annotating results.	661
Measurement examples.	661

Measurement box.	661
Placing the measurement box.	662
Marker orientation.	663
Setting the measurement box angle.	663
Multiple-angle search for a marker.	664
Determining the rotation tolerance of a marker.	664
Subregions of the measurement box.	666
Search algorithm.	669
Identifying edges.	669
First derivative filter.	670
Marker characteristics.	671
Edge markers: fundamental characteristics.	672
Polarity.	672
Position and position variation.	673
Contrast and contrast variation.	673
Length.	673
Line equation.	674
Edge markers: advanced characteristics.	675
Edge width.	675
Minimum/maximum position.	676
Edge strength.	676
Edge threshold.	678
Determining the strength of the required edge.	678
Marker reference.	679
Weight factors.	679

Stripe markers: fundamental characteristics.	680
Polarity.	681
Contrast and contrast variation.	681
Width and width variation.	682
Position.	682
Length.	683
Line equation.	683
Stripe markers: advanced characteristics.	684
Minimum and maximum position.	684
Inside edge and inside-edge variation.	684
Position inside stripe.	685
Multiple marker characteristics.	686
Measurements between two markers.	687
Steps to taking measurements between two markers.	687
Calculating with multiple markers.	688
Angle.	688
Line equation and distance.	689
Measurement examples.	689

Chapter 15: Metrology..... 697

MIL Metrology module.	698
Steps to measure and validate expected features.	701
Basic concepts.	702
Features.	704
Reference frame.	704
(Physically) measured features.	709
Constructed features.	717
Multiple features.	726

Geometric tolerances.	727
Adding a geometric tolerance.	728
Using multiple geometric tolerances.	737
Degrees of freedom.	739
Active edgels.	740
Calculating and retrieving results.	745
Retrieving the results.	745
Annotating the results.	747
Metrology example.	750

Chapter 16: 3D Reconstruction. 763

MIL 3D Reconstruction module.	764
Laser line profiling.	765
Triangulation.	767
Steps to extracting a depth map using laser line profiling.	767
Basic concepts.	768
Laser line profiling requirements.	770
Camera angle requirement.	771
Partially corrected depth map requirements.	772
Fully corrected depth map requirements.	772
Calibrating your 3D reconstruction setup.	774
Calibrating for a partially corrected depth map.	774
Calibrating for a fully corrected depth map.	776
Obtaining results.	779
Generating the depth map.	779
Generating the intensity map.	784
Retrieving the coordinates of a cloud of points.	785
Example.	785

Filling missing data points (gaps).	786
Filling mode.	787
Filling operations.	788
Phenomenae causing missing data.	791
Examples.	794
Triangulation.	795
Performing triangulation.	797
MIL 3D Reconstruction example.	799

Chapter 17: Color processing and analysis. 813

Color processing and analysis overview.	814
Basic concepts.	816
Color spaces and converting between them.	817
Color space.	818
Converting between color spaces.	822
Converting to grayscale.	822
Extracting the luminance.	823
Principal component projection.	825
Distance between colors.	830
Differences in distances.	831
Color distance types.	832
Choosing a distance type.	836

Color matching.	838
Steps to performing color matching.	838
The basics.	839
Defining and adding color-samples to your color matching context.	842
Area identifier image.	843
Operation mode and distance type.	844
Acceptance.	846
Distance tolerance.	847
Basic results.	848
Image results.	851
Advanced color matching settings.	855
Internal conversion.	855
Color band specification.	856
Distance normalization.	856
Color space encoding.	857
Color separation.	859
Separation operation.	860
Color statistics.	862
Covariance.	862
Principal components.	863
Color processing and analysis example.	864

Chapter 18: Specifying and managing your data buffers. . . 877

Data buffers.	878
Target system.	879
Specifying the dimensions of a data buffer.	879
Data type and depth.	880
Attribute.	881
Memory locations.	883

Manipulating and controlling certain data buffer areas.	885
Child buffers.	885
Copying specific buffer areas.	887
Processing a non-rectangular region of interest.	889
Managing data buffers.	889
Loading a data buffer.	890
Saving a data buffer.	890
Loading and saving a sequence of data buffers.	891
Controlling how color image buffers are stored.	891
RGB buffers.	892
RGB data formats.	892
Binary buffers.	894
YUV buffers.	894
YUV16 Packed.	895
YUV9 Planar.	897
YUV12 Planar.	897
YUV16 Planar.	898
YUV24 Planar.	898
Child YUV buffers.	899
Accessing a MIL buffer directly.	899
Mapping a data buffer to user-allocated memory.	901
Pixel conventions.	905

Using buffers with the Bayer color filter.	906
Using MIL to convert the image.	907
How the Bayer image gets converted.	910
Performing color correction.	913
Correcting the white balance of your Bayer images.	913
Performing gamma correction.	915
Buffer overscan region.	916

Chapter 19: Lookup tables. 919

Lookup tables in general	920
LUTs and data buffers.	921
Loading and generating data into LUTs.	921
Generating data directly into the LUT buffer.	921
Loading LUTs with precalculated data.	922
Using LUTs.	923

Chapter 20: Displaying an image. 925

Overview.	926
Supported hardware acceleration modes.	927
Types of displays.	929
Windowed display.	929
Auxiliary display.	930
Network display.	932
Display size and depth.	932
Displaying buffers of different data depths.	933
Displaying an image in a user-defined window.	936
Using MdispSelectWindow().	936
Removing a buffer from the display.	944

Displaying multiple buffers.	945
Display number.	948
Screen tearing.	949
No-tearing modes.	950
Region of interest in a windowed display.	952
Defining an ROI in a display interactively.	952
Outlining an established region.	953
Panning and zooming.	954
Annotating the displayed image non-destructively.	955
Annotating images using the overlay-display mechanism.	955
Overlay buffer behavior.	956
CPU-assisted overlay.	957
Transparency (Keying).	957
Using GDI annotations.	958
Mapping 1-band images through a LUT upon display.	961
Selecting the LUT to use for display.	961
Restrictions when displaying using LUT.	962
Advanced display concepts.	963
Display architectures.	963
Video output controllers.	969
Remote desktop.	970
Continuous grab.	970
Overlay.	971
Large buffers.	971
Optimizing auxiliary displays.	972
Traces and user traces.	973

Chapter 21: Generating graphics. 975

MIL and graphics in general.	976
Preparing for graphics.	976
Drawing graphics.	977
Writing text.	980

Chapter 22: Grabbing with your digitizer. 981

Cameras and video sources.	982
Basic concepts.	983
Data format.	984
Device number.	984
Multiple cameras.	986
Simultaneous acquisition.	986
Data input channels of acquisition paths.	988
Switching between cameras of the same type.	989
Switching between cameras of different types.	989
Ultra-fast channel switching.	990
Channel locking and unlocking.	990
Grabbing a single field.	991
Line-scan cameras.	991
Grabbing to the display.	991
Grabbing a sequence of frames in real-time.	992
Grabbing and processing.	993
Grab mode.	993
Multiple buffering.	994
Grabbing large images.	998

Reference levels, lookup tables, and scaling.	999
Black and white reference levels	999
Color image reference levels	1000
Mapping grabbed data through a LUT	1000
Scaling.	1002
Grabbing with triggers and exposures.	1003
Asynchronous reset mode and next valid frame mode.	1003
Setting the exposure.	1005
Software triggers.	1006
Auto-focusing.	1006
Search strategies.	1007

Chapter 23: JPEG and JPEG2000 compression. 1013

Introduction.	1014
JPEG lossless.	1014
JPEG lossy.	1014
Interlaced JPEG.	1015
JPEG2000 lossless and lossy.	1015
Control options.	1015
General steps.	1016
Compression.	1016
Decompression.	1017
Sequences.	1018
Multi-band buffers, color formats, and control settings - JPEG.	1018
Multi-band buffers, color formats, and control settings - JPEG2000.	1018
Application-specific markers.	1018
Controlling a JPEG compression.	1019
JPEG lossless.	1019
JPEG lossy.	1020
Restart markers.	1021
JPEG2000.	1022
Preparation of source image.	1025
Discrete wavelet transform (DWT).	1025
Quantization.	1028
Bit-plane decomposition.	1029
Arithmetic encoding.	1029
Post-processing.	1030

Improving results.	1031
Working with tables.	1033
Inquiring values in default tables.	1033
Using your own table.	1034

Chapter 24: Distributed MIL. 1035

Distributed MIL overview.	1036
Steps to create a Distributed MIL application.	1038
Basic concepts.	1039
Preparing the master and remote computers.	1040
Single or multiple cluster mode.	1041
Setting up the Distributed MIL server on the remote computers.	1041
Listening port/server connection port.	1042
Port range used by remote computers in multiple cluster mode.	1043
Firewall configuration.	1044
Licensing issues.	1045
Allocating DMIL remote systems.	1046
Allocating the systems.	1046
Setting the default system to a remote system.	1048
Using DMIL remote systems.	1049
Execution of MIL functions on remote computers.	1049
Defaults on remote computers.	1050
Files.	1050
Asynchronous calls.	1050
Multi-threading.	1051
Executing a user-defined function on the remote computer.	1051
Displays and DMIL remote systems.	1052
Best practice.	1054

Chapter 25: Multi-processing, multi-core, and multi-threading. 1057

Multi-processing.	1058
Transparent multi-core use.	1058
Setting the maximum number of cores per thread.	1059
Steps to using multi-core processing.	1061
Multi-threading.	1062
MIL and multi-threading.	1063

Chapter 26: Distribution and licensing. 1071

Distribution of MIL applications.	1072
Redistributing MIL or MIL-Lite DLL files and device drivers with your application.	1072
Redistributing directly from the MIL or MIL-Lite DVD.	1073
Redistributing using your own setup program.	1073
Interactive redistribution using your custom DVD.	1073
Silent redistribution.	1075
Uninstalling.	1076
MIL and MIL-Lite licenses.	1077
MIL provisional licenses.	1078
MIL permanent licenses.	1079
Summary of activation procedures for all MIL licenses.	1080
MIL-Lite licenses.	1080
Summary of MIL and MIL-Lite permanent licenses.	1081
Hardware license-key.	1082
Software license-key.	1082

Hiding the MIL licensing process.	1084
Generating the lock code.	1084
Getting a software license-key.	1085
Entering the software license-key.	1085
Gencode utility.	1086
Synopsis.	1086
Prototype and description.	1086
Parameters.	1086
Protecting your own MIL software application using a Matrox hardware fingerprint.	1089

Chapter 27: The MIL function development module. 1091

MIL Function Development module.	1092
Steps to create a user-defined MIL function.	1093
Basic concepts.	1098
Characteristics of a user-defined MIL function.	1099
Remote and local functions.	1100
Asynchronous and synchronous functions.	1100
Modules, opcodes, and error messages.	1101
User defined error codes.	1101
Parameter registration and return values.	1102
Return values.	1103
Master/slave dynamics on a remote system.	1105
Compilation.	1105
Executing the slave function on a remote system.	1107
Remote systems with on-board processors.	1108
Remote systems in a Distributed MIL cluster.	1109
Associating a MIL identifier with a user-defined object.	1110

Chapter 28: Using MIL with a Processing FPGA..... 1113

Using MIL with a Processing FPGA - overview.	1114
Processing an image with a Processing FPGA.	1116
Steps to develop a function that performs an operation using a Processing FPGA.	1118
Primitive function and execution of operation by PU.	1120
Source and destination image buffers.	1121
Setting and retrieving results from PU registers.	1123
Cascaded and parallel processing.	1128
Cascaded processing operation.	1128
Parallel processing operation.	1129
Issuing commands to the Processing FPGA and retrieving results.	1130
Developing a user-defined MIL function to run the primitive function.	1133
Master function and slave function and execution of operation specified by command context.	1134
Operation synchronization.	1134

Chapter 29: Using MIL under Linux. 1137

Working with Linux.	1138
Licensing.	1138
Limited and unsupported features.	1138
Board restrictions.	1138

Chapter

1

Introduction

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.

Introduction to MIL and MIL-Lite

The Matrox Imaging Library (MIL) is a hardware-independent, modular imaging library.

MIL has been designed for fast application development and ease of use. MIL is capable of running solely with the Host CPU, but can take advantage of specialized acceleration Matrox hardware if it is available and is more efficient. MIL has a completely transparent management system and handles physical objects (such as imaging boards) as virtual objects, allowing for platform-independent applications. MIL uses the notion of systems to identify boards, and more than one board can be controlled by a single application program. MIL allows multiple systems to collaborate in a single application program across a network (supported using Distributed MIL).

MIL comes in a full version, and a version that includes only core functions. The latter is referred to as MIL-Lite.

MIL-Lite allows you to grab, display, archive, transfer, and annotate images. It also allows you to perform some processing operations that are typically useful to preprocess grabbed images.

MIL-Lite can:

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

The full version of MIL contains all the functionality of MIL-Lite, a more extensive set of processing operations, and a comprehensive set of analysis operations. The image processing operations include point-to-point, statistical, spatial filtering, morphological, geometric (for example, warping and polar-to-rectangular

transformations), and Fast Fourier transform operations. The analysis operations include: measurement, blob analysis, character recognition (template-based or feature-based), correlated and geometric pattern matching, edge finding, metrology, registration, and code read/write operations.

Creating your own MIL functions

If the available MIL or MIL-Lite operations do not provide the required functionality, you can use the MIL Function Development module to create your own pseudo-MIL functions (also known as user-defined MIL functions).

The MIL help file refers to both versions of MIL simply as MIL, unless it is necessary to distinguish between them. Note that the **Availability** button of each function description indicates whether or not it is supported by MIL-Lite.

Please review the MIL Readme file for information on recent updates before using MIL.

MIL and MIL-Lite modules

The following is a summary of MIL and MIL-Lite functionality.

Functionality available in both MIL and MIL-Lite

MIL and MIL-Lite share core functionality, including application and system control, image acquisition and archiving, graph annotations, and basic image processing operations that are typically useful to preprocess grabbed images.

Image acquisition

In addition, images can be loaded from disk or acquired from the wide range of supported input devices (if hardware permits). Sequences of images can also be loaded and saved in AVI format. You can also perform a Bayer color conversion, allowing you to grab images from cameras using Bayer filters, and then convert the images into 3-band color or single-band monochrome images.

Compression

MIL and MIL-Lite allow you to compress and decompress images. MIL supports both lossy and lossless JPEG and JPEG2000 compression algorithms. MIL-Lite supports these algorithms if specialized compression Matrox hardware is available or if you have the compression/decompression runtime license package installed.

Graphics	Using the basic graphics tools in MIL and MIL-Lite, you can annotate or alter images. MIL and MIL-Lite have functions to write text; they also have graphics functions to draw rectangles, arcs, lines, and dots. The full version of MIL also has functions to draw analysis results; these functions are part of the analysis modules themselves.
Image processing	Select functions from the Image Processing module are available in MIL and MIL-Lite, allowing you to perform basic preprocessing operations on grabbed images. You can carry out basic statistical operations, as well as simple point-to-point and geometric operations. Deinterlacing is also available.
Threading	Threading is also supported on MIL and MIL-Lite, allowing you to allocate MIL selectable threads and synchronization events on a specified multi-threaded MIL system. This allows you to synchronize the execution of MIL calls on an on-board processor or a remote computer.
Distributed MIL	Distributed MIL allows multiple systems to collaborate in a single application program across a network.

Functionality available only in MIL

The full version of MIL provides more extensive processing functionality as well as analysis functionality.

Image processing capabilities	<p>The full version of the MIL Image Processing module allows you to smooth, accentuate, qualify, or modify selected features of an image. In addition, it can perform sophisticated point-to-point, statistical, spatial filtering, and morphological operations, as well as Fast Fourier transform operations, and geometric operations which include warping and polar-to-rectangular transformations.</p> <p>The MIL 3D Reconstruction module allows you to use a 3D reconstruction setup to scan an object and acquire 3D (three-dimensional) information from 2D (two-dimensional) images.</p> <p>The MIL Color Analysis module allows you to perform color-based operations such as matching, projection, and distance. You can use these operations to quickly design powerful applications that can identify colors, segment colors (supervised), detect defects, enhance color-to-grayscale conversion, separate superimposed colors, and measure the relative presence of a color in an image.</p>
-------------------------------	---

Blob analysis capabilities

The MIL Blob Analysis module allows you to identify and measure connected regions (commonly known as blobs) 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: the MIL Geometric Model Finder module and the MIL Pattern Matching module. These modules offer flexible solutions to machine vision problems such as alignment, measurement, and inspection of objects. The pattern is referred to as a model.

The MIL Geometric Model Finder module finds models using an algorithmic approach, that uses edge-based geometric features instead of a pixel-to-pixel correlation. Its capabilities include:

- Finding models through a range of angles and scales.
- Finding 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 performs normalized grayscale correlation (NGC) pattern matching. Its capabilities include finding:

- The number of occurrences of a model in a target image.
- Occurrences of a model at any angle in the target image.

Edge extraction capabilities

The MIL Edge Finder module provides powerful operations that allow you to extract different types of edges in images, such as object contours or thin curvilinear shapes. It also provides edge analysis capabilities and can measure a wide assortment of edge features, such as edge length and Feret diameter.

Registration capabilities

The MIL Registration module provides operations that calculate the transformations that optimally position images in a global coordinate system; this is referred to as image registration. The module provides operations to create a mosaic from a series of images using the resulting positions of the registration calculation. It can also convert coordinates between two of the following coordinate systems: the global coordinate system, any image's coordinate system, and the mosaic's coordinate system.

Character recognition capabilities

MIL provides two modules with character recognition capabilities: the MIL Optical Character Recognition (OCR) module, which is template-based, and the MIL String Reader module, which is feature-based.

The MIL OCR module is template-based and provides a powerful function set for reading and verifying character strings in images, providing results such as quality scores and validity flags. The module is especially designed to operate on character strings in degraded images, with up to 90° of rotation in the target string as well as on clean images with no rotation. The OCR module supports multiple strings and non-uniform spacing between characters.

The MIL String Reader module is feature-based and provides a set of powerful functions that allow you to perform robust and fast character recognition. Strings, and their associated fonts, can be quickly defined from either an image, or your system fonts (such as, Arial or Times New Roman). The read operation can then be called, and will often yield accurate results without having to alter many of the module's default values. Being feature-based, String Reader is invariant to changes in scale, aspect ratio, and contrast. It also offers considerable tolerance towards variations in perspective and angle. In addition, the String Reader module supports multiple user-defined grammar rules and multi-font definition in a single context, making applications simple to write and maintain.

Being template-based, the OCR module might perform a faster and more robust read operation than the String Reader module, provided that you have a lot of information about the target string, such as its exact size, scale, and spacing. On the other hand, if you have a minimum amount of information, and/or information that tends to vary (such as bad lighting, poor contrast, and variations in scale and perspective), String Reader is most likely the better choice.

Bar code capabilities

The MIL Code module allows you to read and write several types of 1D codes (bar codes) and 2D codes (for example, Data Matrix codes), as well as composite codes. Various encoding schemes are supported by both the reading and writing operation of the Code module. In addition, the module supports error correction schemes that allow you to both detect errors and recover data from degraded images. The module also provides functionality for verifying the quality of printed codes, to verify if the scanned code meets your quality standards.

Geometric tolerance capabilities

The MIL Metrology module measures features and validates geometric tolerances in a target image. Features and tolerances are added to a virtual template. Features can be physically measured from an image, or they can be geometrically constructed. Once features and tolerances have been added to the template, the template itself can be positioned at a new location, depending on where features have been found in the target image. Measurements are made and results are returned with a high degree of subpixel accuracy.

Calibration capabilities

The MIL Camera 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 and/or calculate 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.

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/SSE2/SSE3/SSE4).

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 (SSE/SSE2/SSE3/SSE4) accelerate performance of floating point operations and include additional integer and cacheability instructions that significantly enhance performance.

Requirements to run MIL

The library

MIL is available as a set of DLLs under the Windows and Linux operating systems. The following computer requirements should be respected to ensure that MIL operates properly.

Under Windows**Supported Windows versions**

The following Windows operating systems are supported for MIL 9.0:

- **Microsoft Windows Vista.** A minimum of 512 Mbytes of RAM is required (the recommended minimum is 1024 Mbytes). This does not include non-paged (DMA) memory needed for any of the systems.
 - **Microsoft Windows XP.** A minimum of 256 Mbytes of RAM is required (the recommended minimum is 1024 Mbytes). This does not include non-paged (DMA) memory needed for any of the systems.
- ❖ Note that **Microsoft Windows CE** is also supported for MIL 9.0, but only for Matrox smart cameras. See your Matrox smart camera manual for computer requirements.

Hardware requirements

A minimum of 100 Mbytes of free hard disk space is required to install the MIL development environment. A minimum of 40 Mbytes of free hard disk space is required to install the MIL runtime environment. In addition, a computer with a CPU that supports the SSE instruction set is required.

Software requirements

The MIL DVD includes MIL libraries that support the Microsoft Visual C++ .NET 2003/2005/2008 (unmanaged) compiler under Windows Vista/XP. The DVD also includes ActiveMIL ActiveX controls for Microsoft Visual Basic .NET 2003/2005/2008, Microsoft Visual C# .NET 2003/2005/2008, and Microsoft Visual C++ .NET 2003/2005/2008 RAD tools.

Web browser requirement for MIL help

To run MIL help, you will need Microsoft Internet Explorer version 7.0 and above.

Hibernate and standby/sleep modes

MIL does not support the hibernate and standby/sleep modes under Windows. When the Windows operating system enters any of these modes, the response of a running MIL application varies depending on the Windows flavor used:

- **Microsoft Windows XP.** The MIL application prevents the system from entering the hibernate or standby mode.
- **Microsoft Windows Vista.** The MIL application detects when the operating system enters the hibernate or sleep mode. Once the system comes out of these modes, it will prompt the user to restart the system for proper functionality of the application.

Under Linux

Supported Linux distributions

The following Linux distributions are supported for MIL 9.0:

- **Novell SUSE Linux Enterprise 10.0 (or later).** 512 Mbytes of RAM is the recommended minimum amount of RAM.
- **RedHat Enterprise Linux 5.0 (or later).** 512 Mbytes of RAM is the recommended minimum amount of RAM.

Hardware requirements

A minimum of 700 Mbytes of free hard disk space is required to install the MIL development environment. A minimum of 200 Mbytes of free hard disk space is required to install the MIL runtime environment. In addition, a computer with a CPU that supports the SSE instruction set is required.

Software requirements

GCC version 4.1 (GNU compiler collection) and G++ version 4.1 (Unix-based C++ compiler) are required for MIL 9.0.

Web browser requirement for MIL help

To run MIL help, you will need Mozilla Firefox version 2.0 and above.

Before you start

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

1. Register MIL on the Matrox website, <http://www.matrox.com/imaging/en/support/register/>. This ensures that you are on our mailing list and will receive any information on product updates and promotions.
2. Install MIL on your hard disk using the installation details (described later in this chapter). See the *MIL Readme* help file for additional documentation.
3. Review the **Default Values** tab in MilConfig to make sure that the default setup configuration matches your system configuration.

For more information on defaults, see the *Using the defaults* section in *Chapter 2: Building an application*.

4. Run our sample program *MappStart.exe*, in the examples directory, to test the installation.

Installation

In addition to your MIL DVD, you will require a hardware license-key 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 (USB port only for Linux). To redistribute your MIL applications, see the *Distribution of MIL applications* section in *Chapter 26: Distribution and licensing*.

Under Windows

To install your MIL software:

1. Attach the development hardware license-key to the parallel or USB port of your computer, depending on which hardware license key was purchased. 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 DVD in an appropriate drive. Execute the *setup.exe* program.

During installation, you will be asked a number of questions. The type of questions include:

- Whether you accept the licensing agreement.
- The drive and directory on which to install the program.
- The type of Matrox hardware installed in your computer (for example, Matrox Corona-II). Note that under Windows XP and Windows Vista, 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 define your default configuration.
- 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 Matrox graphics controllers or a Matrox DualHead graphics controller that integrates two video output controllers.

- The amount of linear non-paged memory to reserve for MIL grab buffers. The amount of reserved linear non-paged memory also establishes the amount of remaining RAM available to your operating system. Except for Windows XP (32-bit only), you can reserve up to approximately 70% of the physical RAM present in your computer.

Under Linux

For Linux, ensure that the following development tools have been installed:

- GCC (GNU compiler collection) and G++.
- LSB (Linux standard base support).
- gtk2-devel (optional, needed for the mdispgtk and mdispwindowgtk examples).
- qt3-devel (optional, needed for the mdispqt and mdispwindowqt examples).

To install your MIL software:

- Attach the development hardware license-key to the 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.
- Place the installation DVD in an appropriate drive.
- Enter the following Linux operating system commands in a Linux terminal window:
 - `>chmod +x mil-full-9.0-installer.run.`
 - `>./mil-full-9.0-installer.run` (as root or by using the `sudo` command).

During installation, a MIL group will be added to the system, and you will be prompted to add a valid user to this group. Any user in this group can change the configuration in MilConfig, modify examples, and recompile drivers. Root user does not need to be added to this group.

During installation, you will be asked a number of questions. The type of questions include:

- Whether you accept the licensing agreement.
- Whether you wish to restart your computer after installation.
- The type of Matrox hardware installed in your computer (for example, Matrox Solios).
- The digitizer and display format to define your default configuration.
- Whether by default, displayed images should be displayed in a window on the desktop or without a window on a dedicated (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 two graphics controllers.
- The amount of linear non-paged memory to reserve for MIL grab buffers.
- System users to add to the MIL admin group (who can change the configuration in MilConfig).
- Whether to uninstall previous MIL versions if found; if the answer is no, the installation will be aborted.

To uninstall MIL completely, type the following command: `>mil-uninstall` (as root or by using the `sudo` command).

General installation notes

It is important to note 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 setup program will not install MIL. The application can use the version of MIL already installed on the computer.

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 newer version cannot be installed on that computer.

After installation, read the Readme help files for last minute information. To help familiarize yourself with how MIL is actually used, use the MIL Example Launcher installed with MIL (Launcher allows you to run small MIL applications we provide and allows you to view their source code).

Note that the installation program also installs Matrox Intellicam (your frame grabber configuration program, not supported under Linux), MIL interactive utilities, and the MilConfig utility. Although the installation program does not install Matrox Inspector (a program designed to let you work with images for image capture, storage, and processing applications), the Matrox Inspector installation DVD is included with MIL. Matrox Inspector has a 30-day temporary license: for permanent use you will need a MIL permanent development license.

MIL can be run without the hardware license-key if you activate a MIL provisional license. Once activated, the provisional license allows use of MIL on your computer 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 permanent license. For more information on activating a MIL license, see the *MIL and MILLite licenses* section in *Chapter 26: Distribution and licensing*.

A MIL provisional license can only be installed once. Any attempt to tamper with your computer's calendar, before the date of expiry, will disable MIL. In that event, MIL can only be re-used once a permanent license is obtained.

Note that MIL users also receive the privilege to run (but not debug) MIL-Lite applications if Matrox boards are present.

MilConfig

MilConfig (*MILConfig.exe*), located in your *Matrox Imaging\Tools* directory, provides licensing, MIL reserved non-paged memory 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 MIL reserved non-paged memory assigned or RAM available to your system at any time by running MilConfig (alternatively, you can adjust the memory by uninstalling and reinstalling MIL). Should you require technical support, use the MilConfig utility's **Troubleshooting** tab to generate a text file that contains all the necessary system information required for basic troubleshooting; this file can then be forwarded to your Matrox technical support representative.

Auto updates

MilConfig has auto update settings which allow the user to automatically receive notice when a new version of MIL is available to download from the internet. A valid Matrox maintenance plan is required to download the new version of MIL.

Testing the installation

The MIL package includes a sample program, *MappStart.cpp*, that allows you to test the installation process and become familiar with running a MIL application. This test program allocates a MIL application, opens communication with the default target system, displays a welcoming message, pauses, and frees the system resources.

```

/*****
/*
* File name: MAppStart.cpp
*
* Synopsis: This program allocates a MIL application and system, then displays
*           a welcoming message using graphics functions. It also shows how
*           to check for errors.
*/
#include <mil.h>

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

```

```

/* Allocate a default MIL application, system, display and image. */
MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay, M_NULL, &MilImage);

/* If no allocation errors. */
if (!MappGetError(M_GLOBAL, M_NULL))
{
    /* Perform graphic operations in the display image. */
    MgraColor(M_DEFAULT, 0xF0);
    MgraFont(M_DEFAULT, M_FONT_DEFAULT_LARGE);
    MgraText(M_DEFAULT, MilImage, 160L, 230L, MIL_TEXT(" Welcome to MIL !!! "));
    MgraColor(M_DEFAULT, 0xC0);
    MgraRect(M_DEFAULT, MilImage, 100L, 150L, 530L, 340L);
    MgraRect(M_DEFAULT, MilImage, 120L, 170L, 510L, 320L);
    MgraRect(M_DEFAULT, MilImage, 140L, 190L, 490L, 300L);

    /* Print a message. */
    MosPrintf(MIL_TEXT("\nSYSTEM ALLOCATION:\n"));
    MosPrintf(MIL_TEXT("-----\n\n"));
    MosPrintf(MIL_TEXT("System allocation successful.\n\n"));
    MosPrintf(MIL_TEXT("      \"Welcome to MIL !!!\"\n\n"));
}
else
    MosPrintf(MIL_TEXT("System allocation error !\n\n"));

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

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

return 0;
}

```

Communicating properly

During application development, you can use *MappStart.cpp* 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 or a grab example.

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\MilLib* directory.

- ❖ Note that MIL functions run at a slower speed during debug mode. Therefore, once you are done debugging your application, you should compile in release mode to achieve maximum processing speed.

The following is a list of MIL library files (for the Windows XP/Vista operating systems) that can be included in your program. Note that you must include *mil.lib* to compile your application program.

Windows library files	Description
<code>mil.lib</code>	Core library (includes all the functionality available in MIL-Lite).
<code>mil3dmap.lib</code>	3D Reconstruction module library.
<code>milblob.lib</code>	Blob Analysis module library.
<code>milcal.lib</code>	Camera Calibration module library.
<code>milcolor.lib</code>	Color Analysis module library.
<code>milcode.lib</code>	Code module library.
<code>miledge.lib</code>	Edge Finder and Analysis module library.
<code>milim.lib</code>	Image Processing module library.
<code>milmeas.lib</code>	Measurement module library.
<code>milmetrol.lib</code>	Metrology module library.
<code>milmod.lib</code>	Geometric Model Finder module library.
<code>milocr.lib</code>	Character Recognition module library.
<code>milpat.lib</code>	NGC Pattern Matching module library.
<code>milreg.lib</code>	Registration module library.
<code>milstr.lib</code>	String Reader module library.

The following is a list of MIL library files (for the Linux operating system) that can be included in your program. Note that you must include *libmil.so* to compile your application program.

Linux library files	Description
libmil.so	Core library (includes all the functionality available in MIL-Lite).
libmil3dmap.so	3D Reconstruction module library.
libmilblob.so	Blob Analysis module library.
libmilcal.so	Calibration module library.
libmilcolor.so	Color Analysis module library.
libmilcode.so	Code module library.
libmiledge.so	Edge Finder and Analysis module library.
libmilim.so	Image Processing module library.
libmilmeas.so	Measurement module library.
libmilmetrol.so	Metrology module library.
libmilmod.so	Geometric Model Finder module library.
libmilocr.so	Character Recognition module library.
libmilpat.so	NGC Pattern Matching module library.
libmilreg.so	Registration module library.
libmilstr.so	String Reader module library.

Note that you need to recompile your MIL applications after a major upgrade of MIL. If it is only a minor version upgrade (for example, upgrading from MIL 9.0 to MIL 9.1), it is usually not necessary. Consult the *What's new readme* to check for any exceptions to this general rule.

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, as well as the required MIL header file and libraries, it is expected that you read *Chapter 2: Building an application*.

The next set of chapters, up to and including *Chapter 15: Metrology*, deal with enhancement, transformation and analysis operations. It is expected that you read only the chapters that you need; for the chapters on image processing, 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.

Besides Chapters 1 and 2, this user guide has been written for using MIL under Windows XP/Vista. Using MIL under Linux is very similar; refer to *Chapter 29: Using MIL under Linux* for a list of the differences.

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. Finally, in general, Host refers to the CPU in one's computer, while system refers to your Matrox imaging board and its associated resources.

Command descriptions

Descriptions of the individual functions are found in the MIL Reference.

Examples

Throughout this manual, examples have been provided to simplify concepts and get you started quickly. The source listing of most examples can be found on disk in the *Matrox Imaging\Mi\Examples* directory. To run and edit these examples, you can use the example launcher, *ExampleLauncher.exe*, located in the *Matrox Imaging\Tools* directory.

- ❖ Note that some examples will display images in grayscale even if the images are grabbed with a color camera. This is due to the fact that those examples perform statistical and/or analysis operations that do not support color processing.

MIL's examples offer many advantages:

- The *MappStart* example is very simple, but it allows you to quickly test MIL's installation and become familiar with the basics behind running a MIL application. The *MappStart* example allocates an application, opens communication with the default system, displays a welcoming message, pauses, and then closes communication with the system. *MappStart* also demonstrates error handling capabilities.
- The examples use your default settings which have been specified at installation; therefore, very little time must be spent configuring your programming environment. However, if at some point you want to change the default settings, or are experiencing problems running the examples, your default settings can be changed on the **Default Values** tab of the MilConfig utility, (*MILConfig.exe*). This utility is located in the *\Matrox Imaging\Tools* directory. You can also change the default settings by clicking the **Change Defaults** button in the example launcher, which will take you directly to the **Default Values** tab of the MilConfig utility.

Note that some systems cannot run some of the examples because they do not have the hardware capability or enough memory. You should skip these examples or modify them.

Chapter

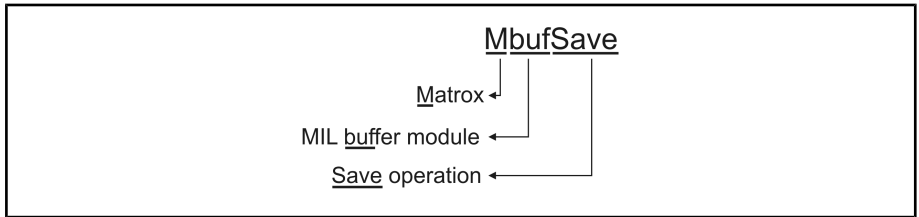
2

Building an application

This chapter shows you the basics in building an application.

Organization of the MIL modules

All functions in MIL are grouped into modules. Functions are named based on their module and functionality. Each function name is prefixed with the letter M for Matrox, followed by a few letters representing the function's module name, and ends with a word that describes the function's purpose. For example, **MbufSave()** saves a data buffer in a file.



Similar MIL functions in different modules

Functions are grouped into modules according to their main functionality. For example, the Buffer module contains all the functions in MIL that are specific to allocating, controlling, inquiring about, importing, and saving buffers. All MIL modules are based on the same structure and you can find similar functions in different modules. These functions differ depending on the specifications of the module, but they all perform the same basic task.

The following specifies MIL functions that are available in most MIL modules. For more information on a specific function, see the MIL Reference.

- **M...Alloc.** Allocates the specified MIL object (for example, buffer, display, or context) and its required resources. You must allocate these before you use them. Once you are finished using them, you should release them using **M...Free**.
- **M...Free.** Frees an allocated MIL object and its resources.
- **M...Control.** Sets the controls for the specified MIL object (for example, buffer, context, or element(s) that are contained in the context). An **M...Control** function allows you to customize how the operations of that specific module are performed. Note that most control types have a default setting.
- **M...Inquire.** Inquires about a specified setting. Most settings that can be set using **M...Control** can be inquired using the corresponding **M...Inquire** function.

- **M...Save.** Saves the specified MIL object (for example, the specified context or buffer) to a specified file.
- **M...Restore/ M...Load.** Restores the specified MIL object (for example, the specified context or buffer) from a specified file.
- **M...HookFunction.** Allows you to attach or detach a user-defined function to a specified event. Once an event-handler function is defined and hooked to an event, it is automatically called when the event occurs. The event-handler function is also referred to as a hook-handler function.
- **M...GetHookInfo.** Retrieves information about the event that caused the hook-handler function to be called. This function should only be called within the scope of a hook-handler function call.

Processing and analysis modules

The following specifies MIL functions that are common in most MIL processing and analysis modules. For more information on a specific function, see the MIL Reference.

- **M...AllocResult.** Allocates the specified result buffer. Result buffers store results obtained from the module's **M...Calculate**, **M...Read**, or **M...Find** operations. When the result buffer is no longer required, release its memory, using **M...Free**.
- **M...Calculate/M...Read/M...Find.** Perform the operation(s) specific to the module, respecting the controls specified using **M...Control**. The results of the operations are stored in the specified result buffer.
- **M...GetResult.** Retrieves the results of the specified type from the specified result buffer.
- **M...Draw.** Draws specific source information or results into the specified destination buffer.
- **M...Stream.** Loads, restores, or saves the specified MIL object (for example, context) from/to a file or memory stream.

Interactive functionality

Under Windows XP/Vista, many MIL functions can launch dialog boxes that allow you to specify certain information, test this information using specified test images, and/or view results interactively at runtime. The most common functions that support dialog boxes are listed below:

- **M...Save** or **M...Export** with **M_INTERACTIVE**. Opens the **File Save As** dialog box from which you can interactively specify the drive, directory, and name of the file.
- **M...Restore** or **M...Import** with **M_INTERACTIVE**. Opens the **File Open** dialog box from which you can interactively specify the drive, directory, and name of the file.
- **M...GetResult** with **M_INTERACTIVE**. Opens a dialog box containing a spreadsheet that displays the results stored in the result buffer. All the result types are displayed in separate columns. If there are queued results, only the queued results will be displayed.
- **M...Stream** with **M_INTERACTIVE**. Opens a dialog box from which you can interactively specify the drive, directory, and name of the file, when the **StreamType** parameter is set to **M_FILE**.
- **M...Control** with **M_INTERACTIVE**. Opens a dialog box that allows you to edit the control types interactively. For **M_CONTEXT**, **M_INTERACTIVE** opens a dialog box that allows you to edit all the control types of the specified context interactively. For an element in the context, **M_INTERACTIVE** opens a dialog box that allows you to edit the control types of the specified element. Some dialog boxes provide a tab that allows you to test the settings on images interactively.
- **M...Inquire** with **M_INTERACTIVE**. Opens a read-only dialog box that displays the settings of the inquire types. For **M_CONTEXT**, **M_INTERACTIVE** opens a read-only dialog box that displays the setting of all the inquire types of the specified context. For an element of the context, **M_INTERACTIVE** opens a read-only dialog box that displays the setting of the inquire types of the specified element.

For a full list of all the functions that support dialog boxes, type **M_INTERACTIVE** into the edit field of the **Search** tab, in the left pane of this help file.

- ❖ Note that Linux does not support Interactive Mode, that is, Linux does not support **M_INTERACTIVE**.

Allocating an application and its systems

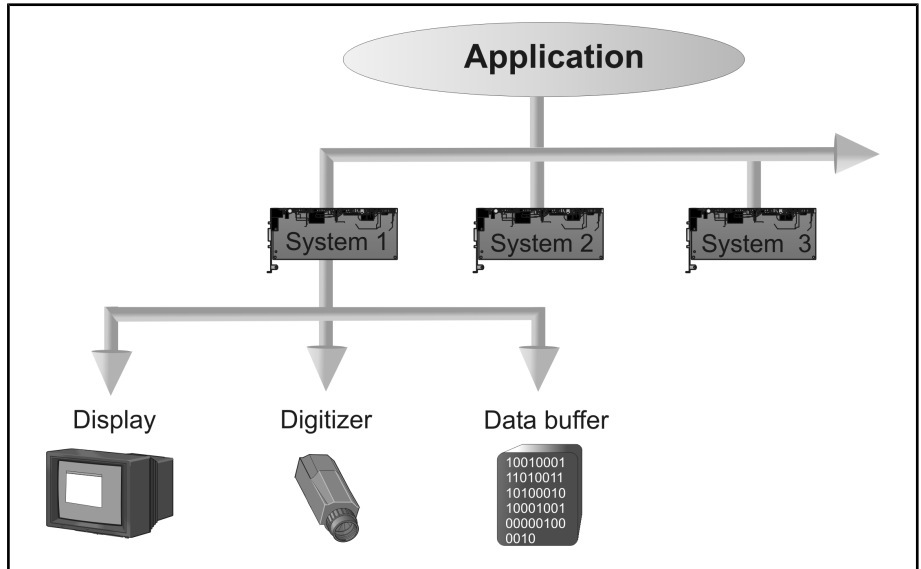
Mandatory allocations for an application

Initialization

At the beginning of each application, you must:

1. Allocate a MIL application using **MappAlloc()**. This creates a control and execution environment for your imaging application.
2. Allocate a MIL system using **MsysAlloc()**. A MIL system typically refers to a set of hardware components capable of grabbing, storing, processing, and/or displaying images. There are two types of MIL systems:
 - **Board-type system.** Consists of a Matrox imaging board (or third-party board that is compatible with MIL), the Host CPU and memory, and any available graphics controller. You will need to allocate a MIL system for each Matrox imaging board (or third-party board) that you want to use, whether the board is installed on the master or remote computer.
 - **Host-type system.** Consists of the Host CPU and memory, and any available graphics controller. You can allocate a Host-type system on the main Host computer or remote computer when a Matrox imaging board (or third party board) is not available or is not required.

When you allocate a system, MIL opens communication channels and initializes the system (or hardware resources). Once Host communication has been established with a system, you can allocate its memory resources, display, and input capabilities. When a remote computer is specified during system allocation, the resources of the specified computer are made available: its specified Matrox imaging board (or third-party board), its CPU, and its memory.



At the end of each application, free each system using **MsysFree()**, and then free the application using **MappFree()**.

Board-type versus Host-type systems

On a computer without a Matrox imaging board (or supported third-party board), you can allocate a Host-type system which can store, process, and display images, but cannot grab images from a camera.

On a computer with a Matrox imaging board (or supported third-party board), you can allocate a Host-type system and/or a board-type system. If you allocate a Host-type system, the system will be able to store, process, and display images. If you allocate a board-type system, the system will be able to store, process, and display images, and provide additional functionality depending on the installed board.

The board-type systems that allow the use of third-party (non-Matrox) boards are:

- **IEEE 1394b IIDC system.** Supports the Matrox 4Sight-M 1394b adapter board, Matrox Concord F-series adapter boards, and MIL-compatible third-party 1394 adapter boards. IEEE 1394b IIDC systems allow image grabs from IEEE 1394 IIDC cameras.
- **GigE Vision system.** Supports Matrox Concord G-series network interface cards (NICs), and MIL-compatible third-party NICs. GigE Vision systems allow image grabs from GigE Vision cameras.
- **GPU system.** Supports most display boards that support DirectX 9.0 or later. Accelerates image processing operations.

❖ Note, systems can have many data buffers, displays, and digitizers.

The functionality provided by Matrox imaging boards (and third-party boards) varies from one board to the next. Some boards only provide frame-grabbing functionality (for example, Matrox CronosPlus), while others provide grabbing and processing functionality (for example, Matrox Odyssey).

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

Multiple systems

You can allocate more than one system per application and then use their identifiers to access their devices and memory resources. Note that, unless your Matrox imaging board supports multi-processing, two applications running at the same time cannot share a board. Therefore, in general, only one application can allocate a system for a board at a given time.

Processing can be done between many systems. 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.

Distributed MIL

Multiple MIL systems can be used together within a MIL application. Using Distributed MIL, multiple MIL systems can collaborate in a MIL application across a network (for example, a local area network, or a wide area network such as the internet). A group of MIL systems working together in an application is called a cluster. Clusters are managed by a MIL application running on a master computer. Systems allocated on remote computers are referred to as DMIL remote systems.

To develop or run a Distributed MIL application, MIL/MIL-Lite must be installed on the master and slave computers. When developing a Distributed MIL application that uses functionality only available in the full version of MIL, the master computer requires a MIL development license; the remote slave computer(s) only require appropriate runtime licenses for Distributed MIL and the MIL modules that they actually use. When only running such an application, all computers in the cluster require appropriate runtime licenses for Distributed MIL and the MIL modules that they actually use; sending a command to another computer is not using a module so no specific license is needed in this case. When developing or running a Distributed MIL application that uses only MIL-Lite functionality, the master and remote computers require a Distributed MIL supplemental license.

For more information on Distributed MIL, see *Chapter 24: Distributed MIL*.

Allocating and displaying an image buffer

After you have allocated an application along with any required system(s), you are ready to grab and display an image. This section covers how to allocate and display an image buffer. For the basics required to start grabbing, see the *Grabbing images* section later in this chapter.

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. To store color image data, MIL uses the concept of bands; your buffer needs one band per color component of your image. For example, an RGB image needs a buffer with three bands.

You can allocate a monochrome image buffer using **MbufAlloc2d()**. This function requires that you specify the following image buffer attributes:

- The system on which to allocate the buffer.
- The image buffer's size in X and Y.
- The depth of the buffer: 1-, 8-, 16-, or 32-bit.
- 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 (or analysis) 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.

You can allocate a color image buffer using **MbufAllocColor()**. This function requires that you specify the number of color bands in addition to the attributes listed above. Note that although most cameras acquire color images, some analysis modules only operate on monochrome images (MIL cannot perform statistical analysis or image analysis operations on color image buffers). In these cases, you must grab into a color buffer, convert this image to a monochrome image using **MimConvert()**, and store it in a one-band monochrome buffer.

To operate on a specific band or region of an image buffer, you can use **MbufChild...**

For more information on buffers and other buffer attributes, see *Chapter 18: Specifying and managing your data buffers*.

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()**. If you have allocated a displayable buffer (**M_DISP**), select it to this display, using **MdispSelect()**; you can use the same function to stop displaying it.

❖ Note that the image buffer and the display should be allocated on the same system.

The following example code shows you how to allocate and display an image buffer.

```
/* Allocate a system for a Matrox Morphis board. Use a different */  
/* value or "M_DEFAULT" for other types of systems.             */  
MappAlloc (M_SETUP, &MilApplication);  
MsysAlloc (M_SYSTEM_MORPHIS, M_DEFAULT, &MilSystem);  
MdispAlloc (MilSystem, M_DEFAULT, MIL_TEXT("M_DEFAULT"), M_WINDOWED, &MilDisplay);  
  
/* Allocate an image buffer with the default dimensions to do graphics. */  
MbufAlloc2d (MilSystem, 512,512, 8+M_UNSIGNED, M_IMAGE+M_DISP+M_GRAB, &MilImage);  
  
/* Draw and display a circle */  
MbufClear (MilImage, 0L);  
MgraArcFill (M_DEFAULT, MilImage, 256L, 240L, 100L, 100L, 0.0, 360.0);  
MgraText (M_DEFAULT, MilImage, 238L, 234L, MIL_TEXT(" MIL "));  
  
/* Display the image buffer. */
```

```

MdispSelect (MilDisplay, MilImage);

/* Print a message. */
MosPrintf (MIL_TEXT("A circle was drawn in the displayed image buffer.\n"));

/* Free all allocations */
MbufFree (MilImage);
MdispFree (MilDisplay);
MsysFree (MilSystem);
MappFree (MilApplication);

```

- ❖ For more information on displaying an image, see *Chapter 20: Displaying an image*. To allocate a display on a DMIL remote system, see *Chapter 24: Distributed MIL*.

Grabbing images

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 a video source (typically a video camera). To allocate a digitizer, use **MdigAlloc()** and specify a digitizer configuration format (DCF) that is compatible with the output format of your camera. This configures the required acquisition paths (camera interface) on the frame grabber so that they can accept input from the camera. With a call to **MdigGrab()**, you can then grab into a grab image buffer (an image buffer with an **M_GRAB** attribute).

Allocate the grab image buffer on the same system, and of the same data format, as the digitizer. For color cameras, use color image buffers.

By default, when **MdigGrab()** is issued, it grabs a complete frame of data, as defined by the DCF used to allocate the digitizer and not by the size of the image buffer. For more details on how to grab with the digitizer, see *Chapter 22: 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 a specified buffer until you issue **MdigHalt()**. Since you are grabbing continuously into the same buffer, this function is only really useful if you select the buffer to a display, using **MdispSelect()**, prior to starting the continuous grab, so that you can view what is being grabbed.

If your camera supports remote lens adjustment, you can use **MdigFocus()** to automatically adjust the lens motor of the camera to achieve optimum focus in your images. For more information, see the *Autofocusing* section in *Chapter 22: Grabbing with your digitizer*.

Sequential grabbing

Typically, you need to grab and process images continuously. To ensure that no frames are missed, the **MdigProcess()** function allows you to grab images sequentially, store them into a list of image buffers, and process them as they are being grabbed.

MdigProcess() has the same requirements as **MdigGrab()**, except **MdigProcess()** accepts an array of grab image buffers and can cause a user-defined function to be called after an image has been grabbed into any of the image buffers.

MdigProcess() can round-robin through the buffers; that is, after it has finished grabbing into the last buffer in the array, it begins grabbing again into the first buffer.

For more information and examples, see the *Multiple buffering* subsection in the *Grabbing and processing* section in *Chapter 22: Grabbing with your digitizer*.

An example

The following example grabs a single image from the camera.

```

/*****
/*
/* File name: msimple.cpp
/*
/* Synopsis: This program allocates a displayable image buffer, clears its contents,
/*           draws a filled circle and some text and then displays it.
/*
#include <mil.h>
#include <conio.h>

```

```

#include <stdlib.h>

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

    /* Allocate a system for a Matrox Morphis board. Use a different */
    /* value or "M_DEFAULT" for other types of systems. */
    MappAlloc(M_SETUP, &MilApplication);
    MsysAlloc(M_SYSTEM_MORPHIS, M_DEFAULT, M_DEFAULT, &MilSystem);
    MdispAlloc(MilSystem, M_DEFAULT, MIL_TEXT("M_DEFAULT"), M_WINDOWED, &MilDisplay);

    /* Allocate a digitizer for RS170 camera. Use a different */
    /* value or "M_DEFAULT" for other types of cameras. */
    MdigAlloc(MilSystem, M_DEV0, MIL_TEXT("M_RS170"), M_DEFAULT, &MilDigitizer);

    /* Allocate an image buffer with the default dimensions to do graphics. */
    MbufAlloc2d(MilSystem, 512, 512, 8+M_UNSIGNED, M_IMAGE+M_DISP+M_GRAB, &MilImage);

    /* Draw and display a circle */
    MbufClear(MilImage, 0L);
    MgraArcFill(M_DEFAULT, MilImage, 256L, 240L, 100L, 100L, 0.0, 360.0);
    MgraText(M_DEFAULT, MilImage, 238L, 234L, MIL_TEXT(" MIL "));

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

    /* Print a message. */
    MosPrintf(MIL_TEXT("A circle was drawn in the displayed image buffer.\n"));
    MosPrintf(MIL_TEXT("Press <Enter> to grab an image.\n"));
    MosGetchar();

    /*Grab an image into MIL image buffer*/
    MdigGrab(MilDigitizer, MilImage);

    /* Print a message. */
    MosPrintf(MIL_TEXT("An image was grabbed in the displayed image buffer.\n"));
    MosPrintf(MIL_TEXT("Press <Enter> to end.\n"));
    MosGetchar();

    /* Free all allocations */
    MbufFree(MilImage);
    MdigFree(MilDigitizer);
    MdispFree(MilDisplay);
    MsysFree(MilSystem);
    MappFree(MilApplication);

    return 0;
}

```

Dealing with color

MIL supports grabbing, displaying, accessing, and processing color images.

- ❖ MIL-Lite does not support processing.

MIL can represent an object in color with a single color buffer, allocated with **MbufAllocColor()**.

Grabbing

You grab from a color video source (typically a camera) into a color image buffer, as you would into a grayscale image buffer, by calling **MdigGrab()** or **MdigGrabContinuous()**.

Before performing a color grab, you must allocate a digitizer, using **MdigAlloc()** (or **MappAllocDefault()**), specifying a digitizer configuration format (DCF) appropriate for your color camera. 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 acquisition path of the digitizer, 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 can update your defaults using **MilConfig**.

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 digitizer, using **MdigLut()**.

You can associate either a single-band LUT buffer or a LUT buffer that has the same number of color bands as the digitizer. If you associate a single-band LUT buffer with the digitizer, each component of the digitizer LUT is loaded with the same data. If you associate a multi-band LUT buffer with the digitizer, each component of the digitizer LUT is loaded with the data from corresponding color band of the LUT buffer.

To disassociate the LUT buffer from the digitizer, you need to re-associate the default LUT with the digitizer, using **MdigLut()** with **M_DEFAULT**.

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**), and then select it for display, using **MdispSelect()**. To stop displaying the image buffer and leave the display blank, use **MdispSelect()** with **M_NULL**.

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.

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).

Managing 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()** function reference descriptions for more details.

Note, all the MIL data allocation, access, and generation (**Mbuf...** and **MgenLut...**) functions can handle color image buffers.

Using the defaults

During the installation of your MIL software, you are asked a number of questions (such as, the type of Matrox hardware installed in your computer), so that the installer knows what to install. This information is also used to define the default settings that are configurable. You can change these default settings later, using MilConfig, the MIL configuration utility.

Using MilConfig to change your default settings

Review the **Default Values** tab in MilConfig to make sure that your computer's default settings match your application's required default settings. If they don't match, you can use MilConfig to change them. For example, you can use MilConfig to change the default system, the default image buffer size and attributes, the default display settings, and the default digitizer's DCF.

Using your defaults

You can use the **MappAllocDefault()** macro to allocate a MIL application and your default system. On this system, **MappAllocDefault()** also allows you to allocate your default image buffer, default display, and default digitizer. Alternatively, you can use **MappAlloc()**, **MsysAlloc()**, **MbufAllocColor()**, **MdispAlloc()**, and **MdigAlloc()** with **M_DEFAULT** to allocate these defaults. By allocating these using their default settings, you create a more portable, device-independent application, since these settings are not hard-coded in your application, and are determined when your client installs the MIL software.

If the MIL driver for a Matrox frame grabber has been installed, the size, number of bands, and attributes for the default image buffer will be established based on the specified DCF for the default digitizer. If a monochrome DCF is specified as the default, a single-band monochrome buffer is allocated as the default image buffer. If a color DCF is specified, a multi-band color buffer is allocated as the default image buffer. The default buffer size is the same as that of the image capture-size specified in the DCF.

When allocating both the default image buffer and the default display using **MappAllocDefault()**, the image buffer is given a displayable attribute, cleared, and selected to the display.

- ❖ Note that although there are advantages to using the default settings, it can make debugging more difficult since the settings for the defaults are not determined within your application.

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 function or a sequence of functions, using **MappGetError()**. In addition, you can assign a user-defined function to handle the event of a MIL error using **MappHookFunction()**.

For an example that uses **MappGetError()** to determine if the allocation of the default application, default system, default display, and default image buffer is a success, see *MAppStart.cpp* in the *Testing the installation* section in *Chapter 1: Introduction*.

Multiple systems

There is no limit to the number of systems an application can access; it is limited only by the physical devices present in your computer or accessible from your computer. To use multiple Matrox imaging boards, you have to allocate a MIL system for each board. If necessary, you can use **MappInquire()** with **M_INSTALLED_SYSTEM_DESCRIPTOR + n** to select the system to use at runtime.

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 mappable memory, for example, on the Host or any MIL system; no memory is actually allocated to this

newly created buffer. This technique 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**. For more information about creating data buffers, see

Chapter 18: Specifying and managing your 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.

Windowed displays from different systems on the same computer will automatically display together on the same screen in different windows.

MIL custom data types, extensions, and portability functions

MIL features data types and functions that can improve your application's portability and operating system independence.

MIL custom data types

In addition to the standard data types available in C and C++, MIL introduces the following predefined data types, to ensure better compatibility and portability of MIL applications under different operating systems:

- *MIL_ID*. Specifies a value that is a MIL identifier. The *MIL_ID* data type is used to identify objects (for example, buffers, systems, and contexts) which are allocated using MIL within an application.
- *MIL_INT*. Specifies a value that is an integer. The *MIL_INT* data type is used as a supplement to the more conventional *long* and *int* data types. The *MIL_INT* data type is a 32-bit value on a 32-bit operating system, and a 64-bit value on a 64-bit operating system.

- *MIL_UINT*. Specifies a value that is an unsigned integer. Its data depth is the same as that of the *MIL_INT* data type.
- *MIL_DOUBLE*. Specifies a value that is a double precision number (floating-point).

In addition to the *MIL_INT* and *MIL_UINT* data types, MIL also has signed and unsigned, fixed length versions of these data types: *MIL_INT8*, *MIL_UINT8*, *MIL_INT16*, *MIL_UINT16*, *MIL_INT32*, *MIL_UINT32*, *MIL_INT64*, and *MIL_UINT64*.

MIL also introduces the following predefined data types, to ensure better compatibility and portability of applications regardless of whether they are compiled for ASCII or Unicode character sets:

- *MIL_TEXT_CHAR*. Specifies an array of characters. The *MIL_TEXT_CHAR* data type is used as a replacement for the more conventional *char* data type.
 - *MIL_TEXT_PTR*. Specifies a pointer to an array of characters. The *MIL_TEXT_PTR* data type is used as a replacement for the more conventional *char** data type.
 - *MIL_CONST_TEXT_PTR*. Specifies a pointer to an array of constant characters. The data type is used as a replacement for the more conventional *const char** data type.
- ❖ There is a macro called **MIL_TEXT** which converts the provided string to the proper encoding (ASCII or Unicode) for the application. Its syntax is:
MIL_TEXT("Text to be converted.").

M_MIL_USE_SAFE_TYPE extension

When using void pointers, it is possible to make a mistake and pass a value with the wrong data type. These errors can cause unexpected behavior such as: stack corruption, array overflow, uninitialized returned data, and segmentation faults. MIL includes the **M_MIL_USE_SAFE_TYPE** extension to help you find these errors.

- ❖ Note that this extension is not supported under Linux.

When the **M_MIL_USE_SAFE_TYPE** extension is enabled, an error message is returned if the data type of the variable whose address is passed to a void pointer parameter does not match the expected data type. By default, the **M_MIL_USE_SAFE_TYPE** extension is enabled when compiling in debug mode.

- ❖ Note that the **M_MIL_USE_SAFE_TYPE** extension is only available if you are using a C++ compiler under Windows.

You might want to skip the verification performed by the **M_MIL_USE_SAFE_TYPE** extension if:

- The data type of a value is unknown at the time the function is called. This typically occurs when the MIL function is wrapped inside another function in the application. For example:

```
MIL_INT UserWrapperAroundMbufInquire(MIL_ID BufId, MIL_INT InquireType, void *UserVarPtr){
return MbufInquire(BufId, InquireType, UserVarPtr);
}
```

- The data type of the value passed to a function is not one of the expected data types accepted by a function (for example, when a custom data type is defined in the application).

```
MIL_INT8 *Data = malloc(sizeof(MIL_DOUBLE) / sizeof(MIL_INT8));
MmodGetResult(ResultId, M_GENERAL, M_NUMBER, Data);
Number = *((double*)Data);
free(Data);
```

To disable the **M_MIL_USE_SAFE_TYPE** extension entirely, include the **#define M_MIL_USE_SAFE_TYPE 0** statement before the **#include <mil.h>** statement in the application code. When the extension is excluded, all MIL functions are called without any data type verification.

Portability functions

In addition to the functions available in the C/C++ standard library, MIL introduces the following functions to ensure better compatibility and portability of applications regardless of whether they are compiled for ASCII or Unicode character sets:

Function in the C/C++ standard library	Portable MIL version of the function	Description
abs	MIL_INT MosAbs(MIL_INT n)	Calculates the absolute value.
getch	MIL_INT MosGetch	Gets a character from the console.
getchar	MIL_INT MosGetchar	Reads a character from standard input.
kbhit	MIL_INT MosKbhit	Checks the console for keyboard input.
main	MIL_INT MosMain	Specifies the first function to execute.
printf	MIL_INT MosPrintf(MIL_TEXT_PTR format, ...)	Prints formatted output to the console.
sleep	MIL_INT MosSleep(MIL_INT waitTime)	Makes the application sleep for a specified time.
sprintf	MIL_INT MosSprintf(MIL_TEXT_PTR buffer, MIL_INT format, MIL_TEXT_PTR locale, ...)	Puts formatted data in a string.
strcat	MIL_INT MosStrcat(MIL_TEXT_PTR strDestination, MIL_INT size, MIL_TEXT_PTR strSource)	Joins strings.
strcmp	MIL_INT MosStrcmp(MIL_TEXT_PTR string1, MIL_TEXT_PTR string2)	Compares strings.
strcpy	MIL_INT MosStrcpy(MIL_TEXT_PTR strDestination, MIL_INT size, MIL_TEXT_PTR strSource)	Copies characters of one string to another.
strlen	MIL_INT MosStrlen(MIL_TEXT_PTR str)	Gets the length of a string.
strlwr	MIL_INT MosStrlwr(MIL_TEXT_PTR str)	Converts a string to lowercase.
strupr	MIL_INT MosStrupr(MIL_TEXT_PTR str)	Converts a string to uppercase.

For more information on the above-mentioned functions of the C/C++ standard library, refer to MSDN.

Porting a 32-bit MIL application to MIL 64-bit

As of MIL 9.0, MIL supports 64-bit operating systems. For more details about supported operating systems, refer to the *Requirements to run MIL* section in *Chapter 1: Introduction*.

When porting an application to MIL 64-bit, there are certain modifications that must be made to your application's code. This section explains these changes and the circumstances under which they are required.

Modified functions

In the 64-bit version of MIL, there are actually two versions of the following functions:

- **M...Control.**
- **MgraArc().**
- **MgraArcFill().**
- **MgraDot().**
- **MgraLine().**
- **MgraRect().**
- **MgraRectFill().**
- **MgraText().**
- **MmetSetPosition().**
- **MmodDefine().**
- **MregSetLocation().**

The two versions of these functions differ in the expected data types of their parameters: one version has a *Double* suffix, and one version has an *Int64* suffix (for example, **MblobControlDouble** and **MblobControlInt64**). The *Double* version of these functions should be used when the relevant parameter (typically the **ControlValue** parameter) requires a *MIL_DOUBLE* value and the *Int64* version should be used when a *MIL_INT* value is required.

In C++ (*.cpp), the original functions are available as inline functions which will automatically call the appropriate version of the function depending on the type of parameter received. No change to your source code should be required if you are compiling your 64-bit MIL application in C++.

In C (*.c), the original functions are actually macros that call the most typically required version of the function. This means that if you are compiling your MIL application in C, only minor modifications to your code will be necessary. Compiler warnings will be produced if you pass a function a double value where an integer is expected; in this case, you will need to explicitly change this function call to the proper version of the function.

Retrieving a pointer to a modified function

Advanced users might want to retrieve a pointer to one of the modified functions. In this case, only the *Double* or *Int64* version should be used, since the original function is actually a macro or an overloaded function. For example, you must retrieve a pointer to **MmetSetPositionDouble** or **MmetSetPositionInt64**, and not **MmetSetPosition()**.

Note that when retrieving pointers to functions, the following are also affected: **MimArith()**, **MimArithMultiple()**, and **MimStat()**. For these functions, a *Double* version also exists; you must use it instead of the original.

Void pointers

Most MIL functions have parameters of a fixed data type. However, there are some functions, predominantly **M...Inquire** and **M...GetResult** functions, where one (or more) of their parameters is a void pointer. The expected data type for these void pointer parameters is determined in one of two ways:

- The data type is intrinsically determined by the setting of another parameter.

For example, consider the **MbufInquire()** function. When the **InquireType** parameter of this function is set to **M_ALLOCATION_OVERSCAN_SIZE**, a *MIL_INT* value is returned, and, therefore, the **UserVarPtr** parameter expects the address of a variable of type *MIL_INT*. However, when the **InquireType** parameter is set to **M_ANCESTOR_ID**, the **UserVarPtr** parameter expects the address of a variable of type *MIL_ID*.

```
// Two calls to the same function where different data types are expected.
//
// Value1 should point to a MIL_INT
MbufInquire(BufId, M_ALLOCATION_OVERSCAN_SIZE, &Value1);
// Value2 should point to a MIL_ID
MbufInquire(BufId, M_ANCESTOR_ID, &Value2);
```

- The data type is determined by a modifier (for example, **M_TYPE_DOUBLE**) added to the setting of another parameter.

For example, consider the **MblobInquire()** function. The **InquireType** parameter specifies the blob feature about which to inquire. The **UserVarPtr** parameter specifies the address at which to write the returned value. The **UserVarPtr** parameter is a void pointer, and by default requires that you pass it the address of a *MIL_INT* value. However, if **M_TYPE_DOUBLE** is added to the setting of **InquireType**, the **UserVarPtr** parameter expects the address of a variable of type *MIL_DOUBLE*.

```
// Two calls to the same function where different data types are expected.
//
/* Value3 should point to a MIL_INT */
MblobInquire(BlobResId, M_DRAW_RELATIVE_ORIGIN_X, &Value3);
/* Value4 should point to a double */
MblobInquire(BlobResId, M_DRAW_RELATIVE_ORIGIN_X+M_TYPE_DOUBLE, &Value4);
```

In these cases, for the 64-bit version of MIL, you will have to use the required custom data types documented as of MIL 9.0. For example, the following code for the 32-bit version of MIL:

```
long SizeX;
MbufInquire(BufId, M_SIZE_X, &SizeX);
```

Should be changed for the 64-bit version of MIL as follows:

```
MIL_INT SizeX;
MbufInquire(BufId, M_SIZE_X, &SizeX);
```

M_MIL_USE_SAFE_TYPE extension

When using void pointers, it is possible to make a mistake and pass a value with the wrong data type. These errors can cause unexpected behavior such as: stack corruption, array overflow, uninitialized returned data, and segmentation faults. To ease the transition from 32-bit to 64-bit, use the MIL

M_MIL_USE_SAFE_TYPE extension. It will help you find calls to functions whose prototypes have been modified. By default, the **M_MIL_USE_SAFE_TYPE** extension is enabled when compiling in debug mode.

For more information about the **M_MIL_USE_SAFE_TYPE** extension, see the *MIL custom data types, extensions, and portability functions* section earlier in this chapter.

❖ Note that this extension is not supported under Linux.

Project processor definitions

When porting an application to MIL 64-bit, you must verify that your project processor definitions (PPDs) include WIN64 and _AMD64 (these PPDs appear in the Properties page of your application). Your application will not compile if it is missing WIN64 and _AMD64.

❖ Not supported under Linux.

MIL under Linux

Using MIL under Linux is very similar to using it under Windows. The remaining chapters of this User Guide describe how to use MIL under Windows. For a list of the differences of using MIL under Linux, see the *Working with Linux* section in *Chapter 29: Using MIL under Linux*.

The MIL Reference contains information that applies to both the Linux and Windows operating systems. To view only Linux-related information, click on the **Customize help** button at the top of any reference page. In the presented page, you can select Linux as you operating system.

How to create a portable application

You can create a more portable, device-independent, and operating system-independent application if you use the following coding techniques in your application:

- Limit the amount of board-specific code (for example, board-specific control types and inquire types) in your application.
- Use the defaults when allocating your system and other MIL objects (for example, your digitizer, buffer, and display). By doing this, you create a more portable application, since these settings are not hard-coded in your application, and are determined when the user installs the MIL software. For more information on using the defaults, see the *Using the defaults* section earlier in this chapter.
- Use MIL's custom data types and portability functions, as outlined in the *MIL custom data types, extensions, and portability functions* section earlier in this chapter, instead of their operating system-specific equivalents.
- Avoid operating system-specific features (for example, using **MbufAlloc2d()** with **M_DIRECTX**, which is specific to a Windows-based operating system).

Chapter

3

Image processing

This chapter describes the steps to developing a typical application with the MIL Image Processing module.

Image processing overview

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.

MIL and image processing

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 convolutions. They include operations that can smooth images, enhance and detect 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.
- **Geometric transformation operations.** These operations are used to resolve distortion problems. They can properly orient an image, flip the image vertically or horizontally, and resize an image to obtain the right aspect ratio.

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.

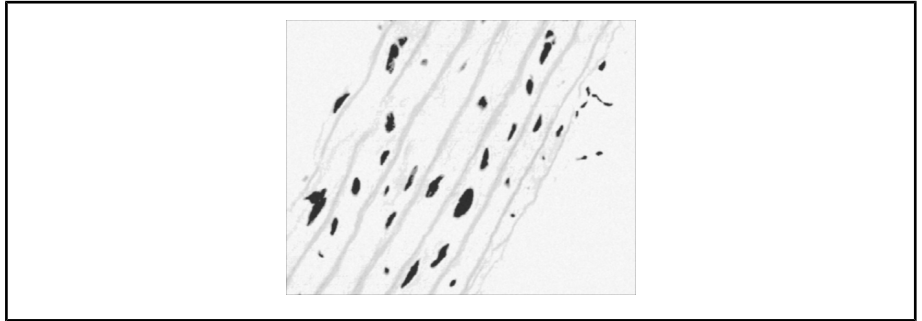
Image processing pertains to more than one field and no single application program can solve the problems associated with each one of these fields. Therefore, this section describes what we believe to be a typical problem, where the solution makes use of most of the supported operations. It also outlines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

The following steps provide a basic methodology for performing image processing with MIL:

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 (*MImProcessing.cpp*) shows you how to encode these steps, using an existing image of a tissue sample (*Cell.mim*).

```

/*****
/*
 * File name: MImProcessing.cpp
 *
 * Synopsis: This program show the usage of image processing. Under MIL lite,
 *           it binarizes the image of a tissue sample to show dark particles.
 *           Under MIL full, it also uses different image processing primitives
 *           to determine the number of cell nuclei that are larger than a
 *           certain size and show them in pseudo-color.
 */
#include <mil.h>

/* Target MIL image file specifications. */
#define IMAGE_FILE          M_IMAGE_PATH MIL_TEXT("Cell.mim")
#define IMAGE_SMALL_PARTICLE_RADIUS  1

int MosMain(void)
{
    MIL_ID    MilApplication,      /* Application identifier. */
             MilSystem,           /* System identifier. */
             MilDisplay,          /* Display identifier. */
             MilImage,            /* Image buffer identifier. */
             MilExtremeResult = 0; /* Extreme result buffer identifier. */
    MIL_INT   MaxLabelNumber = 0; /* Highest label value. */
    MIL_INT   LicenseModules = 0; /* List of available MIL modules */

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

    /* Restore source image and display it. */
    MbufRestore(IMAGE_FILE, MilSystem, &MilImage);
    MdispSelect(MilDisplay, MilImage);

    /* Pause to show the original image. */
    MosPrintf(MIL_TEXT("\nIMAGE PROCESSING:\n"));
    MosPrintf(MIL_TEXT("-----\n\n"));
    MosPrintf(MIL_TEXT("This program extracts the dark particles in the image.\n"));
    MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
    MosGetch();

    /* Binarize the image with an automatically calculated threshold so that
       particles are represented in white and the background removed.
    */
    MimBinarize(MilImage, MilImage, M_LESS_OR_EQUAL, M_DEFAULT, M_NULL);
}

```

```

/* Print a message for the extracted particles. */
MosPrintf(MIL_TEXT("These particles were extracted from the original image.\n"));

#ifdef !MIL_LITE
/* If MIL IM module is available, count and label the larger particles. */
MsysInquire(MilSystem, M_LICENSE_MODULES, &LicenseModules);
if (LicenseModules & M_LICENSE_IM)
{
    /* Pause to show the extracted particles. */
    MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
    MosGetch();

    /* Close small holes. */
    MimClose(MilImage, MilImage, IMAGE_SMALL_PARTICLE_RADIUS, M_BINARY);

    /* Remove small particles. */
    MimOpen(MilImage, MilImage, IMAGE_SMALL_PARTICLE_RADIUS, M_BINARY);

    /* Label the image. */
    MimLabel(MilImage, MilImage, M_DEFAULT);

    /*The largest label value corresponds to the number of particles in the image.*/
    MimAllocResult(MilSystem, 1L, M_EXTREME_LIST, &MilExtremeResult);
    MimFindExtreme(MilImage, MilExtremeResult, M_MAX_VALUE);
    MimGetResult(MilExtremeResult, M_VALUE, &MaxLabelNumber);
    MimFree(MilExtremeResult);

    /* Multiply the labeling result to augment the gray level of the particles. */
    MimArith(MilImage, (MIL_INT)(256L/(MIL_DOUBLE)MaxLabelNumber), MilImage,
                                                    M_MULT_CONST);

    /* Display the resulting particles in pseudo-color. */
    MdispLut(MilDisplay, M_PSEUDO);

    /* Print results. */
    MosPrintf(MIL_TEXT("There were %ld large particles in the original image.\n"),
                                                    MaxLabelNumber);
}
#endif

/* Pause to show the result. */
MosPrintf(MIL_TEXT("Press <Enter> to end.\n\n"));
MosGetch();

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

return 0;
}

```

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 multiple buffering, a technique whereby you can grab data into one buffer while others buffers are being processed. For more information, see the *Multiple buffering* subsection in the *Grabbing and processing* section in *Chapter 22: Grabbing with your digitizer* and for an example on how to implement multiple buffering, see *MdigProcess.cpp* file in MIL's example directory.

Denoising

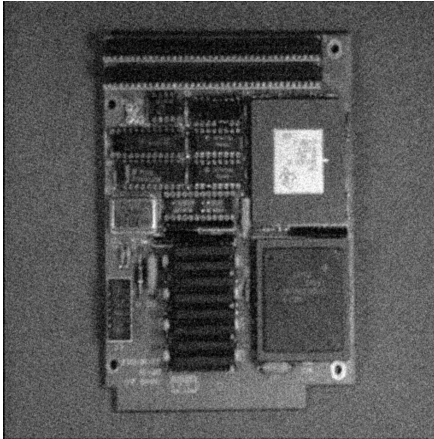
Spatial filtering and area open and close operations are effective methods to reduce noise. Spatial filtering operations determine each pixel's value based on its neighborhood values. They allow images to be separated into high-frequency and low-frequency components. There are two main types of spatial filters that can remove noise: low-pass linear filters and rank filters. Area open and close operations are efficient in removing salt-and-pepper noise in an image.

Low-pass spatial linear filters

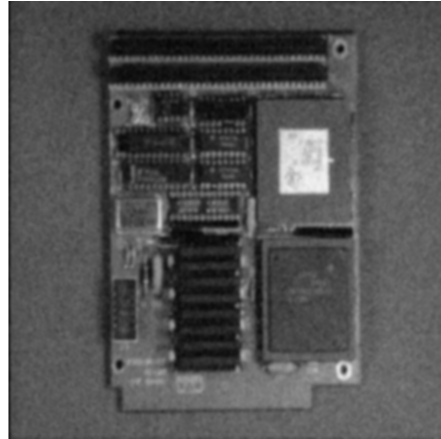
Low-pass spatial linear filters are effective in reducing Gaussian random noise (and high-frequency systematic noise), provided that the noise frequency is not too close to the spatial frequency of significant image data. These filters replace each pixel with a weighted sum of each pixel's neighborhood.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

You can apply low-pass spatial linear filters using the **MimConvolve()** function. MIL provides a predefined low-pass linear filter called **M_SMOOTH**, which you will find satisfies most applications. If you want to control the degree of smoothness (strength of denoising) applied by the neighborhood operation, use **MimConvolve()** with the **M_SHEN_FILTER** or **M_DERICHE_FILTER** macro, set its **FilterSmoothness** macro parameter to the required smoothness value, and set its **FilterOperation** macro parameter to **M_SMOOTH**. The **M_DERICHE_FILTER** macro applies a Canny-Deriche filter, and the **M_SHEN_FILTER** macro applies

a Shen-Castan filter. For the Shen-Castan filter, the neighborhoods' influence decreases much faster as the distance from the central pixel increases, and it tends to produce more enhanced or sharpened edges, compared to the Canny-Deriche filter.



Gaussian noise



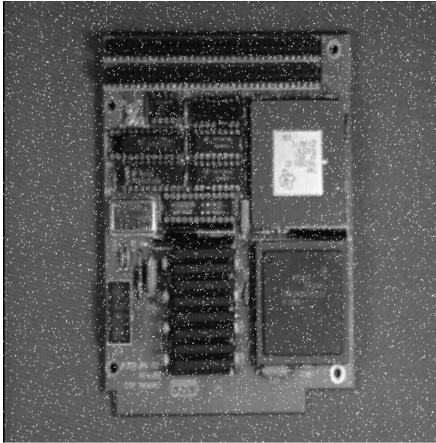
Low-pass spatial linear filter

- ❖ Earlier in the 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.

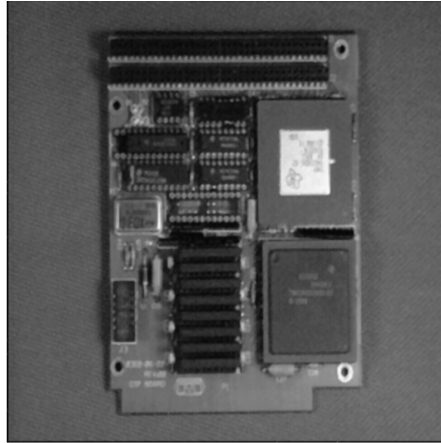
Rank filters

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, use **MimRank()** with **M_MEDIAN**. You will find that the median filter will most often suit your application needs.



Salt-and-pepper noise



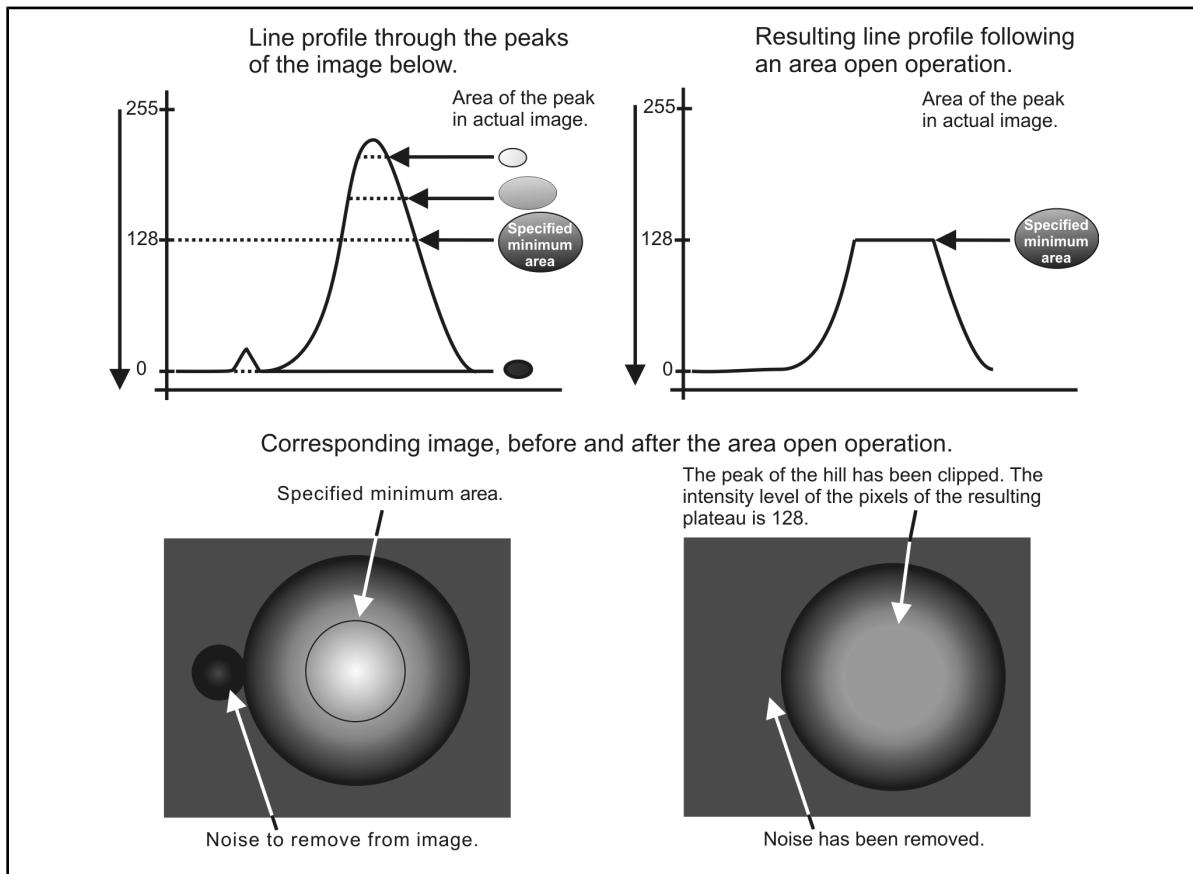
Rank filter

Area open and area close

To eliminate salt-and-pepper noise in your image, you can perform an area open or area close operation, using **MimMorphic()** with **M_AREA_CLOSE** or **M_AREA_OPEN**. Area open can be used to eliminate small regions of light noise pixels, while area close can do the same to small regions of dark noise pixels.

To understand what the area open and area close operations do, it is useful to think of an image as a topographic surface with hills and valleys. 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. The area open operation clips

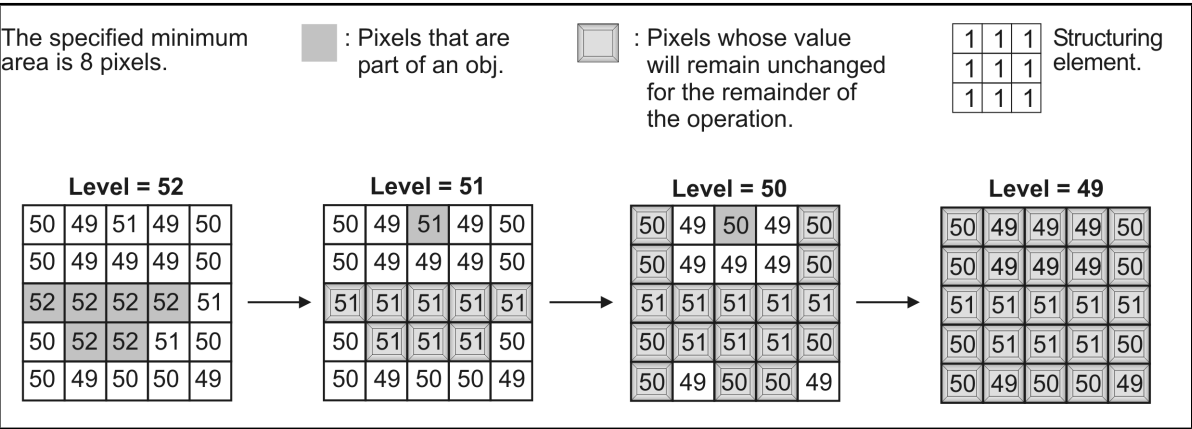
the peaks of hills until each has a plateau with an area equal to or greater than the specified minimum area. If an area of this size never occurs, the peak will be clipped until all the pixels in the hill have the same intensity as the background pixels. The area close operation is similar to the area open operation, except the clipping of intensity levels begins at the lowest level. In this case, you can imagine the bottom of the basins being filled until they have a flat surface area equal to or greater than the specified minimum area.



To implement the area open operation, MIL analyzes the image and clips off peaks one intensity level at a time, starting at the highest pixel intensity level (255 for an 8-bit image). At each intensity level, the following steps are performed:

1. Pixels with an intensity level greater than the current intensity level and that have not been previously locked are set to the current intensity level.
2. The algorithm then considers all pixels with a value equal to or less than the current value to be foreground pixels, and connected foreground pixels as part of the same object.
3. The area of each object is compared to the specified minimum area. If it is greater or equal to the specified minimum area, the pixels in the object are locked at their current value for the remainder of the algorithm. If the current intensity level is 0, the operation is complete. If not, the algorithm proceeds to the next lower intensity level, and returns to step 1.

In the event that the area of an object at every pixel intensity level never equals or exceeds the specified minimum area, its pixels will eventually be clipped to the background pixel intensity level. Also, it is possible that distinct objects at one pixel intensity level merge together at a lower pixel intensity level. This situation is illustrated in the following images, which depict a few steps in the area open operation.



As was mentioned earlier, an object is composed of connected foreground pixels. Determining which pixels are connected depends on the selected structuring element, **M_3X3_RECT** or **M_3X3_CROSS**. These are the only two structuring elements that can be used with the area open and area close operations.

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. This section describes basic erosion and dilation performed using **MimErode()** and **MimDilate()**. Both these functions use predefined kernels to perform operations. For a description of advanced erosion and dilation, performed by **MimMorphic()**, see the *Custom morphological operations* section in *Chapter 4: Advanced image processing*.

They are also the basic operations used to perform the opening and closing operations discussed in the following section.

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

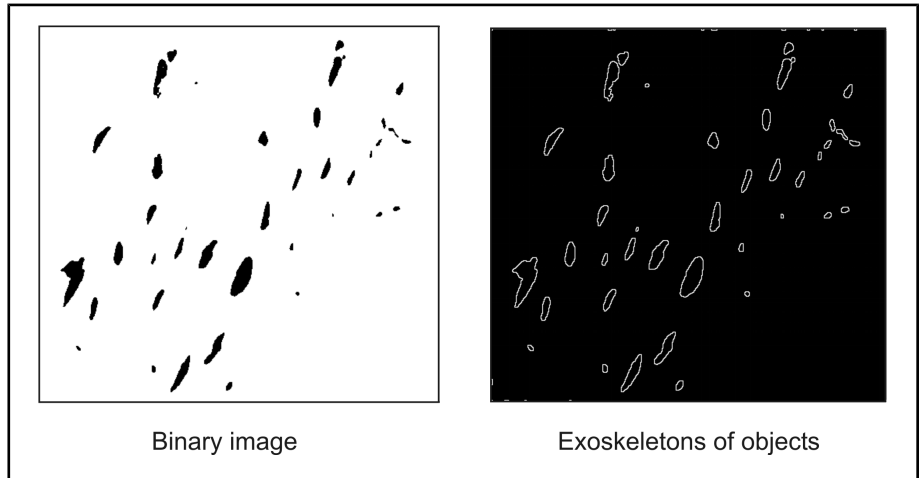
You can perform a basic dilation operation on 3 by 3 neighborhoods, using **MimDilate()**.

- 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()** function is similar to the **MimErode()** function 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: MImPerimeter.cpp
*
* Synopsis: This program finds the exoskeletons of the perimeters of
*           dark objects in an image.
*/
#include <mil.h>
#include <conio.h>

/* Source MIL image file specifications. */
#define IMAGE_FILE           M_IMAGE_PATH MIL_TEXT("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 MosMain(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, M_NULL);

/* Restore source image into an image buffer and displays it. */
MbufRestore(IMAGE_FILE, MilSystem, &MilImage);
MdispSelect(MilDisplay, 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);

/* Pause to show the original image. */
MosPrintf(MIL_TEXT("\nThis program finds the exoskeletons"));
MosPrintf(MIL_TEXT("of the particles in the displayed image.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* 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. */
MosPrintf(MIL_TEXT("Exoskeletons of the object's perimeters are being calculated.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to end.\n"));
MosGetch();

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

```

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()** function 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()**. For more information on erosion and dilation, see the *Custom morphological operations* section in *Chapter 4: Advanced image processing*.

Basic geometric transforms

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. For information on the advanced geometric functions, see *Chapter 4: Advanced image processing*.

Resizing an image	The MimResize() function resizes an image along the horizontal and/or vertical axis. This can help resolve aspect-ratio problems. If both the horizontal and vertical resizing factors are set to the same value, this function can reduce or magnify an image to an appropriate size.
Rotating an image	In some instances, the orientation of an image can also cause erroneous conclusions. When an object is rotated from its original position, you can realign it in memory by the required angle, using MimRotate() .
Translating an image	MimTranslate() displaces an image by a specified number of pixels in the x and/or y direction, with subpixel accuracy.
Flipping an image	MimFlip() flips an image horizontally (left to right) or vertically (top to bottom). Note that flipping horizontally allows you to get a mirror copy of the original image.
Interpolation	Geometric functions are performed according to a specified interpolation mode. For information on interpolation, see <i>Chapter 4: Advanced image processing</i> .

Image manipulation in general

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. The following sections 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()**).
- Calculate a number of general statistics for all pixels that satisfy a specified condition (**MimStat()**).
- 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 function 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()**.

The following example shows how to load an image, and calculate and draw its intensity histogram.

```

/*****
/*
 * File name: MimHistogram.cpp
 *
 * Synopsis: This program loads an image of a tissue sample, calculates its intensity
 *           histogram and draws it.
 */
#include <mil.h>

/* Target MIL image file specifications. */
#define IMAGE_FILE M_IMAGE_PATH MIL_TEXT("Cell.mim")

/* Number of possible pixel intensities. */
#define HIST_NUM_INTENSITIES 256
#define HIST_SCALE_FACTOR 8
#define HIST_X_POSITION 250
#define HIST_Y_POSITION 450

/* Main function. */
int MosMain(void)
{
    MIL_ID      MilApplication,          /* Application identifier */
               MilSystem,               /* System identifier. */
               MilDisplay,              /* Display identifier. */
               MilImage,                /* Image buffer identifier. */
               MilOverlayImage,         /* Overlay buffer identifier. */
               HistResult;              /* Histogram buffer identifier. */

```

```

MIL_INT    HistValues[HIST_NUM_INTENSITIES]; /* Histogram values.          */
MIL_INT    XStart[HIST_NUM_INTENSITIES], YStart[HIST_NUM_INTENSITIES],
           XEnd[HIST_NUM_INTENSITIES],   YEnd[HIST_NUM_INTENSITIES];
MIL_DOUBLE AnnotationColor = M_COLOR_RED;
MIL_INT     i;

/* Allocate the default system and image buffer. */
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 and prepare for overlay annotations. */
MdispSelect(MilDisplay, MilImage);
MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

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

/* Calculate the histogram. */
MimHistogram(MilImage, HistResult);

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

/* Draw the histogram in the overlay. */
MgraColor(M_DEFAULT, AnnotationColor);
for(i=0; i<HIST_NUM_INTENSITIES; i++)
{
    XStart[i] = i+HIST_X_POSITION+1,
    YStart[i] = HIST_Y_POSITION;
    XEnd[i]   = i+HIST_X_POSITION+1,
    YEnd[i]   = HIST_Y_POSITION-(HistValues[i]/HIST_SCALE_FACTOR);
}
MgraLines(M_DEFAULT, MilOverlayImage, HIST_NUM_INTENSITIES,
          XStart, YStart, XEnd, YEnd, M_DEFAULT);

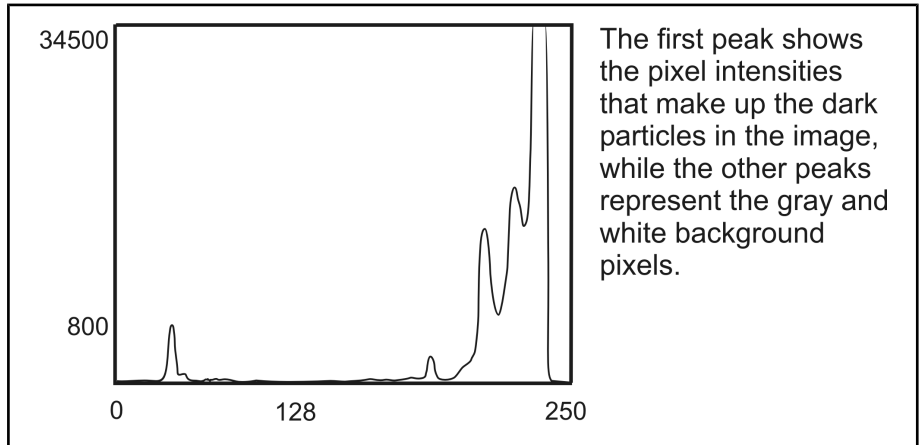
/* Print a message. */
MosPrintf(MIL_TEXT("\nHISTOGRAM:\n"));
MosPrintf(MIL_TEXT("-----\n\n"));
MosPrintf(MIL_TEXT("The histogram of the image was calculated and drawn.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to end.\n\n"));
MosGetch();

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

return 0;
}

```

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



Calculating general statistics

You can calculate various statistics for all pixels that satisfy a specified condition in an image, using **MimStat()**. Setting the condition limits the number of pixels for which the statistics will be calculated. However, the condition can also be set to use all of the pixels in the image.

The types of statistics that can be calculated include the following:

- Maximum pixel value.
- Maximum absolute pixel value.
- Mean pixel value.
- Minimum pixel value.
- Minimum absolute pixel value.
- Number of pixels (that satisfy the condition).
- Standard deviation value.
- Sum of pixel values.

- Sum of absolute pixel values.
- Sum of squared pixel values.

Note that **MimStat()** stores results in a result buffer that should have been previously allocated, using **MimAllocResult()** with **M_STAT_LIST**.

Finding the image extremes

Besides using **MimStat()** to find 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 frame grabber, ensuring full-range digitization. Another use for finding the maximum image pixel value 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.

MimFindExtreme() stores results in a result buffer that should have been previously allocated, using **MimAllocResult()** with **M_EXTREME_LIST**. You can get the resulting values, using **MimGetResult()**, and free the result buffer, using **MimFree()**.

Locating events

Once you have established certain values of interest in an image, you can find the location of pixels that satisfy conditions based on these values, using **MimLocateEvent()**. For example, you can use **MimLocateEvent()** to find the location of all pixels in an image equal to the image's maximum pixel value.

Note that **MimLocateEvent()** stores results in a result buffer that should have been previously allocated, using **MimAllocResult()** with **M_EVENT_LIST**.

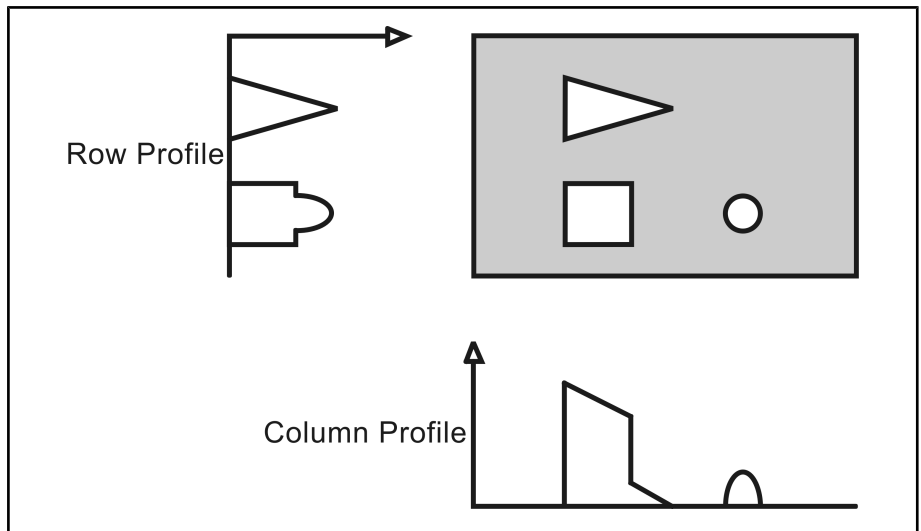
Counting image differences

You can find the number of differences between two images, using **MimCountDifference()**. **MimCountDifference()** stores results in a result buffer that should have been previously allocated, using **MimAllocResult()** with **M_COUNT_LIST**. You can get the resulting values, using **MimGetResult()**, and free the result buffer, using **MimFree()**.

Projecting an image to one dimension

The **MimProject()** function 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° projection of the image is known as the row profile, and the 0° projection is known as the column profile.

The **MimProject()** function 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 **M_PROJ_LIST**. 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()**.

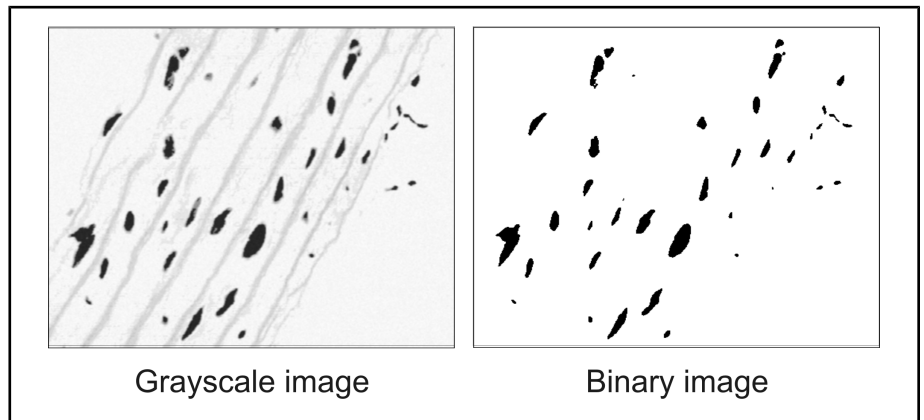
Thresholding your images

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()** function.
- Clipping, using the **MimClip()** function.

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); however, in the case of a floating-point buffer, these values 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.

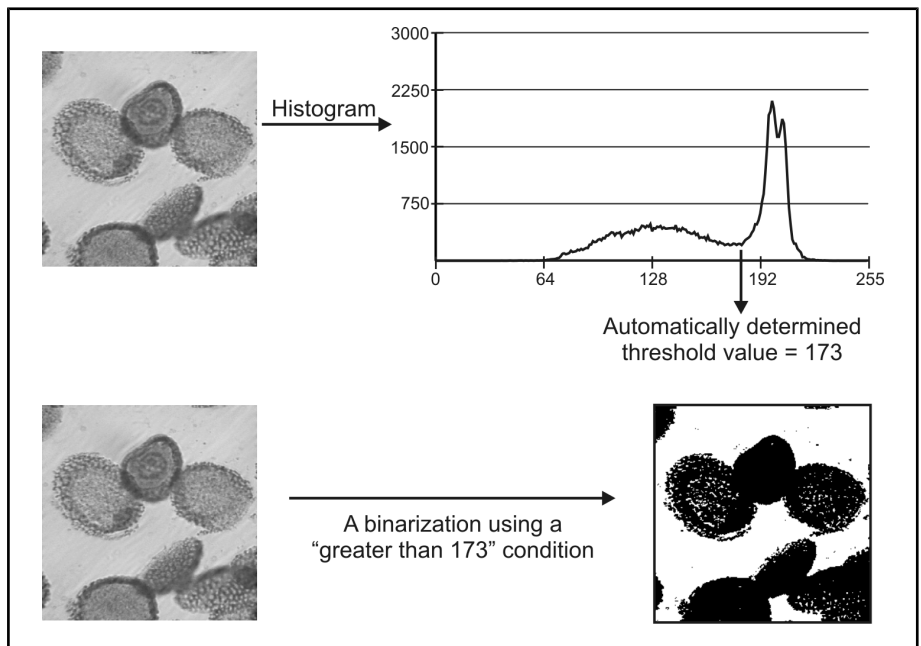


A binarizing operation is performed by comparing each pixel value in the image against one or two specified threshold values (for example, whether each pixel value is above one of the threshold values, or within the range of the two threshold values). Pixels that meet the specified condition are typically set to the maximum value in the image while other pixels are set to 0. Except in the case of a floating-point 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 the previous image, an incorrect threshold value might inappropriately change image pixels into background pixels, resulting in fewer or more particles than the number that 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 principal peaks, though exceptions exist for pure black and full saturation (0, 255).



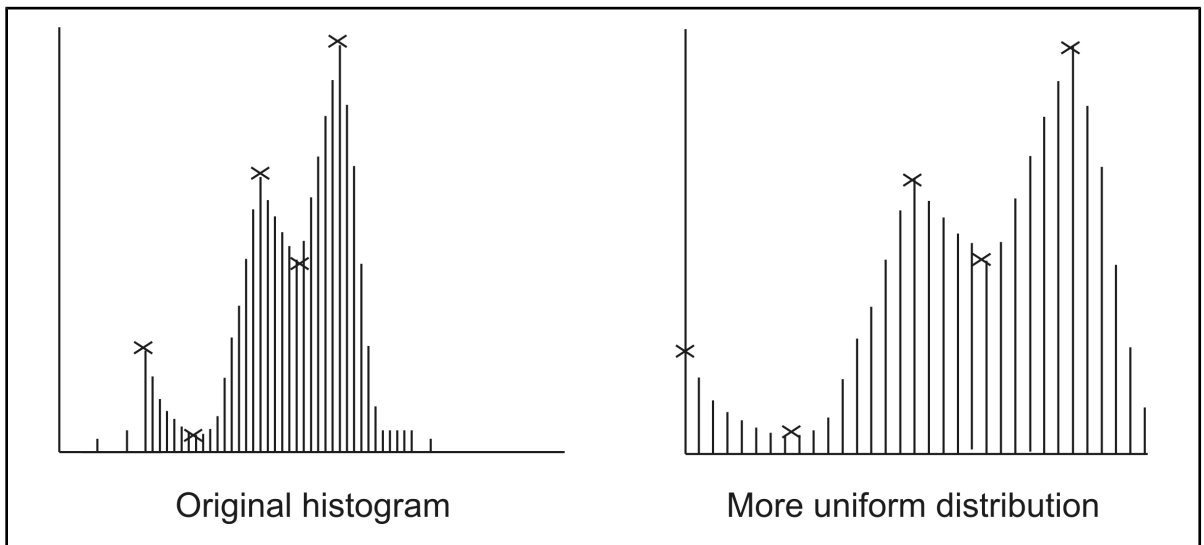
Clipping

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.

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 range, 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()** function 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; this LUT can then be used to enhance the source image, either permanently (using **MimLutMap()**) or upon display (using **MdispLut()**). The transformation LUT can also be applied directly to images as they are being grabbed; to do so, first associate the LUT buffer with the digitizer, using **MdigLut()** and then grab the image.

Enhancing and detecting edges

Many applications perform various edge operations on the image objects to increase the quality of the image or to limit some other operation on the image.

In general, edges can be established from intensity transitions between two or more adjacent pixels in an image. 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 three main categories of edge operations:

- Operations that enhance edges to sharpen the image.
- Operations that detect edges in the image.
- Operations that extract edges in the image.

These edge operations are performed using convolutions (or neighborhood operations that replace each pixel with a weighted sum of each pixel's neighborhood). The weights applied to the neighborhood determine the type of operation that is performed. For example, certain weights produce a horizontal edge detection, while others produce a vertical one.

For the first two types of edge operations, the weights are specified using either a predefined or a custom Finite or Infinite Impulse Response (FIR or IIR) filter. For more information on filters, see the *Custom spatial filters* section in *Chapter 4: Advanced image processing*. The **MimConvolve()** function offers predefined FIR and IIR filters for most common edge enhancement and detection operations. Each offers some advantage over the others. You can try these filters to see which best suits your application needs. If you want to control the degree of smoothness (strength of denoising) applied by the neighborhood operations, use a predefined IIR filter (either the **M_SHEN_FILTER** or **M_DERICHE_FILTER** macro), and set its FilterSmoothness macro parameter to the required smoothness value. The **M_DERICHE_FILTER** macro applies a Canny-Deriche filter, and the **M_SHEN_FILTER** macro applies a Shen-Castan filter. For the Shen-Castan filter, the neighborhood's influence decreases much faster as the distance from the central pixel increases, and it tends to produce more enhanced or sharpened edges, compared to the Canny-Deriche filter.

For information on edge extraction operations, see the *MIL Edge Finder module* section in *Chapter 9: Edge Finder*.

Edge enhancement

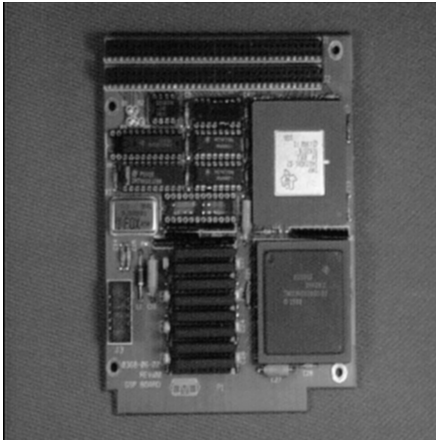
Edge enhancement operations amplify edges, accentuating details in the image. Note that these operations might not produce good results for further processing because when you enhance edges, you also enhance noise pixels.

Sharpening edge enhancement

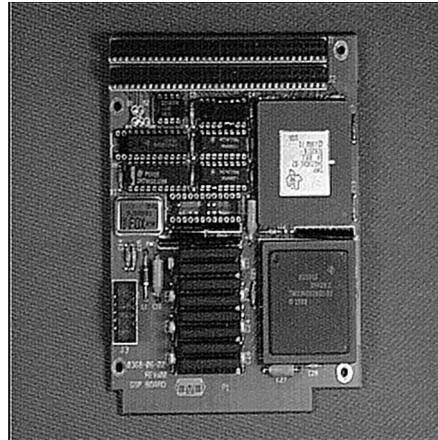
Sharpening edges can be useful to enhance edges in the image while preserving other parts of the image. You can obtain approximately the same result by performing a Laplacian-based edge detection operation on the image and adding the found edges to the original image.

To sharpen edges in an image, you can use **MimConvolve()** with the **M_SHARPEN** or **M_SHARPEN2** predefined FIR filter. To apply a predefined IIR filter, you can also use **MimConvolve()** with the **M_SHEN_FILTER** or **M_DERICHE_FILTER** macro, and set its **M_DERICHE_FILTER FilterOperation** macro parameter to **M_DERICHE_FILTER M_SHARPEN**.

When using predefined FIR filters, the **M_SHARPEN** filter tends to produce more enhanced or sharpened edges than the **M_SHARPEN2** filter.



Original image



Edge enhancement

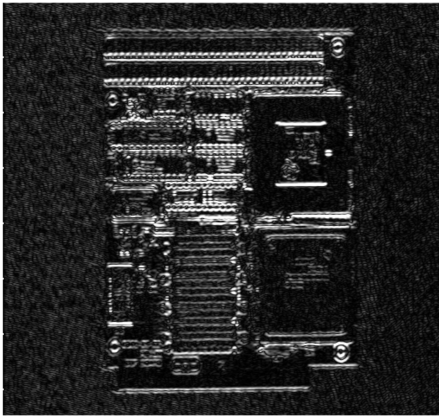
Edge detection

Edge detection operations reveal intensity transitions in the image. The smoother the image, the more gradual the change in intensity, and the weaker the detection will be.

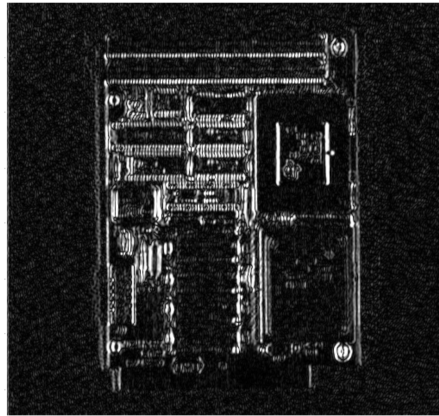
Horizontal and vertical edge detection

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

To detect the horizontal or vertical edges from an image, you can use **MimConvolve()** with the **M_HORIZ_EDGE** or **M_VERT_EDGE** predefined FIR filter, respectively. To apply a predefined IIR filter, you can also use **MimConvolve()** with the **M_SHEN_FILTER** or **M_DERICHE_FILTER** macro, and set its **FilterOperation** macro parameter to **M_HORIZ_EDGE** or **M_VERT_EDGE**.



Horizontal edge detection



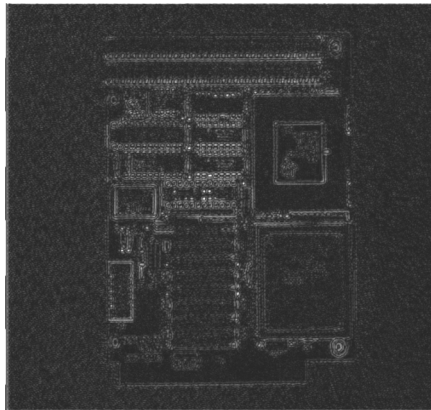
Vertical edge detection

Laplacian-based edge detection

The Laplacian-based operations place emphasis on the maximum values, or peaks, within the image. The edge representation of the resulting image generally looks very similar to the actual image.

To detect the Laplacian-based edges from an image, you can use **MimConvolve()** with the **M_LAPLACIAN_EDGE** or **M_LAPLACIAN_EDGE2** predefined FIR filter. To apply a predefined IIR filter, you can also use **MimConvolve()** with the **M_SHEN_FILTER** or **M_DERICHE_FILTER** macro, and set its **FilterOperation** macro parameter to **M_LAPLACIAN_EDGE**.

When using predefined FIR filters, the **M_LAPLACIAN_EDGE2** filter tends to produce more enhanced or sharpened edges than the **M_LAPLACIAN_EDGE** filter.

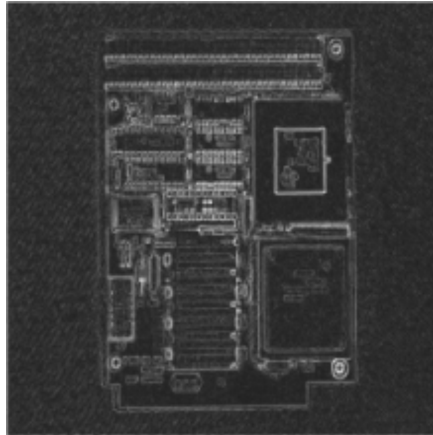


Laplacian-based edge detection

Gradient-based edge detection

When performing a gradient-based 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.

To detect the gradient-based edges from an image, you can use **MimConvolve()** with the **M_EDGE_DETECT** or **M_EDGE_DETECT2** predefined FIR filter. To apply a predefined IIR filter, you can also use **MimConvolve()** with the **M_SHEN_FILTER** or **M_DERICHE_FILTER** macro, and set its **M_DERICHE_FILTER FilterOperation** macro parameter to **M_DERICHE_FILTER M_EDGE_DETECT** or **M_DERICHE_FILTER M_EDGE_DETECT_SQR**.



Gradient-based edge detection

For an advanced gradient-based edge detection operation, you can use **MimEdgeDetect()**. This function allows you to specify the threshold value and produces a gradient intensity image and/or a gradient angle image.

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 negate, take the absolute value, NOT, 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: MDigSubtract.cpp
*
* Synopsis: This program grabs an image which is expected to be the constant dark
*           background, and then subtracts this background from subsequent grabbed
*           images.
*/
#include <mil.h>
#include <conio.h>
#include <stdlib.h>

void MosMain(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.*/
MIL_INT CamSizeX,      /* Camera width variable.      */
        CamSizeY;      /* Camera height variable.     */

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

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

/* Allocate a first image buffer to store and display the result image. */
MbufAlloc2d(MilSystem, CamSizeX, CamSizeY, 8+M_UNSIGNED,
            M_IMAGE+M_PROC+M_GRAB+M_DISP, &MilImage);

/* Allocate a second image buffer to store the background image. */
MbufAlloc2d(M_DEFAULT, CamSizeX, CamSizeY, 8+M_UNSIGNED,
            M_IMAGE+M_PROC, &BackgroundImage);

/* Allocate a third image buffer to grab the changing image. */
MbufAlloc2d(MilSystem, CamSizeX, CamSizeY, 8+M_UNSIGNED,
            M_IMAGE+M_PROC+M_GRAB, &GrabImage);

/* Select the image on the display. */
MbufClear(MilImage, 0);
MdispSelect(MilDisplay, MilImage);

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

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

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

/* When a key is pressed, halt. */
MosPrintf(MIL_TEXT("Continuous subtraction in progress...\n\n"));
MosPrintf(MIL_TEXT("Keeping your camera in the same position, create \
                    motion with a bright\n"));
MosPrintf(MIL_TEXT("object in front of the background.\n"));

```

```

MosPrintf(MIL_TEXT("Press <Enter> to end.\n"));

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

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

```

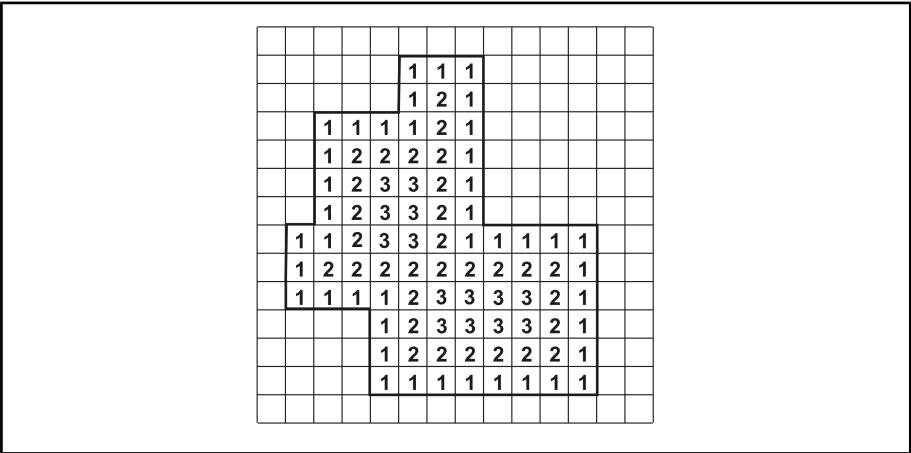
Mapping an image

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()**.

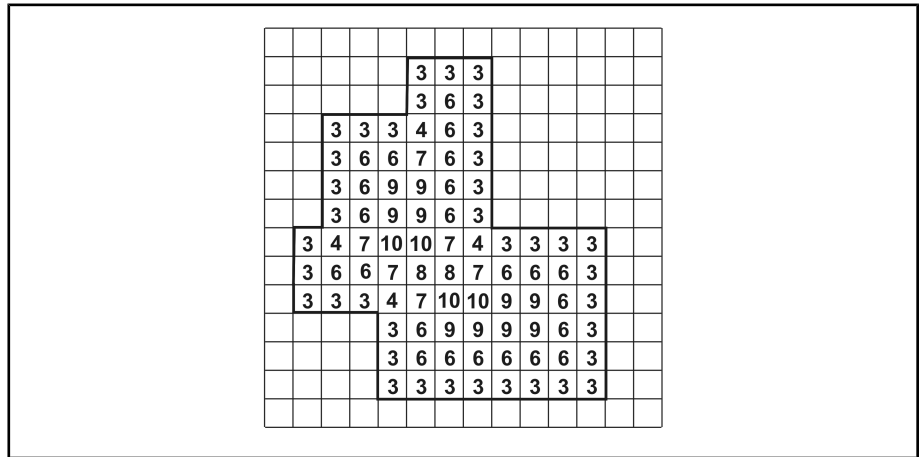
Chessboard transform

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



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.

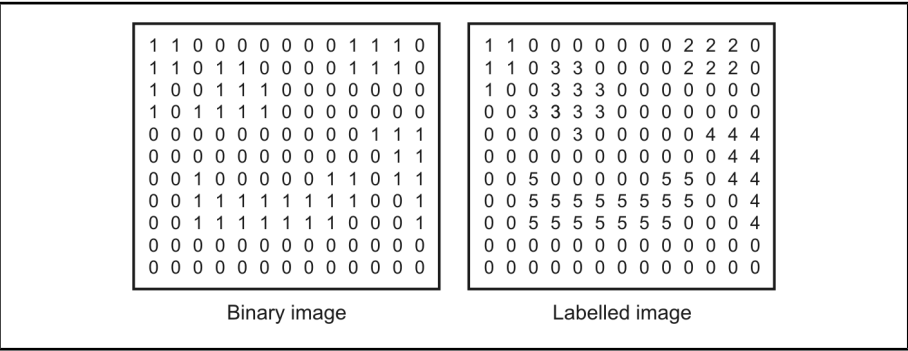


Labeling

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()** function 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:

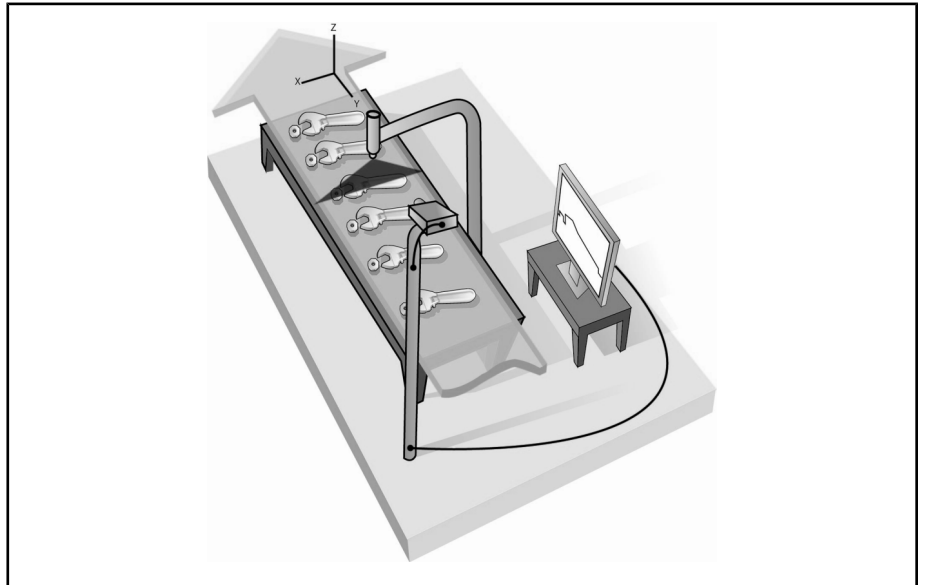
- **M_4_CONNECTED:** If two pixels touch on the vertical or horizontal, they are considered part of the same blob.
- **M_8_CONNECTED:** If two pixels touch on the vertical, horizontal, or diagonal, they are considered part of the same blob.

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

Peak intensity detection and range images

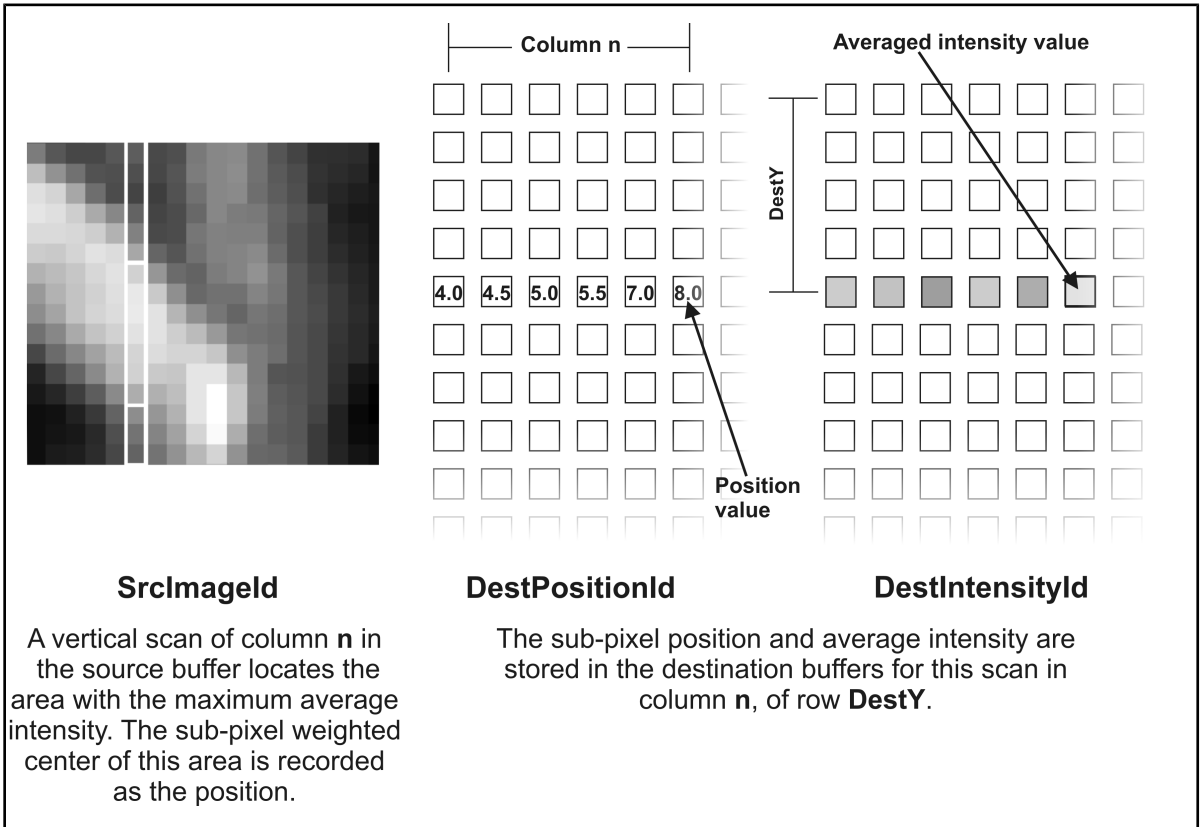
When you need to create an application for acquiring dimensional information about an object, you can use **MimLocatePeak1d()** with a frame grabber and a laser line device to create a structured light application.

In MIL, a structured light application can be built by projecting a laser line, at a specified angle, onto an object. The image of the projected laser line will have variations in intensity that correspond to the surface dimensions of the object. For example, surfaces of the object that are closer to the laser will have higher intensities than surfaces that are further away. **MimLocatePeak1d()** can locate the peak intensity in every row (or column) in a grabbed image, allowing you to interpret the surface dimensions of the object on which the laser is projected. By grabbing a sequence of images as the object moves underneath the laser line, and then processing the images with **MimLocatePeak1d()**, you can create a complete 3D map of an object's surface.



MimLocatePeak1d() performs very fast and efficient peak intensity detection on each row (or column) of pixels in a source image. The function finds the neighborhood in each row (or column) with the greatest average intensity and records the average as the peak intensity value for that row (or column). The sub-pixel weighted center of the neighborhood is recorded as the position of the peak intensity value.

The function uses two destination image buffers to store the results: one for the position and one for the intensity. Although the function writes to two destination image buffers, it only writes to one row of each. For every row (or column) in the source image, the function determines the X-coordinate (or Y-coordinate) and value of the peak intensity and records them in the specified row of the destination image buffers at an offset corresponding to the row (or column) in the source image buffer. For example, for column 6 of the source image, the function determines the Y-coordinate and value of the peak intensity and records them in pixel 6 in the specified row of their corresponding destination buffers. Consecutive calls to **MimLocatePeak1d()** can fill consecutive rows of the destination image buffers, producing a complete pattern of the changes in position and value of the peak intensities in a series of source images.



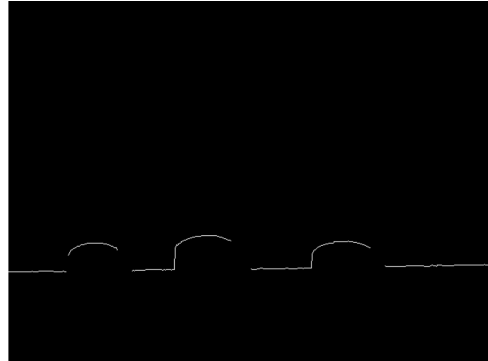
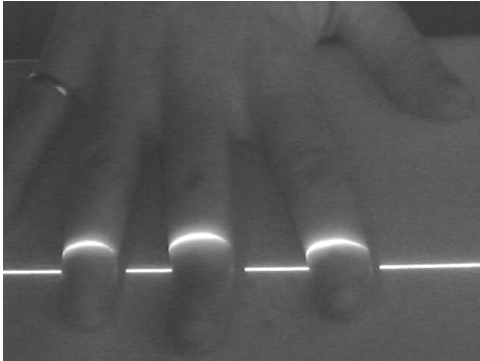
You can control how accurately the position of the average intensity is calculated by setting the width of the neighborhood; larger neighborhoods will result in more accurate weighted centers than smaller ones. However, it is important to consider that large neighborhoods can have a negative effect on performance; it takes longer to find the average intensity of larger neighborhoods than it does for smaller ones.

You can record the position with sub-pixel accuracy by specifying the number of fractional bits for a fixed-point approximation of the position. However, every fractional bit that you use exponentially decreases the maximum position value that you can record.

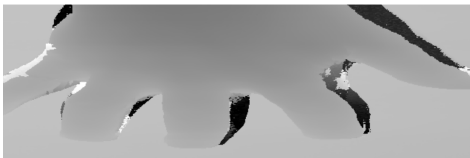
Creating depth maps (range images)

You can create depth maps using the `MimLocatePeak1d()` function. A **depth map** is an image where pixels represent distances of the surface of the object from the plane it is on. Changes in intensity levels represent changes in the surface height. Depth maps are often created using a laser line tool that projects an intensely bright line on a passing object. A camera captures images of the ribbon of light as it undulates with the varying surface heights of the object; a single image capture from the camera is used to calculate the position and average peak intensity values. You can create a depth map that depicts a complete 3D map of the exposed topography using a sequence of laser line-based images.

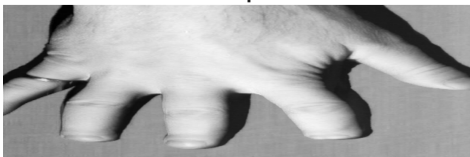
The example below creates a depth map using a structured light application.



A single image (left) of a hand passing through a projected laser line is used as the source image buffer for `MimLocatePeak1d()`. The peak in every column is detected. The image on the right is the output of `MimDraw()` with the destination buffers from `MimLocatePeak1d()` with `M_1D_COLUMNS`.



The destination position buffer



The destination intensity buffer

Successive calls to `MimLocatePeak1d()` with a sequence of laser line image captures creates a range image in the destination position buffer and an intensity map in the intensity buffer. The changes in intensity represent changes in the surface area of the hand.

The height of the destination buffers is proportional to the number of images in the sequence. In this example, the hand moved approximately 2 pixels per image capture so the height of the destination buffers is half of the source images.

A depth map created with successive calls to **MimLocatePeak1d()** is a 3D representation of the surface of the object. However, the gray values of the pixels and the object's geometry are not corrected, that is, you cannot determine the absolute depth of a pixel from its gray value nor the undistorted geometrical shape of the object. If your application requires accurate depth or shape information, see the *Chapter 16: 3D Reconstruction* module for more information on how to create partially or fully corrected depth maps.

Chapter

4

Advanced image processing

This chapter describes different advanced image processing techniques.

Advanced image processing in general

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 such as smoothing, denoising, edge enhancement, and edge detection.

Spatial filtering operations compute results based on an underlying neighborhood process: the weighted sum of a pixel value and its neighbors' values. There are two types of spatial filters: Finite Impulse Response (FIR) filters and Infinite Impulse Response (IIR) filters. FIR filters operate on a finite neighborhood, while IIR filters take into account all values in an image. In MIL, you can specify either a predefined or a custom FIR or IIR spatial filter.

Finite Impulse Response (FIR) filters

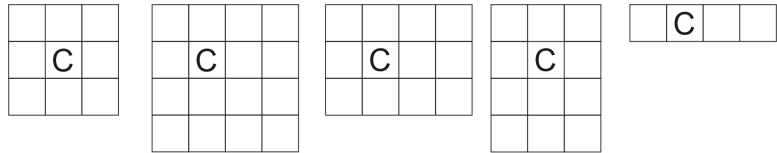
For FIR filters, the weights are known as the kernel values. These kernel values determine the operation type of the spatial filter. For example, applying the following FIR filter results in a sharpening of the image:

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

Whereas, applying the following FIR filter 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):

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

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



Examples of neighborhoods and their center pixel.

Calculate the X-coordinate of the top-left pixel of the central elements, as follows:

- If the width (SizeX) of the kernel is an odd number, the X-coordinate is $(\text{SizeX}-1)/2$.
- If the width (SizeX) of the kernel is an even number, the X-coordinate is $(\text{SizeX}/2)-1$.

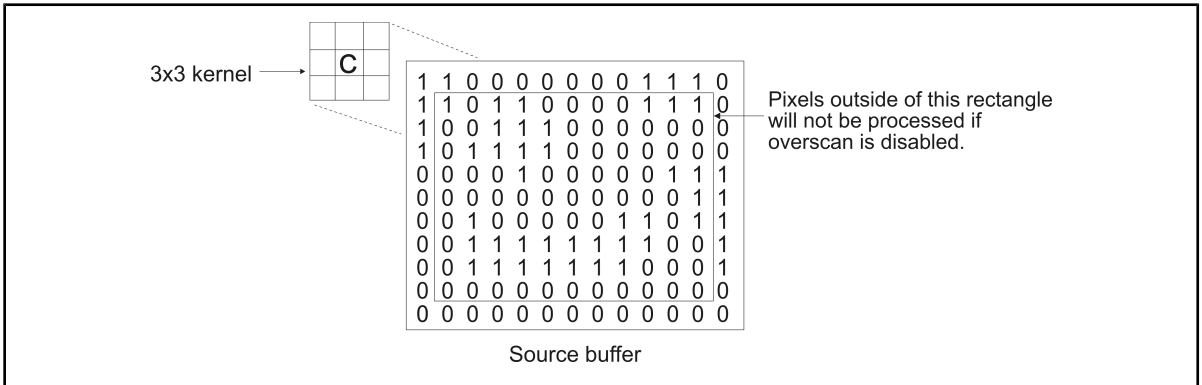
To calculate the Y-coordinate of the top-left pixel of the central elements, the same rules apply.

Regardless of the location of the center pixel, there will be some border pixels that have an incomplete neighborhood. To deal with this issue, the image buffer is overscanned. There are several types of overscan. A transparent overscan uses the parent buffer to provide the overscan pixels needed for the border calculation. 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 border pixels. A replacement overscan allows you to specify a specific value for the overscan pixel values during processing. For more information on overscan, see the *Buffer overscan* section in *Chapter 18: Specifying and managing your data buffers*.

If the predefined FIR filters provided by **MimConvolve()** do not meet your requirements, you can create your own custom FIR filter. To define your own custom FIR filter:

1. Allocate a kernel buffer, using **MbufAlloc1d()** or **MbufAlloc2d()** with **M_KERNEL**. The kernel size is specified when allocating the kernel buffer. Note that the kernel size can be constrained by the available resources.
2. Load the required kernel values into this kernel buffer, using **MbufPut()** or **MbufPut2d()**.
3. If required, modify the setting of the operation control types associated with your custom filter, using **MbufControlNeighborhood()**. The operation control types determine how the convolution operation will be handled. You can control:
 - Whether or not the absolute value of the result is taken (**M_ABSOLUTE_VALUE**).
 - The division (normalization) factor to apply to the result (**M_NORMALIZATION_FACTOR**).
 - Whether or not to saturate the result (**M_SATURATION**).

- The position of the center pixel (**M_OFFSET_CENTER_X** and **M_OFFSET_CENTER_Y**).
- How the operation handles the borders (overscan) of the source buffer (**M_OVERSCAN** and **M_OVERSCAN_REPLACE_VALUE**). If overscan is disabled, the border pixels of the source image are not processed if additional processing time is needed.



To apply your own custom FIR filter, call the **MimConvolve()** function, specifying the identifier of the required kernel buffer (**KernelBufId**).

To increase the speed of the convolution operation when using your custom FIR filter, **MimConvolve()** will automatically separate large kernels, if separable, into two 1-dimensional kernels ($a_{ij} = h_i v_j$). Performing two separate convolutions, once with $H_{n \times 1}$ and once with $V_{1 \times m}$ can be faster and is equivalent to performing one convolution operation using the original $A_{m \times n}$ kernel. MIL will internally separate large kernels when it detects that the separation results in better performance. The following displays an $A_{m \times n}$ kernel separated into two 1-dimensional kernels (H_m and V_n).

$$\begin{pmatrix} a_{00} & \dots & a_{0j} & \dots & a_{0n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i0} & \dots & a_{ij} & \dots & a_{in} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m0} & \dots & a_{mj} & \dots & a_{mn} \end{pmatrix} = \begin{pmatrix} h_0 \\ \vdots \\ h_l \\ \vdots \\ h_m \end{pmatrix} \begin{pmatrix} v_0 & \dots & v_j & \dots & v_n \end{pmatrix}$$

Infinite Impulse Response (IIR) filters

When using IIR filters, the weights are automatically determined by the type of filter, the mode of the filter, the type of operation to perform, and the degree of smoothness (strength of denoising) applied by the filter.

The type of filter determines the distribution of the neighborhoods' influence. MIL supports two types of IIR spatial filters: Canny-Deriche filter and Shen-Castan filter. For the Canny-Deriche filter, the neighborhoods' influence decreases much slower as the distance from the central pixel increases, compared to the Shen-Castan filter. For more information, see the *Customizing the edge extraction settings* section in *Chapter 9: Edge Finder*.

There are two modes in which to perform the filter: kernel mode and recursive mode. When performing an IIR filter in recursive mode, the kernel size would be theoretically infinite if this mode actually used a kernel. When performing a custom IIR filter in kernel mode, the filtering is done using a kernel approximation (FIR) of the filter. Kernel mode is not available for a predefined IIR filter. Kernel mode has been provided to take advantage of your system if it is optimized for kernel mode; kernel mode is typically not as efficient as recursive mode unless your system is optimized for kernel mode.

If the predefined IIR filters provided by **MimConvolve()** do not meet your requirements, you can create your own IIR filter. To define your own IIR filter:

1. Allocate a kernel buffer, using **MbufAlloc1d()** or **MbufAlloc2d()** with **M_KERNEL**.
2. Use **MbufControlNeighborhood()** and specify appropriate **M_FILTER_TYPE**, **M_FILTER_MODE**, **M_FILTER_OPERATION**, and **M_FILTER_SMOOTHNESS** operation control type settings.
 - ❖ Note that the kernel size specified during allocation is ignored, unless you will be performing the filter in kernel mode. In which case, the specified kernel size can be constrained by the available resources. In addition, regardless of the filter mode, the kernel values specified in the kernel buffer are ignored; they are determined automatically.
3. If required, modify the setting of your custom filter, using **MbufControlNeighborhood()**. The operation control types determine how the convolution operation will be handled. You can control:
 - Whether or not the absolute value of the result is taken (**M_ABSOLUTE_VALUE**).
 - The division (normalization) factor to apply to the result (**M_NORMALIZATION_FACTOR**).
 - Whether or not to saturate the result (**M_SATURATION**).

A summary

The following is a summary of spatial filtering options.

FIR	Predefined		Custom		
	Filter Mode:	Kernel.	Filter Mode:	Kernel.	
	Filter:	Predefined selection that determines filter type, filter operation, neighborhood size, filter values, and center pixel (e.g. horizontal edge detect).	Filter type:	"User defined" or default.	
			Filter operation:	Determined by specified kernel values.	
			Neighborhood size:	Determined by specified kernel size.	
			Filter values:	Determined by specified kernel values.	
			Center pixel:	User-specified.	
	Overscan:	Optional.	Overscan:	Optional.	
			Absolute value:	Optional.	
			Normalization:	Optional.	
			Saturation:	Optional.	
IIR	Filter Mode:	Recursive.	Filter Mode:	Recursive.	Kernel (approximation).
	Filter type:	Predefined selection (e.g. Shen, Deriche).	Filter type:	Predefined selection (e.g. Shen, Deriche).	Predefined selection (e.g. Shen, Deriche).
	Filter operation:	Predefined selection.	Filter operation:	Predefined selection.	Predefined selection.
	Neighborhood size:	Infinite.	Neighborhood size:	Infinite.	User-specified kernel size.
	Filter values:	Predetermined based on filter type and operation.	Filter values:	Predetermined based on filter type and operation.	Predetermined based on filter type and operation.
	Filter smoothness:	User specified.	Filter smoothness:	User specified.	User specified.
	Overscan	No overscan.	Overscan:	No overscan.	Optional.
			Absolute value:	Optional.	Optional.
			Normalization:	Optional.	Optional.
			Saturation:	Optional.	Optional.
Note: filter mode = implementation; filter type = distribution of neighborhood's influence.					

An example

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

```

/*****
/*
 * File name: MImConvolve.cpp
 *
 * Synopsis: This program performs a 3x3 convolution using a custom kernel
 *           and calculates the convolution time.
 */
#include <mil.h>

/* Target MIL image specifications. */
#define IMAGE_FILE      M_IMAGE_PATH MIL_TEXT("BaboonMono.mim")
#define ZOOM_VALUE      2

/* Kernel data definition. */
#define KERNEL_WIDTH    3L
#define KERNEL_HEIGHT   3L
#define KERNEL_DEPTH    8L
unsigned char  KernelData[KERNEL_HEIGHT][KERNEL_WIDTH] =
    { {1, 2, 1},
      {2, 4, 2},
      {1, 2, 1}
    };

/* Timing loop iterations. */
#define NB_LOOP 100

int MosMain(void)
{
    MIL_ID MilApplication, /* Application identifier. */
          MilSystem,      /* System identifier. */
          MilDisplay,     /* Display identifier. */
          MilDisplayImage, /* Image buffer identifier. */
          MilImage,       /* Image buffer identifier. */
          MilKernel;      /* Custom kernel identifier. */

    long n;
    MIL_DOUBLE Time;

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

    /* Restore source image into an automatically allocated image buffers. */
    MbufRestore(IMAGE_FILE, MilSystem, &MilImage);
    MbufRestore(IMAGE_FILE, MilSystem, &MilDisplayImage);

    /* Zoom display to see the result of image processing better. */
    MdispZoom(MilDisplay, ZOOM_VALUE, ZOOM_VALUE);

```

```

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

/* Pause to show the original image. */
MosPrintf(MIL_TEXT("\nIMAGE PROCESSING:\n"));
MosPrintf(MIL_TEXT("-----\n\n"));
MosPrintf(MIL_TEXT("This program performs a convolution on the displayed image.\n"));
MosPrintf(MIL_TEXT("It uses a custom smoothing kernel.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Allocate a MIL kernel. */
MbufAlloc2d(MilSystem, KERNEL_WIDTH, KERNEL_HEIGHT,
            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);

/* Convolve the image using the kernel. */
MimConvolve(MilImage, MilDisplayImage, MilKernel);

/* Now time the convolution (MimConvolve()):
   Overscan calculation is disabled and a destination image that
   is not displayed is used to have the real convolution time. Also the
   function must be called once before the timing loop for more accurate
   time (dll load, ...).
 */
MbufControlNeighborhood(MilKernel, M_OVERSCAN, M_DISABLE);
MimConvolve(MilDisplayImage, MilImage, MilKernel);
MappTimer(M_TIMER_RESET+M_SYNCHRONOUS, M_NULL);
for (n= 0; n < NB_LOOP; n++)
    MimConvolve(MilDisplayImage, MilImage, MilKernel);
MappTimer(M_TIMER_READ+M_SYNCHRONOUS, &Time);

/* Pause to show the result. */
MosPrintf(MIL_TEXT("Convolve time: %.3f ms.\n"), Time*1000/NB_LOOP);
MosPrintf(MIL_TEXT("Press <Enter> to terminate.\n"));
MosGetch();

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

return 0;
}

```

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()** function supports different types of morphological operations:

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

You specify the required geometric relationships for each of these operations 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 the *Custom spatial filters* section earlier in this chapter).

- 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 the *Custom spatial filters* section earlier in this chapter) and the position of the neighborhood's center pixel.

Erosion and dilation

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

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

Original image

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

Eroded image

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

Dilated image

1	1	1
1	1	1
1	1	1

Structuring element

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

There are two versions of erosion and dilation:

- Erosion (**M_ERODE**).
 - **Binary erosion.** If the structuring element does not match the corresponding neighborhood values exactly, the center pixel is set to zero; otherwise, it remains unchanged. In effect, binary erosion peels off layers of objects.
 - **Grayscale erosion.** Subtracts each structuring element value from the corresponding pixel value in the neighborhood, and then replaces the center pixel of the neighborhood with the minimum value from the resulting neighborhood values.
- Dilation (**M_DILATE**).
 - **Binary dilation.** If any of the structuring element values match the corresponding neighborhood values, the center pixel is set to the maximum value of the buffer (e.g. 0xff for an 8-bit buffer); otherwise, it remains unchanged. In effect, binary dilation adds layers to the objects.
 - **Grayscale dilation.** Adds each structuring element value to the corresponding pixel value in the neighborhood, and then replaces the center pixel of the neighborhood with the maximum value from the resulting neighborhood values.

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

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

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

Using standard erosion and dilation

MIL also supports **MimErode()** and **MimDilate()**, functions 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 functions 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: MImMorphic.cpp
*
* Synopsis: This program loads an image of some tissue and then do opening on
*           it using two methods to show the differences.
*/
#include <mil.h>
#include <conio.h>

/* Target MIL image file specifications. */
#define IMAGE_FILE           M_IMAGE_PATH MIL_TEXT("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 MosMain(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;    /* Structing 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, M_NULL);

    /* Allocate an image buffer, load and displays it. */
    MbufAlloc2d(M_DEFAULT, IMAGE_WIDTH*2, IMAGE_HEIGHT*2,
                8+M_UNSIGNED, M_IMAGE+M_PROC+M_DISP, &MilImage);

```

```

MbufClear(MilImage, 0);
MdispSelect(MilDisplay, 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 4 processing buffers in the display buffer, restricting the
 * regions to be processed to the top left corner of the original image.
 */
MbufChild2d(MilImage, 0L, 0L, IMAGE_WIDTH, IMAGE_HEIGHT, &MilSubImage0);
MbufChild2d(MilImage, IMAGE_WIDTH, 0L, IMAGE_WIDTH, IMAGE_HEIGHT, &MilSubImage1);
MbufChild2d(MilImage, 0L, IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_HEIGHT, &MilSubImage2);
MbufChild2d(MilImage, IMAGE_WIDTH, IMAGE_HEIGHT,
            IMAGE_WIDTH, IMAGE_HEIGHT, &MilSubImage3);

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

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

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

/* Pause to show the original image. */
MosPrintf(MIL_TEXT("\nThis program does the opening of an image using two different\n"));

MosPrintf(MIL_TEXT("structuring elements.\nPress <Enter> to continue.\n\n"));
MosGetch();

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

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

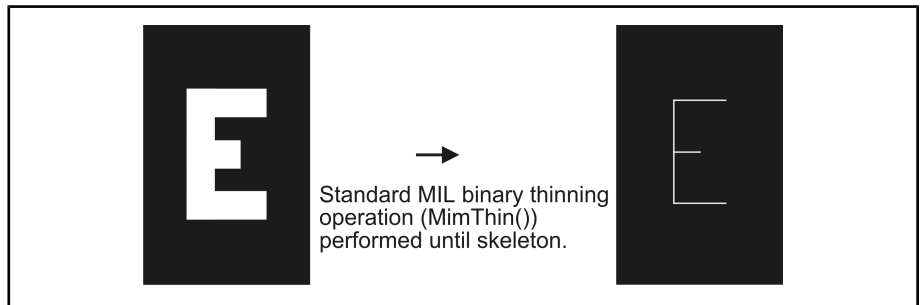
/* Free the allocated buffers. */
MbufFree(MilSubImage0);
MbufFree(MilSubImage1);
MbufFree(MilSubImage2);
MbufFree(MilSubImage3);
MbufFree(BinImage);

/* 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 different directions.

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

There are two versions of thinning and thickening:

- 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 ($\text{MAX}(0) < \text{center pixel} \leq \text{MIN}(1)$) then ($\text{center pixel} = \text{MAX}(0)$) 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 (**M_THICK**).

- **Binary thickening.**

This operation replaces the center pixel by the maximum possible 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 ($\text{MAX}(0) \leq \text{center pixel} < \text{MIN}(1)$) then ($\text{center pixel} = \text{MIN}(1)$) 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, or $\text{MIN}(1)$ 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 **M_DONT_CARE** values.

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

Matching

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

Searching for hits or misses

You can determine which pixels have neighborhoods that match a pattern exactly by performing a 'hit or miss' operation. When the neighborhood of a source image's pixel matches the pattern exactly, the value of the corresponding pixel in the destination image is the maximum value of the buffer (e.g. 0xff for an 8-bit buffer). 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

When an image must undergo processing with several structuring elements, you can increase the efficiency of your program by reducing such serial operations to a single parallel operations by using the **MimConnectMap()** function.

The **MimConnectMap()** function calculates a 9-bit connectivity code for each pixel in a binary source image based on its neighboring pixels and then maps these codes through the specified custom LUT. The specified LUT should contain values that would result if the required structuring elements were applied consecutively. Since each connectivity code has 9 bits, you should supply a LUT buffer with at least 512 (2^9) entries.

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.

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

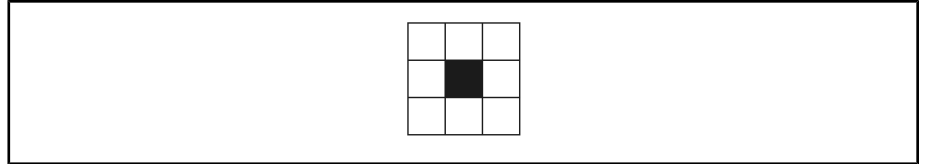
The pixels are connected and mapped as follows:

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

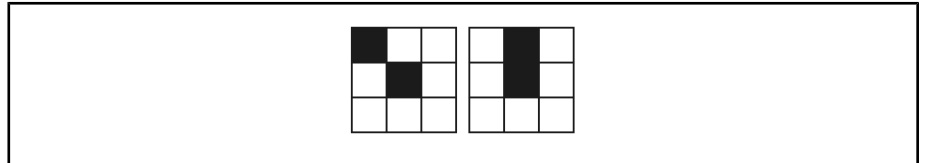
$$\text{Result} = \text{LUTMAP}[\text{Connectivity code}].$$

For example, you might need to isolate specific points of a binary, skeletonized image. The following are examples of types of points which can be located more efficiently using connectivity mapping:

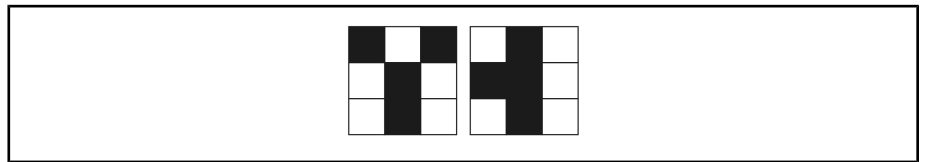
- **Isolated points.** Single points that are standalone.



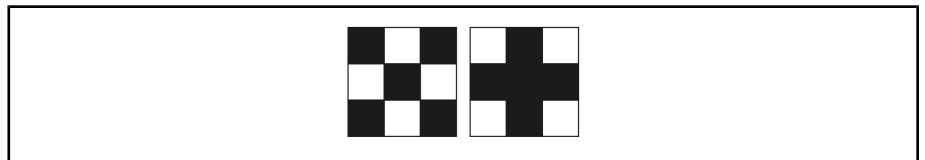
- **End points.** Points that represent the end of a line segment.



- **Triple points.** Points of intersection which occur at the common junction of 3 line segments.



- **Cross points.** Points of intersection which occur at the common junction of 4 line segments.



The source pixels are treated as binary (that is, all non-zero pixels are treated as 1). For example, the connectivity code for the 3x3 neighborhood of an isolated point is 10000000. For end points, two possible connectivity codes are 100001000 and 10000100. For triple points, two possible connectivity codes are 101001010 and 101010100. Two possible connectivity codes for cross points are 110101010 and 101010101.

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). You 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.

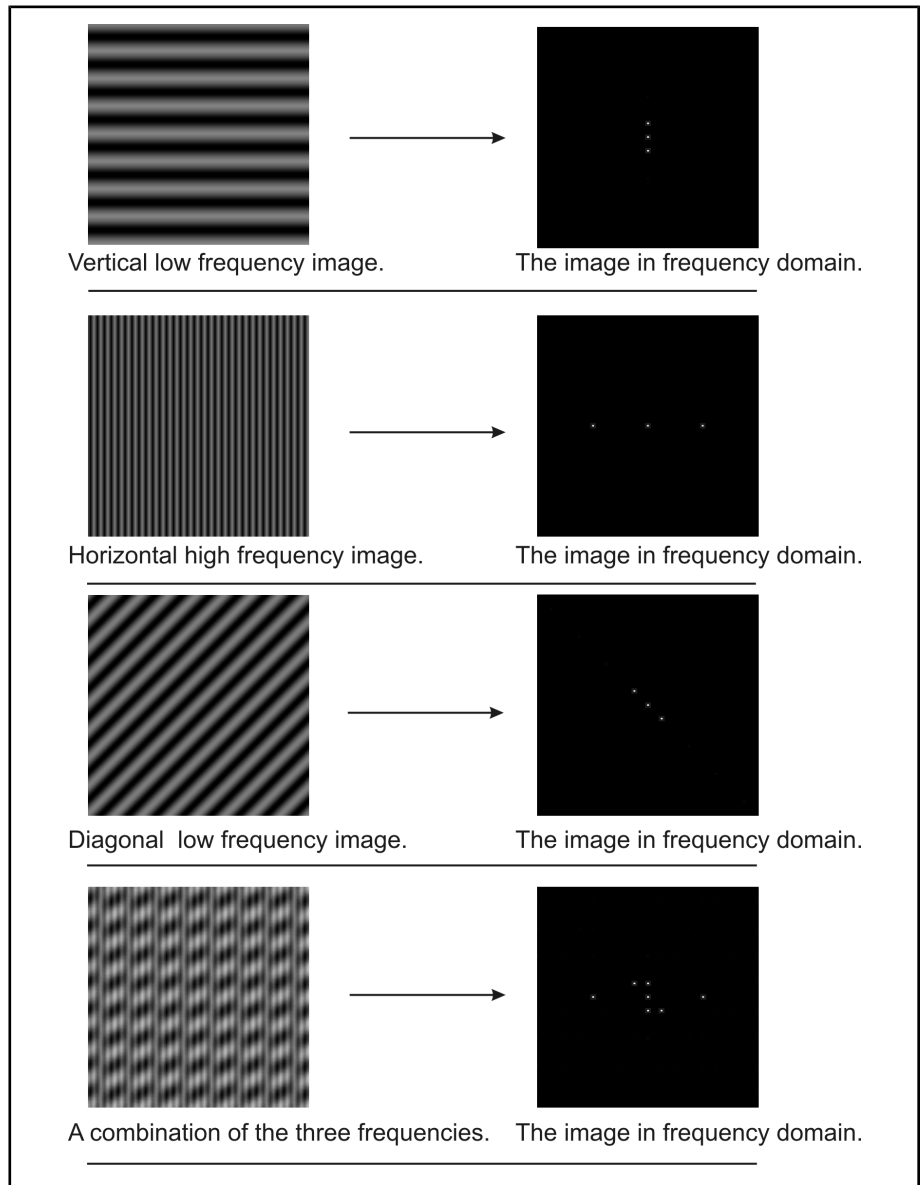
Magnitude and phase

For a more visual understanding of the FFT results, you can calculate the phase and magnitude, using the **MimTransform()** function with **M_MAGNITUDE**. The magnitude is calculated as:

$$\sqrt{R^2 + I^2}$$

where R and I are real and imaginary components of the image, respectively.

The following figures show single-frequency images and their centered magnitude. Since 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 DC component represents the frequency of the pattern. The DC component appears at the center of the image when **M_CENTER** is used.



The spatial shift of each pattern in the image, in degrees, is called the phase. It is calculated using the formula $\text{atan}(I/R)$. You can obtain the phase using the **M_PHASE** setting.

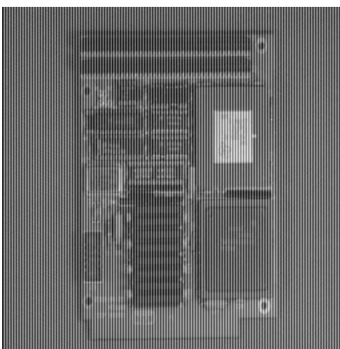
Filtering an image

To filter constant spatial patterns using FFTs:

1. Perform a forward FFT (**M_FORWARD**), calculating the magnitude (**M_MAGNITUDE**) of the image. Scale the image within a displayable range using **M_LOG_SCALE** (this applies the formula, $\log[1+|F(u,v)|]$).
2. Find the frequency components representing the noise and design a mask image to remove these components. To design the mask image, clear the mask to 255, then set the unwanted frequencies to 0.
3. Perform a simple FFT to obtain the real and imaginary components of the image, this time without calculating the magnitude.
4. Perform an AND operation between the mask and both the real and imaginary components of the image to remove the unwanted frequencies.
5. Finally, perform a reverse transform to obtain a filtered image.

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

Image (with noise pattern)



Frequency domain

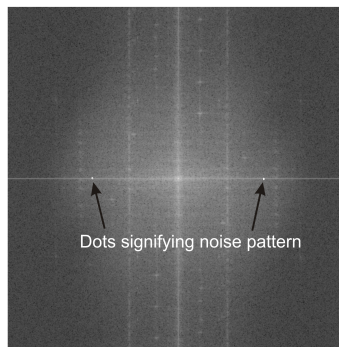
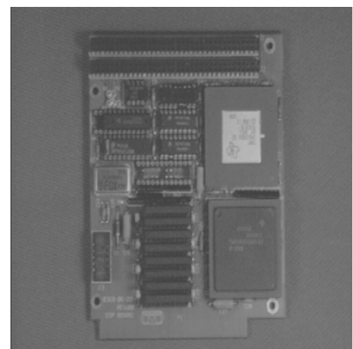


Image (noise pattern removed)



Following is an excerpt from the example *MImFFT.cpp*. 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: MImFFT.cpp
*
* Synopsis: This program uses the Fast Fourier Transform to filter an image.
*/
#include <mil.h>
#include <math.h>

/* Target image specifications. */
#define NOISY_IMAGE           M_IMAGE_PATH MIL_TEXT("noise.mim")
#define IMAGE_WIDTH           256
#define IMAGE_HEIGHT          256
#define X_NEGATIVE_FREQUENCY_POSITION 63
#define X_POSITIVE_FREQUENCY_POSITION 191
#define Y_FREQUENCY_POSITION  127
#define CIRCLE_WIDTH          9

int MosMain(void)
{
    MIL_ID MilApplication, /* Application identifier. */
    MilSystem, /* System identifier. */
    MilDisplay, /* Display identifier. */
    MilImage, /* Image buffer identifier. */
    MilOverlayImage, /* Overlay image buffer identifier. */
    MilSubImage00, /* Child buffer identifier. */
    MilSubImage01, /* Child buffer identifier. */
    MilSubImage10, /* Child buffer identifier. */
    MilSubImage11, /* Child buffer identifier. */
    MilTransformReal, /* Real part of the transformed image. */
    MilTransformIm; /* Imaginary part of the transformed image. */

    float ZeroVal = 0.0;

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

    /* Allocate a display buffer and clear it. */
    MbufAlloc2d(MilSystem, IMAGE_WIDTH*2, IMAGE_HEIGHT*2,
                8+M_UNSIGNED, M_IMAGE+M_PROC+M_DISP, &MilImage);
    MbufClear(MilImage, 0L);

    /* Display the image buffer and prepare for overlay annotations. */
    MdispSelect(MilDisplay, MilImage);

```

```

MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

/* Allocate child buffers in the 4 quadrants of the display image. */
MbufChild2d(MilImage, 0L, 0L, IMAGE_WIDTH, IMAGE_HEIGHT, &MilSubImage00);
MbufChild2d(MilImage, IMAGE_WIDTH, 0L, IMAGE_WIDTH, IMAGE_HEIGHT, &MilSubImage01);
MbufChild2d(MilImage, 0L, IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_HEIGHT, &MilSubImage10);
MbufChild2d(MilImage, IMAGE_WIDTH, IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_HEIGHT,
            &MilSubImage11);

/* Allocate processing buffers. */
MbufAlloc2d(MilSystem, IMAGE_WIDTH, IMAGE_HEIGHT, 32+M_FLOAT,
            M_IMAGE+M_PROC, &MilTransformReal);
MbufAlloc2d(MilSystem, IMAGE_WIDTH, IMAGE_HEIGHT, 32+M_FLOAT,
            M_IMAGE+M_PROC, &MilTransformIm);

/* Load a noisy image. */
MbufLoad(NOISY_IMAGE, MilSubImage00);

/* Print a message on the screen. */
MosPrintf(MIL_TEXT("\nFFT:\n"));
MosPrintf(MIL_TEXT("----\n\n"));
MosPrintf(MIL_TEXT("The frequency spectrum of a noisy image will be computed ")
            MIL_TEXT("to remove the periodic noise.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* The image is Fourier transformed to obtain the magnitude of the
   spectrum. This result will be used to design the filter. */
MimTransform(MilSubImage00, M_NULL, MilTransformReal, MilTransformIm,
            M_FFT, M_FORWARD+M_CENTER+M_MAGNITUDE+M_LOG_SCALE);
MbufCopy(MilTransformReal, MilSubImage10);

/* Draw circles in the overlay around the points of interest. */
MbufCopy(MilTransformReal, MilSubImage11);
MgraColor(M_DEFAULT, M_COLOR_YELLOW);
MgraArc(M_DEFAULT, MilOverlayImage, X_NEGATIVE_FREQUENCY_POSITION,
        Y_FREQUENCY_POSITION+IMAGE_HEIGHT, CIRCLE_WIDTH, CIRCLE_WIDTH, 0, 360);
MgraArc(M_DEFAULT, MilOverlayImage, X_POSITIVE_FREQUENCY_POSITION,
        Y_FREQUENCY_POSITION+IMAGE_HEIGHT, CIRCLE_WIDTH, CIRCLE_WIDTH, 0, 360);
MgraArc(M_DEFAULT, MilOverlayImage, X_NEGATIVE_FREQUENCY_POSITION+IMAGE_WIDTH,
        Y_FREQUENCY_POSITION+IMAGE_HEIGHT, CIRCLE_WIDTH, CIRCLE_WIDTH, 0, 360);
MgraArc(M_DEFAULT, MilOverlayImage, X_POSITIVE_FREQUENCY_POSITION+IMAGE_WIDTH,
        Y_FREQUENCY_POSITION+IMAGE_HEIGHT, CIRCLE_WIDTH, CIRCLE_WIDTH, 0, 360);

/* Put zero in the spectrum where the noise is located. */
MbufPut2d(MilSubImage11, X_NEGATIVE_FREQUENCY_POSITION,
        Y_FREQUENCY_POSITION, 1, 1, &ZeroVal);
MbufPut2d(MilSubImage11, X_POSITIVE_FREQUENCY_POSITION,
        Y_FREQUENCY_POSITION, 1, 1, &ZeroVal);

/* 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, X_NEGATIVE_FREQUENCY_POSITION,
          Y_FREQUENCY_POSITION, 1, 1, &ZeroVal);
MbufPut2d(MilTransformReal, X_POSITIVE_FREQUENCY_POSITION,
          Y_FREQUENCY_POSITION, 1, 1, &ZeroVal);
MbufPut2d(MilTransformIm, X_NEGATIVE_FREQUENCY_POSITION,
          Y_FREQUENCY_POSITION, 1, 1, &ZeroVal);
MbufPut2d(MilTransformIm, X_POSITIVE_FREQUENCY_POSITION,
          Y_FREQUENCY_POSITION, 1, 1, &ZeroVal);

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

/* Print a message. */
MosPrintf(MIL_TEXT("The frequency components of the noise are located ")
          MIL_TEXT("in the center of the circles.\n"));
MosPrintf(MIL_TEXT("The noise was removed by setting these frequency ")
          MIL_TEXT("components to zero.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to end.\n\n"));
MosGetch();

/* Free buffers. */
MbufFree(MilSubImage00);
MbufFree(MilSubImage01);
MbufFree(MilSubImage10);
MbufFree(MilSubImage11);
MbufFree(MilImage);
MbufFree(MilTransformReal);
MbufFree(MilTransformIm);

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

return 0;
}

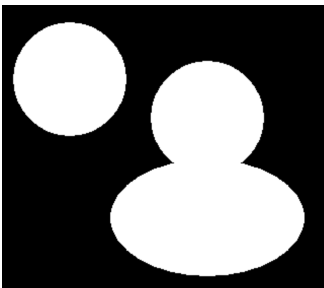
```

Watershed transformations

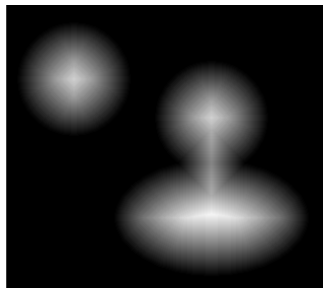
You can perform watershed transformations using **MimWatershed()**. A watershed transformation is generally used in conjunction with other processing operations to segment images, that is, to separate objects from their background and/or from each other.

To understand what a watershed transformation is, it is useful to think of an image as a topographic surface with hills and valleys. 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 from which, if a drop of water were to fall, it would eventually reach that minimum.

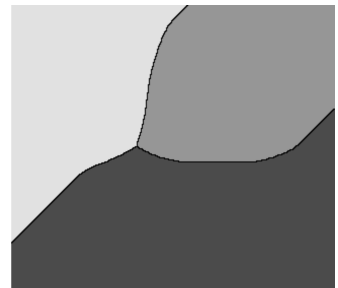
MimWatershed() labels an image's catchment basins and/or builds dividing lines between the catchment basins. These dividing lines are known as the watershed lines of the image. Note that catchment basins can be determined from the image's minima or its maxima.



An image with three blobs.



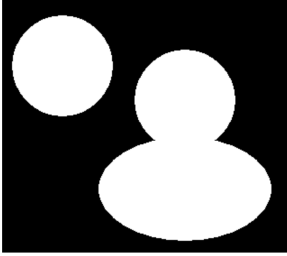
A distance transformation produces a maximum in each blob.



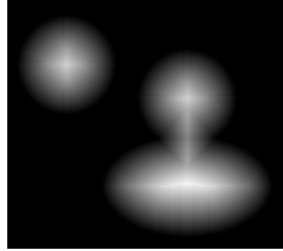
The watershed lines and labelled catchment basins of the resulting image.

Using watersheds to separate touching objects

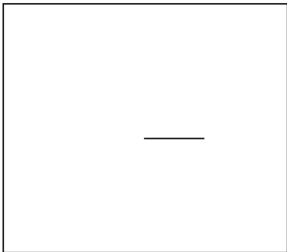
You can use `MimWatershed()` in conjunction with `MimDistance()` and `MimArith()` to separate touching objects in a binary image.



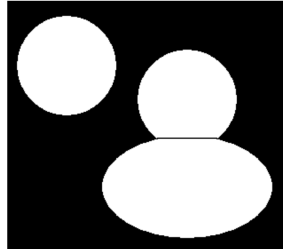
a) Touching blobs in a binary image.



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



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



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

To summarize:

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

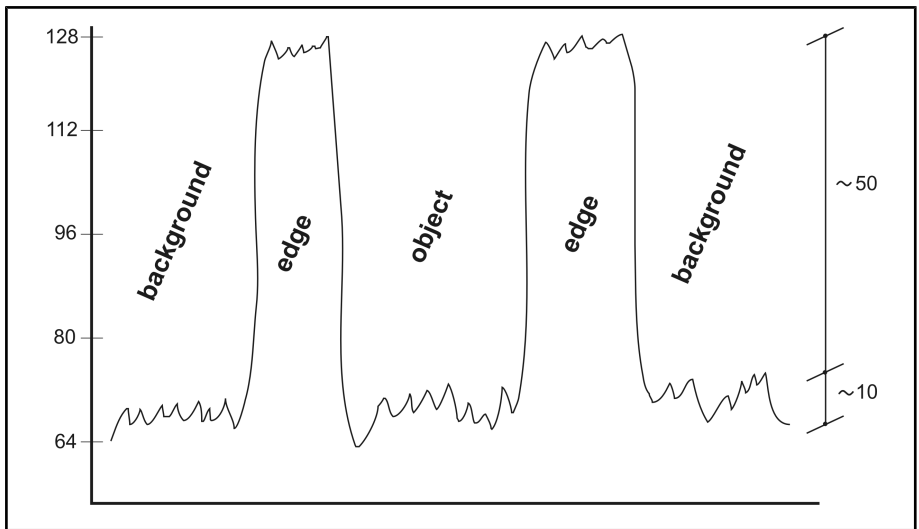
1. Perform an edge detection on the image.
2. Determine, through some analysis of the resulting image, the minimum grayscale variation that is required to produce a new catchment basin (this is discussed in the next section).
3. Perform a watershed transformation on the resulting image. You must specify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

Minimum grayscale variation of a catchment basin

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. To prevent over-segmentation while still separating objects from their background, you can specify the minimum grayscale variation of a catchment basin, using the **MinimumVariation** parameter of **MimWatershed()**.

The minimum variation of a catchment basin can be more easily understood using the topographical surface analogy. When dealing with minima, the minimum grayscale variation can be thought of as the minimum depth of the valley required to produce a catchment basin. The reverse applies when dealing with maxima.

The following shows a line profile across an object in an image (after an edge detection is performed on the image). In this case, the maximum difference between gray levels in the background (as well as within the object) is about 10, and the minimum difference 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 preprocessing 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 the extrema in the source image. Marker images are useful in preventing over-segmentation since you control not only the location of the extrema but also the number of extrema.

If you use a marker image, you can set the **MinimumVariation** parameter to **M_OFF**, **M_NULL** or 0, since you mark off the extrema in a separate image.

There are two types of marker images: non-labeled and labeled.

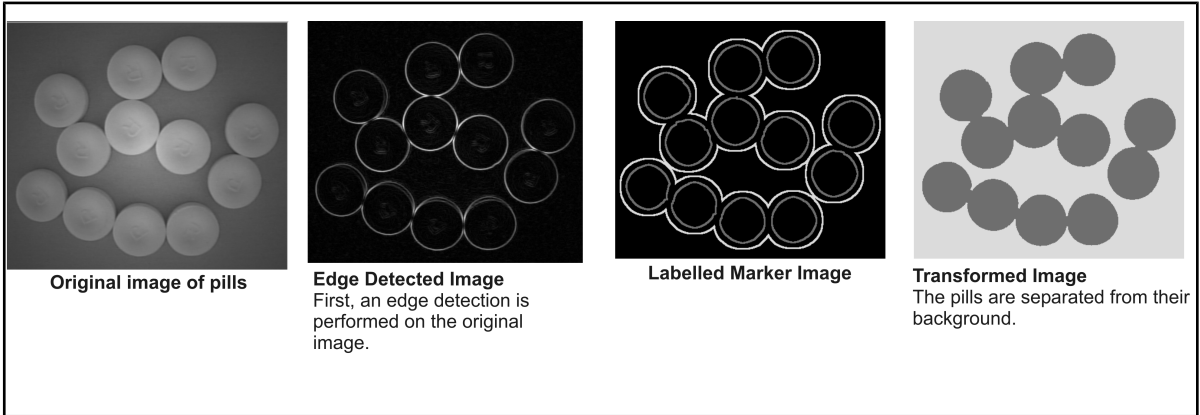
Non-labeled marker images

In a non-labeled marker image, each group of touching pixels with the value zero in the marker image, known as a marker, starts a catchment basin in the corresponding area of the source image. Specifically, each marker in the marker image forces an extremum 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".

Labeled marker images

In a labeled marker image, a marker is a set of pixels that do not have to necessarily touch and that have all the same label value (pixel intensity). Each marker starts a catchment basin in the corresponding area of the source image and each catchment basin of the destination image is assigned the label value of the marker that generated it. Note that, in a labeled marker image, a marker can touch other markers, allowing you to specify adjacent catchment basins. Valid marker label values are 1 to $2^n - 2$ (where n is the marker image buffer depth). Pixels with the label value of $2^n - 1$ are considered to be part of a "don't care" mask and are not processed, accelerating the watershed transformation. Finally, pixels with the label value of zero are interpreted as not being part of a marker. To use a labeled marker image, you must add **M_LABELED_MARKER** to the **ControlFlag** parameter.

The following is an example that performs a watershed transformation using a labeled marker image. It will separate the pills from the background.



To summarize:

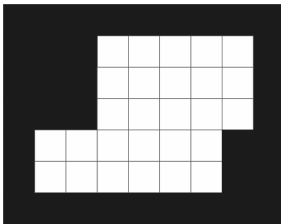
1. Perform an edge detection on the original image.
2. Create a marker image that is able to locate your objects in the image.
 - For this example, the labeled marker image is created from the edge detected image using a series of morphology, edge detection, and blob analysis operations.
 - In this example, the almost-white boundary surrounding all the rings is one marker. This marker serves to define the background and is placed in the corresponding area of the background of the original image.
 - All the rings in the marker image are considered to be one marker. This particular marker serves to identify the pills. These rings are smaller than the pills in the original image and are located in the corresponding areas of the pills.
3. Perform the watershed transformation on the edge detected image. You must specify that the catchment basins be determined from the image's minima. Remark how the resulting catchment basins have the same label value as the marker that generated them.

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 grayscale variation in the source image required to produce a new catchment basin.

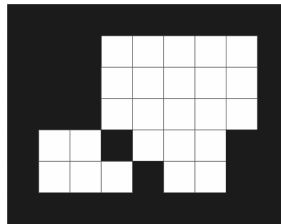
Style of the watershed lines

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**).

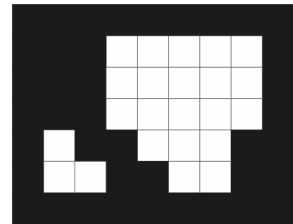
8-connected watershed lines consist of pixels that are horizontally, vertically, or diagonally touching. 4-connected watershed lines consist of pixels that are just horizontally and/or vertically touching. 8-connected watershed lines can separate 4-connected blobs, that is, blobs whose pixels can touch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



A white blob on a black background.



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



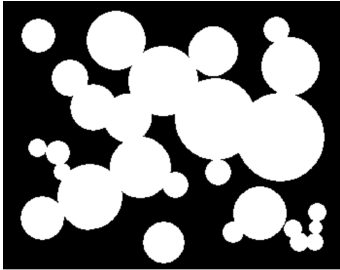
If the blob is 8-connected, the above 4-connected watershed line will separate it.

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

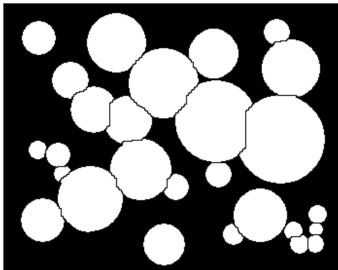
Note that 4-connected watershed lines can also separate 4-connected blobs, however they cause over-separation.

Exact versus straight

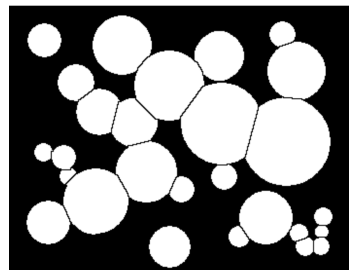
For visual purposes, watershed lines can be traced exactly or forced to be straight.



Original image.



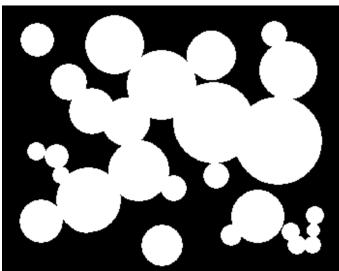
Exactly-traced watershed lines.



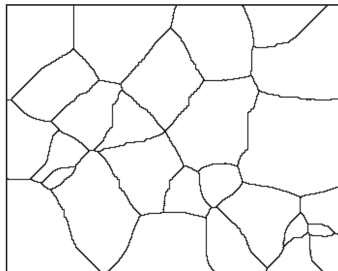
Straight watershed lines.

Skipping the last level

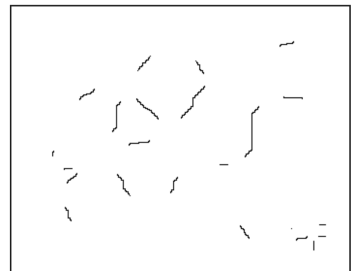
When you perform `MimWatershed()`, you can skip the last intensity level of the transformation (by setting the `ControlFlag` parameter to `M_SKIP_LAST_LEVEL`). In other words, you can prevent an extremum's zone of influence from extending beyond $L_{\max} - 1$ (for a minimum) or $L_{\min} - 1$ (for a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



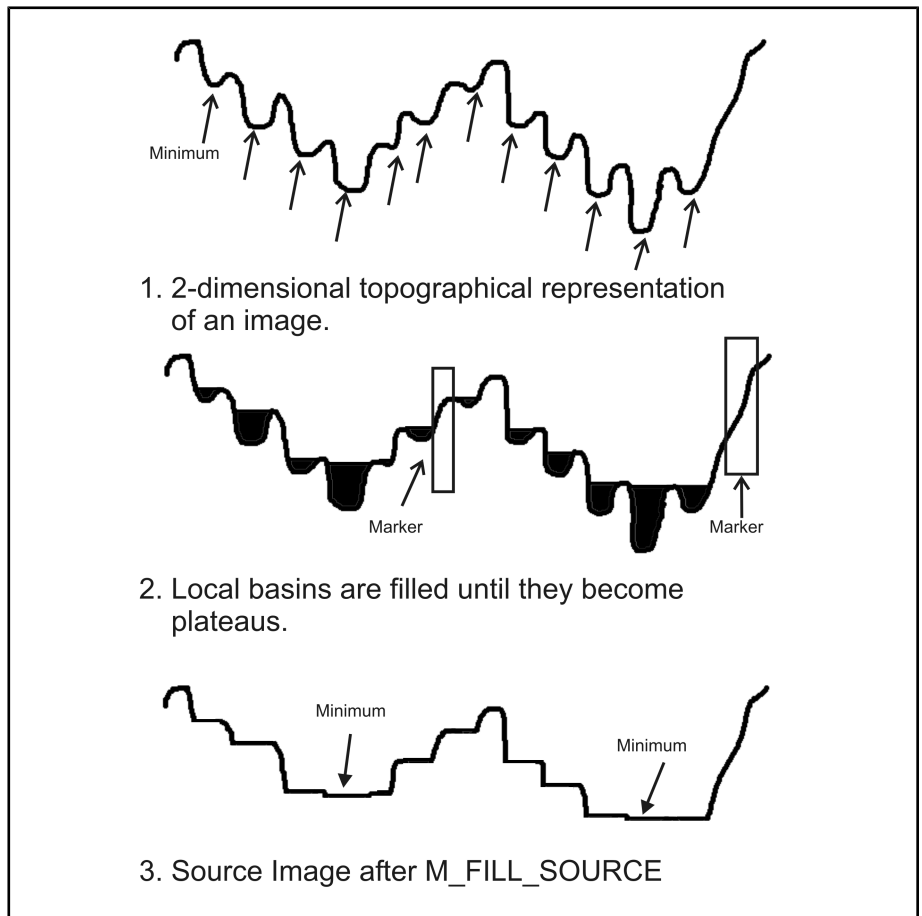
A watershed transform when the last level is skipped. The only watershed lines are those between touching objects.

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

Filling the source

When you perform **MimWatershed()**, you can fill the catchment basins of unwanted extrema in the source image by adding **M_FILL_SOURCE** to the **ControlFlag** parameter. **M_FILL_SOURCE** fills the catchment basins of the unwanted extrema until the local basins become plateaus. Any subsequent watershed transformation on the source image will no longer process the unwanted extrema because they are no longer extrema.

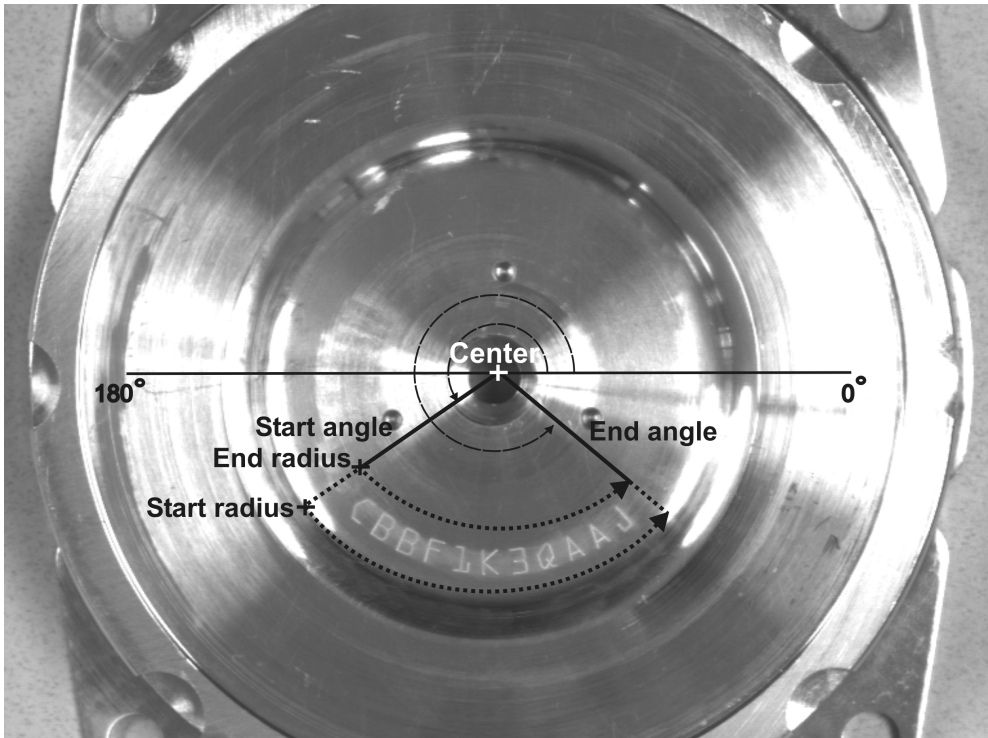
The following images illustrate how **M_FILL_SOURCE** transforms the source image:



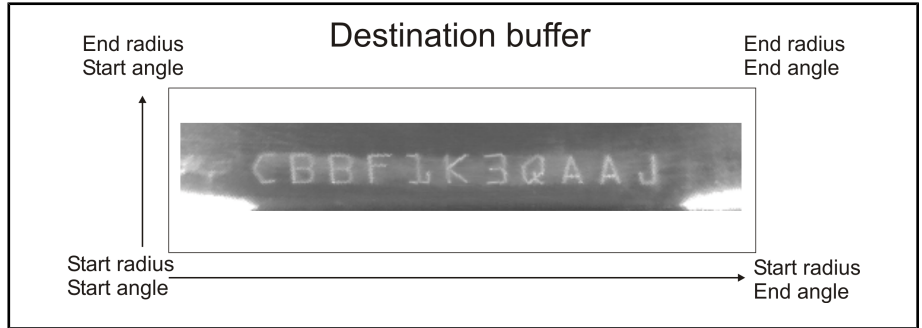
Polar-to-rectangular and rectangular-to-polar transforms

In a rectangular coordinate system, features or information which are curved can be awkward to read, interpret, or analyze. Transforming such features into a polar coordinate system, where the vertical and horizontal axes represent the radius and the angle respectively, can make it more intuitive or easier to read. With MIL, you can perform rectangular-to-polar or polar-to-rectangular transforms, using the `MimPolarTransform()` function.

For example, the text on the following image of a disk is easier to read after 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. The dotted lines define the borders of the zone of interest:



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



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:

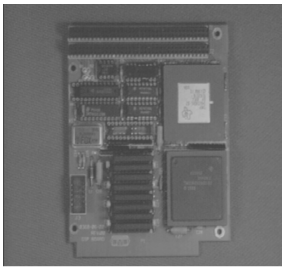
$$\Delta \text{ angle} = \frac{(\text{end angle} - \text{start angle})}{\text{arc length}}$$

The valid range of angles is from -360.0 to 360.0° and the maximum span of the angle must not exceed 360.0°. 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

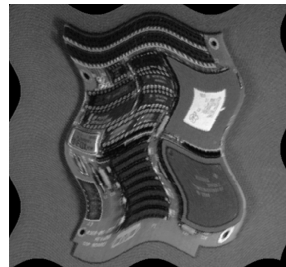
In addition to functions which perform specific geometric transforms (**MimFlip()**, **MimResize()**, **MimRotate()**, **MimTranslate()**, and **MimPolarTransform()**), MIL includes a more general geometric function, **MimWarp()**. It can perform any of the specific transforms, as well as complex warpings. Specifically, it can perform: first-order polynomial warpings, perspective polynomial warpings, and custom warpings.



Source image



Possible transforms using MimWarp()



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. In addition, geometric distortions can also be resolved using the Camera Calibration module. For more information, see *Chapter 5: Camera calibration*.

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 in two different ways:

Using a 3x3 coefficients matrix that is used as follows:

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

where

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

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

Using LUTs (an X-LUT and a Y-LUT) that are the same size as the destination image and are used as follows:

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

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

If you intend on performing the same warping operation on multiple images (which would require using the same LUTs), it might be faster to generate the LUTs for the operation once and repeatedly pass them to **MimWarp()**. However, if different warpings are required, or only one image is to be processed, it is faster to generate the coefficients and call **MimWarp()** than it is to generate the LUTs and call **MimWarp()**.

Example

A warping example, *MImWarp.cpp*, can be found in your examples directory.

```

/*****
/*
* File name: MImWarp.cpp
*
* Synopsis: : This program performs three types of warp transformations.
*           First the image is stretched according to four specified
*           reference points. Then it is warped on a sinusoid, and
*           finally, the program loops while warping the image on a
*           sphere.
*/

```

```

#include <mil.h>
#include <math.h>
#include <malloc.h>

/* Target image specifications. */
#define IMAGE_FILE          M_IMAGE_PATH MIL_TEXT("BaboonMono.mim")
#define INTERPOLATION_MODE  M_NEAREST_NEIGHBOR
#if(INTERPOLATION_MODE == M_NEAREST_NEIGHBOR)
    #define FIXED_POINT_PRECISION  M_FIXED_POINT+0L
    #define FLOAT_TO_FIXED_POINT(x)  (1L * (x))
#else
    #define FIXED_POINT_PRECISION  M_FIXED_POINT+6L
    #define FLOAT_TO_FIXED_POINT(x)  (64L * (x))
#endif
#define ROTATION_STEP      2

/* Utility functions. */
void YieldToGUI();
void GenerateSphericLUT(MIL_INT ImageWidth, MIL_INT ImageHeight,
                        short *MilLutXPtr, short *MilLutYPtr);

/* Main function. */
/* ----- */
int MosMain(void)
{
    MIL_ID    MilApplication, /* Application identifier. */
             MilSystem,      /* System identifier. */
             MilDisplay,     /* Display identifier. */
             MilDisplayImage, /* Image buffer identifier. */
             MilSourceImage,  /* Image buffer identifier. */
             Mil4CornerArray, /* Coefficients buffer identifier. */
             MilLutX,         /* Lut buffer identifier. */
             MilLutY,         /* Lut buffer identifier. */
             ChildWindow;     /* Child Image identifier. */

    float    FourCornerMatrix[12] = {
        0.0, /* X coordinate of quadrilateral's 1st corner */
        0.0, /* Y coordinate of quadrilateral's 1st corner */
        456.0, /* X coordinate of quadrilateral's 2nd corner */
        62.0, /* Y coordinate of quadrilateral's 2nd corner */
        333.0, /* X coordinate of quadrilateral's 3rd corner */
        333.0, /* Y coordinate of quadrilateral's 3rd corner */
        100.0, /* X coordinate of quadrilateral's 4th corner */
        500.0, /* Y coordinate of quadrilateral's 4th corner */
        0.0, /* X coordinate of rectangle's top-left corner */
        0.0, /* Y coordinate of rectangle's top-left corner */
        511.0, /* X coordinate of rectangle's bottom-right corner */
        511.0 }; /* Y coordinate of rectangle's bottom-right corner */

    MIL_INT Precision = FIXED_POINT_PRECISION;
    MIL_INT Interpolation = INTERPOLATION_MODE;
    short *MilLutXPtr, *MilLutYPtr;
    MIL_INT OffsetX = 0;
    MIL_INT ImageWidth, ImageHeight, ImageType, i, j;

```



```

MIL_DOUBLE   FramesPerSecond = 0, Time = 0, NbLoop = 0;

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

/* Restore the source image. */
MbufRestore(IMAGE_FILE, MilSystem, &MilSourceImage);

/* Allocate a display buffers and show the source image. */
MbufAlloc2d(MilSystem, MbufInquire(MilSourceImage, M_SIZE_X, &ImageWidth),
            MbufInquire(MilSourceImage, M_SIZE_Y, &ImageHeight),
            MbufInquire(MilSourceImage, M_TYPE, &ImageType),
            M_IMAGE+M_PROC+M_DISP, &MilDisplayImage);
MbufCopy(MilSourceImage, MilDisplayImage);
MdispSelect(MilDisplay, MilDisplayImage);

/* Print a message. */
MosPrintf(MIL_TEXT("\nWARPING:\n"));
MosPrintf(MIL_TEXT("-----\n\n"));
MosPrintf(MIL_TEXT("This image will be warped using different methods.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Four-corner LUT warping */
/*-----*/

/* Allocate 2 LUT buffers. */
MbufAlloc2d(MilSystem, ImageWidth, ImageHeight, 16L+M_SIGNED, M_LUT, &MilLutX);
MbufAlloc2d(MilSystem, ImageWidth, ImageHeight, 16L+M_SIGNED, M_LUT, &MilLutY);

/* Allocate the coefficient buffer. */
MbufAlloc2d(MilSystem, 12L, 1L, 32L+M_FLOAT, M_ARRAY, &Mil4CornerArray);

/* Put warp values into the coefficient buffer. */
MbufPut1d(Mil4CornerArray, 0L, 12L, FourCornerMatrix);

/* Generate LUT buffers. */
MgenWarpParameter(Mil4CornerArray, MilLutX, MilLutY,
                 M_WARP_4_CORNER+Precision, M_DEFAULT,
                 0.0, 0.0);

/* Clear the destination. */
MbufClear(MilDisplayImage, 0);

/* Warp the image. */
MimWarp(MilSourceImage, MilDisplayImage, MilLutX, MilLutY, M_WARP_LUT+Precision,
        Interpolation);

/* Print a message. */
MosPrintf(MIL_TEXT("The image was warped from an arbitrary"));
MosPrintf(MIL_TEXT(" quadrilateral to a square.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));

```

```

MosGetch();

/* Sinusoidal LUT warping */
/*-----*/

/* Allocate user-defined LUTs. */
MilLutXPtr = (short*)malloc(sizeof(short)*ImageHeight*ImageWidth);
MilLutYPtr = (short*)malloc(sizeof(short)*ImageHeight*ImageWidth);

/* Fill the LUT with a sinusoidal waveforms with a 6-bit precision.*/
for (j=0;j<ImageHeight;j++)
{
    for (i=0;i<ImageWidth;i++)
    {
        MilLutYPtr[i+ (j*ImageWidth)] =
            (short)FLOAT_TO_FIXED_POINT(((j) + (int)((20*sin(0.03*i)))));
        MilLutXPtr[i+ (j*ImageWidth)] =
            (short)FLOAT_TO_FIXED_POINT(((i) + (int)((20*sin(0.03*j)))));
    }
}

/* Put the values into the LUT buffers.*/
MbufPut2d(MilLutX, 0L, 0L, ImageWidth, ImageHeight, MilLutXPtr);
MbufPut2d(MilLutY, 0L, 0L, ImageWidth, ImageHeight, MilLutYPtr);

/* Clear the destination. */
MbufClear(MilDisplayImage,0);

/* Warp the image. */
MimWarp(MilSourceImage,MilDisplayImage,MilLutX,MilLutY,M_WARP_LUT +Precision,
        Interpolation);

/* wait for a key */
MosPrintf(MIL_TEXT("The image was warped on two sinusoidal waveforms.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Continuous spherical LUT warping */
/*-----*/

/* Allocate temporary buffer. */
MbufFree(MilSourceImage);
MbufAlloc2d(MilSystem, ImageWidth*2, ImageHeight, ImageType,
            M_IMAGE+M_PROC, &MilSourceImage);

/* Reload the image. */
MbufLoad(IMAGE_FILE, MilSourceImage);

/* Fill the LUTs with a sphere pattern with a 6-bit precision.*/
GenerateSphericLUT(ImageWidth, ImageHeight, MilLutXPtr,MilLutYPtr);
MbufPut2d(MilLutX, 0L, 0L, ImageWidth, ImageHeight, MilLutXPtr);
MbufPut2d(MilLutY, 0L, 0L, ImageWidth, ImageHeight, MilLutYPtr);

```

```

/* Duplicate the buffer to allow wrap around in the warping. */
MbufCopy(MilSourceImage, MilDisplayImage);
MbufChild2d(MilSourceImage, ImageWidth, 0, ImageWidth, ImageHeight, &ChildWindow);
MbufCopy(MilDisplayImage, ChildWindow);
MbufFree(ChildWindow);

/* Clear the destination. */
MbufClear(MilDisplayImage, 0);

/* Print a message and start the timer. */
MosPrintf(MIL_TEXT("The image is continuously warped on a sphere.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to stop.\n\n"));
MappTimer(M_TIMER_RESET+M_SYNCHRONOUS, M_NULL);

/* Warp the image continuously. */
while (!MosKbhit())
{
    /* Create a child in the buffer containing the two images. */
    MbufChild2d(MilSourceImage, OffsetX, 0, ImageWidth, ImageHeight, &ChildWindow);

    /* Warp the child in the window. */
    MimWarp(ChildWindow, MilDisplayImage, MilLutX, MilLutY,
            M_WARP_LUT+Precision, Interpolation);

    /* Update the offset (shift the window to the right). */
    OffsetX += ROTATION_STEP;

    /* Reset the offset if the child is outside the buffer. */
    if (OffsetX>ImageWidth-1)
        OffsetX = 0;

    /* Free the child. */
    MbufFree(ChildWindow);

    NbLoop++;

    /* Calculate and print the number of frames per second processed. */
    MappTimer(M_TIMER_READ+M_SYNCHRONOUS, &Time);
    FramesPerSecond = NbLoop/Time;
    MosPrintf(MIL_TEXT("Processing speed: %.0f Images/Sec.\n"), FramesPerSecond);
    YieldToGUI();
}
MosGetch();
MosPrintf(MIL_TEXT("\n"));

/* Free objects. */
free(MilLutXPtr);
free(MilLutYPtr);
MbufFree(MilLutX);
MbufFree(MilLutY);
MbufFree(Mil4CornerArray);
MbufFree(MilSourceImage);

```

```

MbufFree(MilDisplayImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);

return 0;
}

/* Generate two custom LUTs used to map the image on a sphere. */
/* ----- */
void GenerateSphericLUT(MIL_INT ImageWidth, MIL_INT ImageHeight,
                        short *MilLutXPtr, short *MilLutYPtr)
{
    MIL_INT      i, j, k;
    MIL_DOUBLE   utmp, vtmp, tmp;
    short        v;

    /* Set the radius of the sphere */
    MIL_DOUBLE   Radius = 200.0;

    /* Generate the X and Y buffers */
    for (j=0; j < ImageHeight; j++)
    {
        k = j*ImageWidth;

        /* Check that still in the sphere (in the Y axis). */
        if (fabs( vtmp = ((MIL_DOUBLE)(j - (ImageHeight/2)) / Radius) ) < 1.0)
        {
            /* We scan from top to bottom, so reverse the value obtained above
               and obtain the angle. */
            vtmp = acos( -vtmp );
            if(vtmp == 0.0)
                vtmp=0.0000001;

            /* Compute the position to fetch in the source. */
            v = (short)((vtmp/3.1415926) * (MIL_DOUBLE)(ImageHeight - 1) + 0.5);

            /* Compute the Y coordinate of the sphere. */
            tmp = Radius*sin(vtmp);

            for (i=0; i < ImageWidth; i++)
            {
                /* Check that still in the sphere. */
                if ( fabs(utmp = ((MIL_DOUBLE)(i - (ImageWidth/2)) / tmp)) < 1.0 )
                {
                    utmp = acos( -utmp);

                    /* Compute the position to fetch (fold the image in four). */
                    MilLutXPtr[i + k] = (short)FLOAT_TO_FIXED_POINT(((utmp/3.1415926) *
                        (MIL_DOUBLE)((ImageWidth/2) - 1) + 0.5));
                    MilLutYPtr[i + k] = (short)FLOAT_TO_FIXED_POINT(v);
                }
            }
        }
        else

```

```

        {
            /* Default position (fetch outside the buffer to
               activate the clear overscan). */
            MillutXPtr[i + k] = (short)FLOAT_TO_FIXED_POINT(ImageWidth);
            MillutYPtr[i + k] = (short)FLOAT_TO_FIXED_POINT(ImageHeight);
        }
    }
    else
    {
        for (i=0; i < ImageWidth ;i++ )
        {
            /* Default position (fetch outside the buffer for clear overscan). */
            MillutXPtr[i + k] = (short)FLOAT_TO_FIXED_POINT(ImageWidth);
            MillutYPtr[i + k] = (short)FLOAT_TO_FIXED_POINT(ImageHeight);
        }
    }
}

/* Windows CE GUI Scheduling Adjustment Handling */
/* ----- */
/* NOTE: Under Windows CE, YieldToGUI funtion improves system responsiveness
 *       in case a normal Windows CE console application thread is processing
 *       in a while loop.
 */
void YieldToGUI()
{
    #if M_MIL_USE_CE
        MosSleep(0);
    #endif
}

```

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:

$$x_s = a_0 x_d + a_1 y_d + a_2 .$$

$$y_s = b_0 x_d + b_1 y_d + b_2 .$$

Warping using a 3x3 coefficients matrix

To perform a first-order polynomial warping using a 3x3 coefficients matrix, you must pass the identifier of a 3x3 coefficients matrix buffer to **MimWarp()** in the **WarpParam1Id** parameter. The elements of the last row of the coefficient matrix should be 0, 0, 1. The coefficients of this matrix can be generated automatically using **MgenWarpParameter()** or can be user supplied. When using **MgenWarpParameter()**, specify how to perform the warping (for example, specify by how much to rotate and resize an image); the function then generates the coefficients required to produce such a warping. To combine coefficients, use separate calls to **MgenWarpParameter()**. For example, to generate coefficients for a rotation and translation, 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()**.

Warping using LUTs

When performing a first-order polynomial warping using LUTs, you can customize the values in your LUTs or alternatively, you can generate the LUTs automatically with **MgenWarpParameter()**. To create the LUTs with the **MgenWarpParameter()** function, you can either pass the same information as when generating the coefficients or pass the 3x3 coefficients matrix, itself. After the LUTs are generated, pass them to **MimWarp()**.

Perspective polynomial warpings

A perspective polynomial warping is used to map an arbitrary quadrilateral onto a rectangle or to map a rectangle onto an arbitrary quadrilateral.

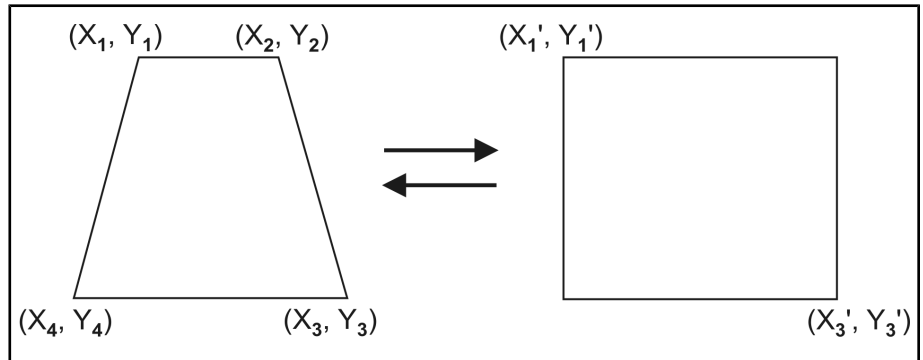
Perspective polynomial warpings are performed by associating points in the source buffer with pixels in the destination buffer according to the following equations:

$$x_s = (a_0 x_d + a_1 y_d + a_2) / (c_0 x_d + c_1 y_d + c_2).$$

$$y_s = (b_0 x_d + b_1 y_d + b_2) / (c_0 x_d + c_1 y_d + c_2).$$

Warping using a 3x3 coefficients matrix

To perform a perspective polynomial warping using a 3x3 coefficients matrix, you must pass the identifier of a 3x3 coefficients matrix to **MimWarp()**. The coefficients of this matrix can be generated automatically using **MgenWarpParameter()** or can be user supplied. When using **MgenWarpParameter()**, you must specify the coordinates of the four corners of the quadrilateral or the two opposite corners of the rectangle. The coordinates are illustrated in the image below.



After all coefficients are generated, pass the coefficient buffer to **MimWarp()** to perform the warping.

Warping using LUTs

When performing a perspective polynomial warping using LUTs, you can customize the values in your LUTs or alternatively, you can generate the LUTs automatically with **MgenWarpParameter()**. To create the LUTs with the **MgenWarpParameter()** function, you can either pass the same information as when generating coefficients or pass the 3x3 coefficients matrix, itself. After the LUTs are generated, pass them to **MimWarp()**.

Custom warplings

You can perform a custom warping using LUTs, since X_s and Y_s can be arbitrarily mapped. Load the X-LUT and Y-LUT with your custom values and pass them to **MimWarp()**.

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 (for an 8x8 matrix):

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

where u and v are coordinates in the frequency domain.

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

and

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

The reverse DCT is defined as:

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

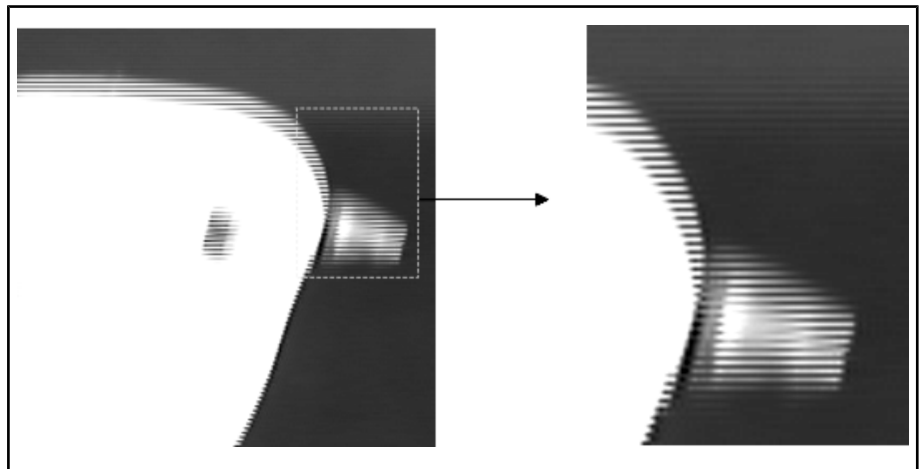
where x and y are coordinates in the spatial domain.

Frequency 0, also called the DC component, is plotted in the top-left corner of the spectrum. All other components in the spectrum are called AC components. A DCT concentrates the low frequency components of the image in the first few coefficients (top left-corner) of the spectrum. MIL divides the image into independent blocks of 8x8 pixels and performs the transform on each individual block. Centering of the spectrum is not supported in MIL.

Deinterlacing

Interlacing artifacts result when analyzing an image grabbed with an interlaced camera. To enhance the quality of the image, you can apply a deinterlacing algorithm using **MimDeinterlace()**.

To understand and differentiate between deinterlacing algorithms, knowledge of interlacing artifacts and the reason they occur in an image is important. When an interlaced camera acquires images, it captures two separate fields per frame. However, these fields are not acquired at the same moment in time. They are separated by $1/60^{\text{th}}$ of a second in NTSC or by $1/50^{\text{th}}$ of a second in PAL. In the event that an object was in motion when it was being acquired, its position would be slightly shifted from one field to the next, resulting in interlacing artifacts.



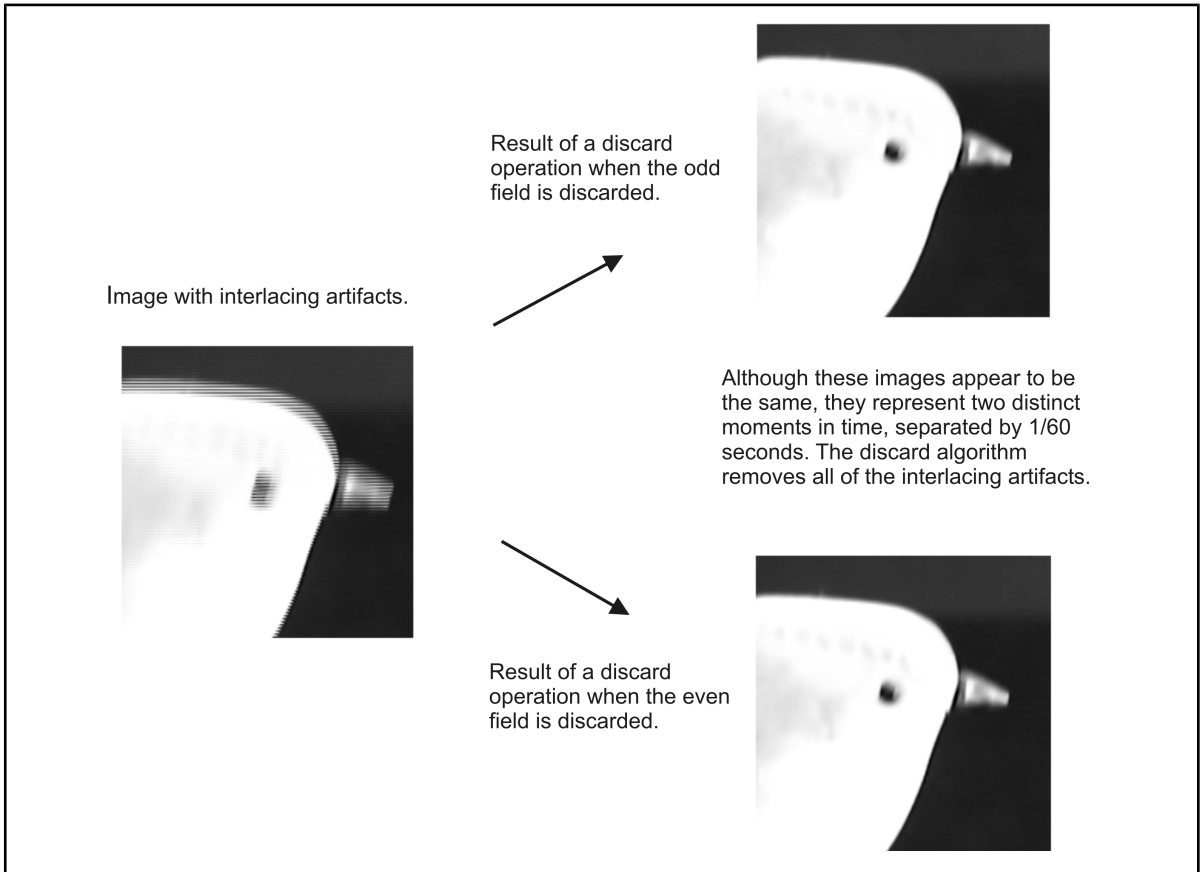
Taking a close look at the image, you can see a series of horizontal lines inside and around the edges of the bird. These lines are known as interlacing artifacts. The image is clearly formed by the superposition of two distinct fields, at two distinct moments in time.

MIL supports three different deinterlacing algorithms to reduce or remove interlacing artifacts: the discard, averaging, and bob algorithms. Note that interlacing artifacts are only present in objects in motion. MIL can apply these algorithms to all the pixels in an image or only to the pixels in the image that are part of objects in motion.

Discard algorithm on the entire image

The discard algorithm, the simplest deinterlacing algorithm, removes one field from the image (the odd or even rows) and interpolates the remaining rows. For example, if you choose to discard the odd rows, the first odd row would be the average of the first and second even rows. This algorithm is advantageous because it removes all the interlacing artifacts and has a fast processing time. However, if the sequence of deinterlaced images is viewed as a video, the motion in the sequence will not be as fluid as the original sequence because each discarded field represents a unique moment in time. Also, the areas in the image that did not contain interlacing artifacts will lose sharpness in their edges.

The following example depicts the result of the discard algorithm on an image containing interlacing artifacts.



Averaging algorithm on the entire image

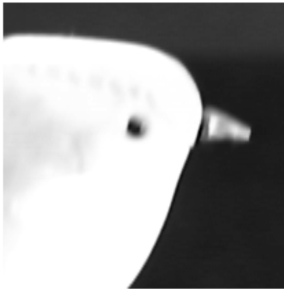
The averaging deinterlacing algorithm is equivalent to performing the discard algorithm once on the even field and once on the odd field and averaging the two resulting frames. In reality, MIL proceeds in a much more efficient way, making the processing speed of this function similar to the speed of the discard algorithm.

The interlacing artifacts are completely removed and this process preserves the fluidity of the video because complete fields are not discarded and therefore no individual moments in time are discarded. However, the images suffer from ghost effects (objects in motion appear in double) when looking at the stilled frames

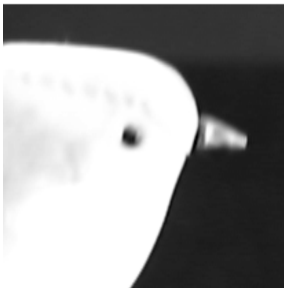
because two distinct moments in time are being merged together. These effects are barely noticeable when the images are viewed as a sequence. The loss of sharpness in background objects' edges also occurs when using this algorithm, but not as badly as with the discard algorithm.

The example below illustrates the ghost effects that arise from the averaging algorithm.

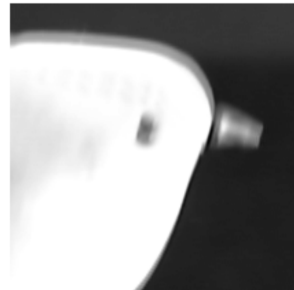
Result of a discard operation when the odd field is discarded.



Result of a discard operation when the even field is discarded.



The averaging algorithm takes these two images and averages them to form one output image.

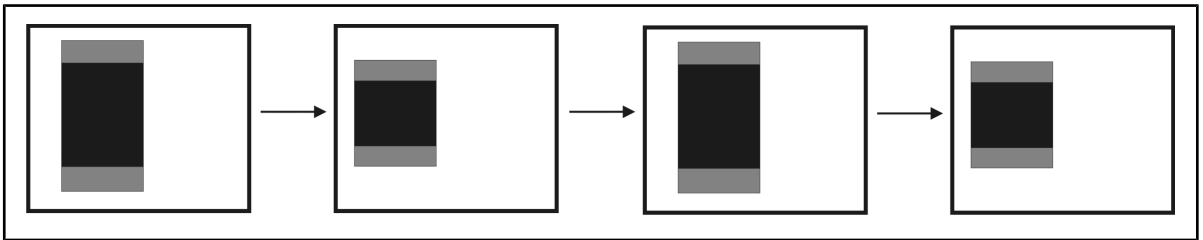


The ghost effects are only present around the edges of the bird (the object in motion).

Bob algorithm on the entire image

The bob algorithm applies the discard algorithm twice, once to each field. The two frames that result each become an output frame. Therefore, for every input frame, there are two output frames. This algorithm completely removes the interlacing artifacts and the motion in the video is fluid. The processing speed of this algorithm is also fast, although not as fast as the discard or averaging algorithms. As for the image quality, the same loss of sharpness in static objects' edges is observed as with the discard algorithm. In addition, if the sequence of deinterlaced images is viewed in motion, the edges of static objects flutter occasionally.

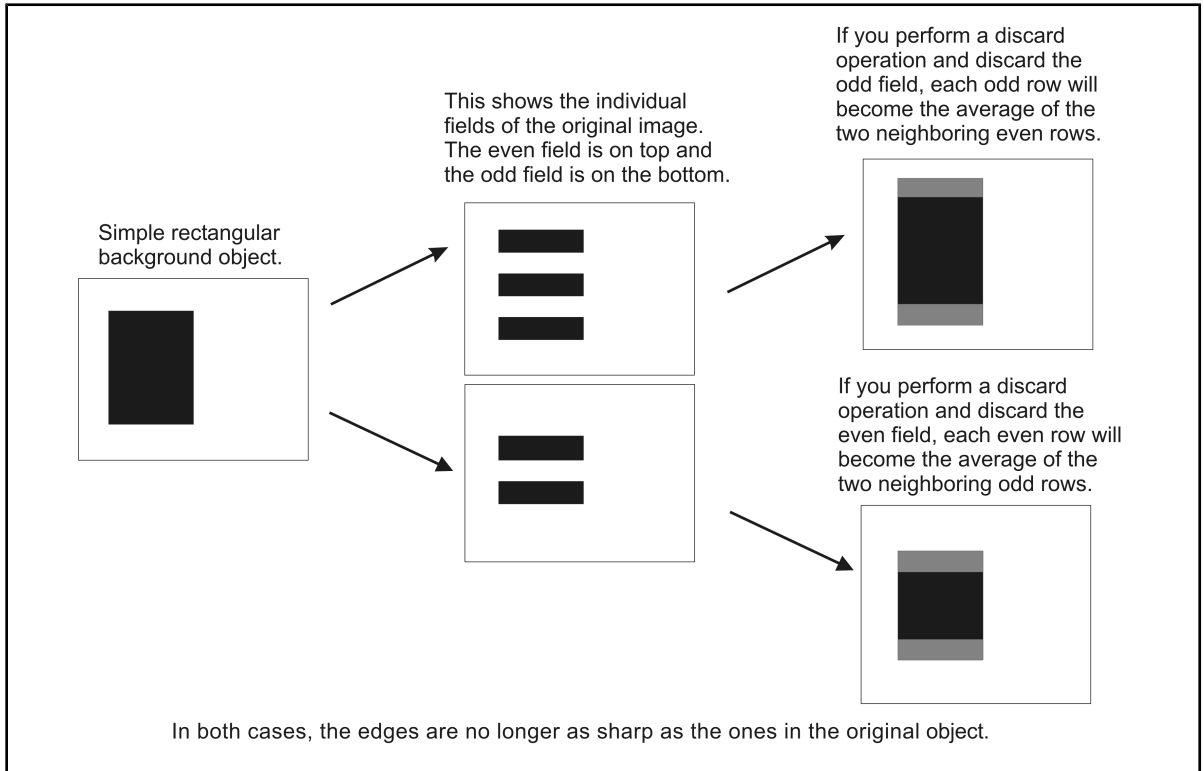
The illustration below shows a sequence of frames, each representing the same background object. This is an example of the fluttering that can occur when viewing a sequence of deinterlacing images.



If the frames are displayed very quickly ($1/30^{\text{th}}$ of a second on a progressive scan display), the top and bottom edges of the rectangle appear to be moving up and down, and it is this motion that we call fluttering.

Adaptive algorithms

MIL also supports an adaptive version of each algorithm: the adaptive discard, adaptive average, and adaptive bob algorithms. Since interlacing artifacts are caused by differences in successive frames and are only present in the areas of the image containing objects in motion, there is no need to apply the algorithms to all the pixels in the image; this actually results in a loss in the sharpness in the edges of static objects.

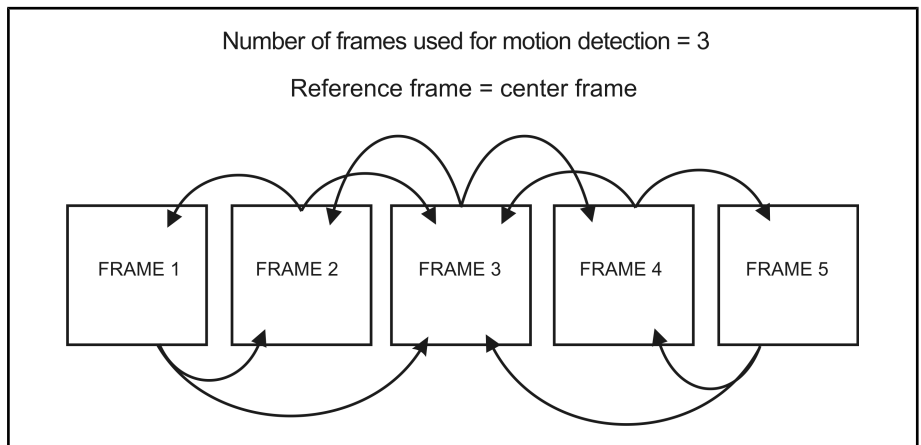


Therefore, prior to performing their respective deinterlacing operation, the adaptive algorithms first perform motion detection to determine the areas of the image depicting moving objects. To achieve this, each pixel in the frame to deinterlace is compared with the pixels at the same location in a group of neighboring frames. If the difference between the maximum and minimum pixel intensity is above a specified threshold value, the pixel in the frame to deinterlace is considered part

of an object in motion. If the difference is below the threshold value, the pixel is considered part of a background object. After all the pixel locations have been checked, the deinterlacing algorithm is applied to the pixels in motion and the background pixels remain untouched.

Note that the adaptive algorithms are more effective in preserving the sharpness and detail in background objects, however they can fail to remove some subtle interlacing artifacts. In addition, they have a longer processing time than their non-adaptive counterparts.

If an adaptive algorithm is selected, each frame in the input sequence of images will undergo motion detection. Using **MimControl()**, you can set the number of neighboring frames used for motion detection (**M_MOTION_DETECT_NUM_FRAMES**), the location of the frame to be processed within this group of frames (**M_MOTION_DETECT_REFERENCE_FRAME**), and the threshold value used to differentiate between pixels of objects in motion and pixels of background objects (**M_MOTION_DETECT_THRESHOLD**). Note that in the motion detection process, MIL will automatically adjust the index of the reference frame within the group of neighboring frames for border frames, when necessary. For example, suppose you have specified a group of 3 frames to be used for motion detection, with the reference frame set to the center frame. In an input sequence of 5 images, the following image illustrates which frames are used for motion detection.



The frames at the extremities of the sequence cannot be the center frame in a group of 3 frames (because they do not have an adjacent frame on both their left and right side). Therefore, MIL takes this into account and changes the reference frame to avoid these problematic situations. In the example, the first frame has no frame on its left. The reference frame will be changed to the left frame (instead of the center frame) and the motion detection will be performed using the two frames on the right.

The image below is the result of the motion detection process. The white pixels represent pixels of objects in motion and the black pixels represent pixels of background objects. To generate images like this one, you use **MimControl()** with **M_MOTION_DETECT_OUTPUT** set to **M_ENABLE**. Note that in this case, the deinterlacing algorithm will not be performed.



Chapter

5

Camera calibration

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

Camera calibration - overview

MIL's Camera Calibration module (**Mcal...**) allows 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.

Defining the pixel-to-world mapping is known as **camera 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:
 - 3D Reconstruction.
 - Blob Analysis.
 - Code.
 - Edge Finder.
 - Image Processing.
 - Measurement.

- Metrology.
 - Geometric Model Finder.
 - Optical Character Recognition.
 - Pattern Matching.
- ❖ 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 function description.

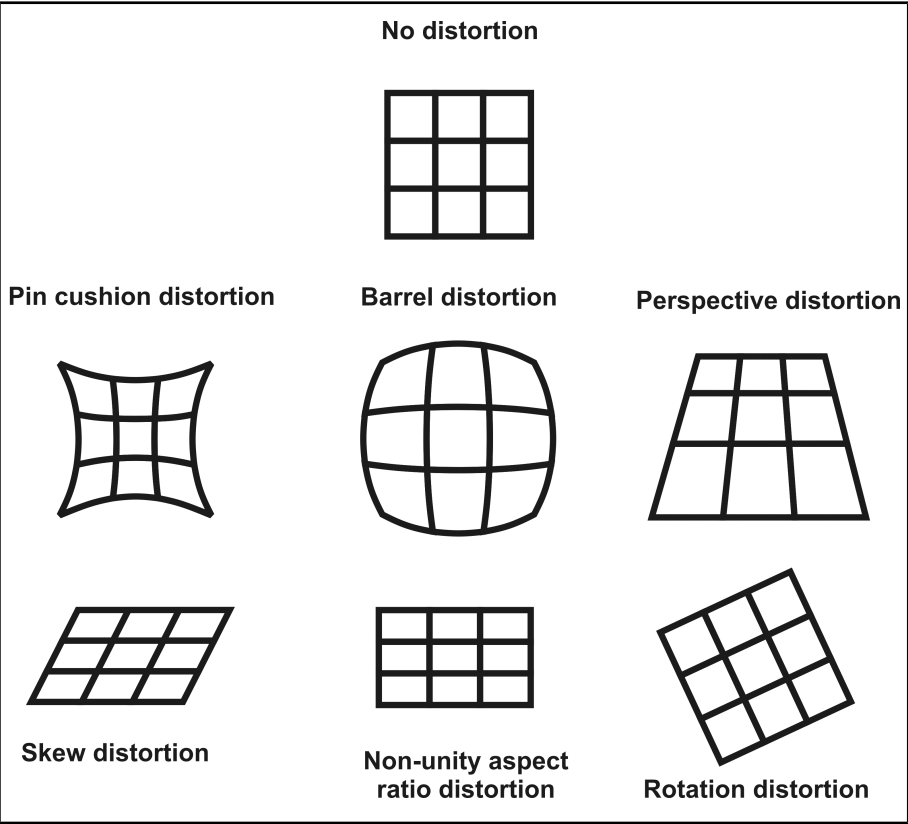
After calibration, pixels of images taken by your calibrated camera are mapped to one plane only in the real world. Using the Camera Calibration module solely is not enough to obtain data about the depth of different points in your images. To retrieve depth information from your two-dimensional images, you must perform a 3D Reconstruction.

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 frame grabbers.
- **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.

The following image gives examples of the different types of distortion:



Calibration mechanism

To perform the calibration, the module uses calibration points and a calibration mode. Calibration points are points in the image with known positions in the world. The module uses these points with the specified calibration mode to determine the world coordinates of all other points in the image. The calibration points can be explicitly specified or can be automatically calculated from an image of a grid and the world description of the grid. You can either use two-dimensional (2D) or three-dimensional (3D) calibration modes depending on the type of your imaging distortions and camera setup.

Two 3D-based calibration modes are available in MIL. The first is based on the technique developed by Roger Y. Tsai and the second is used for robotics camera setup that has the camera attached on a robotic arm. The position and orientation of the robot's arm are returned by the robot's controller and used to calibrate the camera's position on the robotic arm. Both 3D-based calibration modes support full 3D movement of your camera. However, all results are returned in a 2D coordinate system. This means that all points in an image are assumed to be on the same plane even if the plane is slanted.

Steps to performing a calibration

The following steps provide a basic methodology for using the MIL Camera Calibration module:

1. Allocate a calibration object, using **McalAlloc()**, and physically move the camera and place the working area in its field of view.
2. If calibrating in a 3D-based calibration mode (**M_TSAI_BASED** or **M_3D_ROBOTICS**), specify the required camera attributes using **McalControl()**. For example, specify the aspect ratio of the camera's CCD elements if the ratio is not 1 and specify the pixel coordinates of the principal point (the intersection of the camera's optical axis with the image plane).
3. If calibrating in **M_3D_ROBOTICS** mode, perform the following steps:
 - a. Get the position and orientation of the tool holding the camera, with respect to the robot base, using the robot software.
 - b. Set the position and orientation for the tool coordinate system in the calibration object with respect to the robot base coordinate system using **McalSetCoordinateSystem()**.
 - c. Call **McalGrid()** or **McalList()** with **M_ACCUMULATE** to store calibration points in the calibration object.

- d. Physically move the camera to new a position over the working area. You must rotate the tool along at least two non-parallel axes.
 - e. Repeat the previous steps at least twice, to make a minimum total of three calls with **M_ACCUMULATE**.
4. Calibrate your camera setup, using either **McalGrid()** or **McalList()** with **M_FULL_CALIBRATION** to map image pixels to world points. These mappings are stored in the calibration object.
5. Do one of the following:
 - To transform pixel coordinates or results to their real-world equivalents, use **McalTransformCoordinate()**, **McalTransformCoordinateList()**, **McalTransformCoordinate3dList()**, or **McalTransformResult()**.
 - To physically correct an image, use **McalTransformImage()**.
 - To obtain look-up tables (LUTs) from images and then transform them with **MimWarp()**, use **McalTransformImage()** with **M_EXTRACT_LUT_X** and **M_EXTRACT_LUT_Y**.
 - To automatically get results from other MIL modules in real-world units, associate the calibration object to an image or digitizer, using **McalAssociate()**.
6. If necessary, save your calibration object using **McalSave()** or **McalStream()**.

7. Free the calibration object using **McalFree()**.
- ❖ If you move either the camera or relative coordinate system (i.e. taking measurements in a different plane), you can remain calibrated in the following two cases:
 - If the movement is known, specify the transformation, using **McalSetCoordinateSystem()** with the camera coordinate system or the relative coordinate system as the target coordinate system.
 - If the movement is not known and using 3D-based calibration object, you can have the new position calculated automatically using **McalGrid()** or **McalList()** with **M_DISPLACE_CAMERA_COORD** or **M_DISPLACE_RELATIVE_COORD**, depending on what you have moved.

Basic concepts

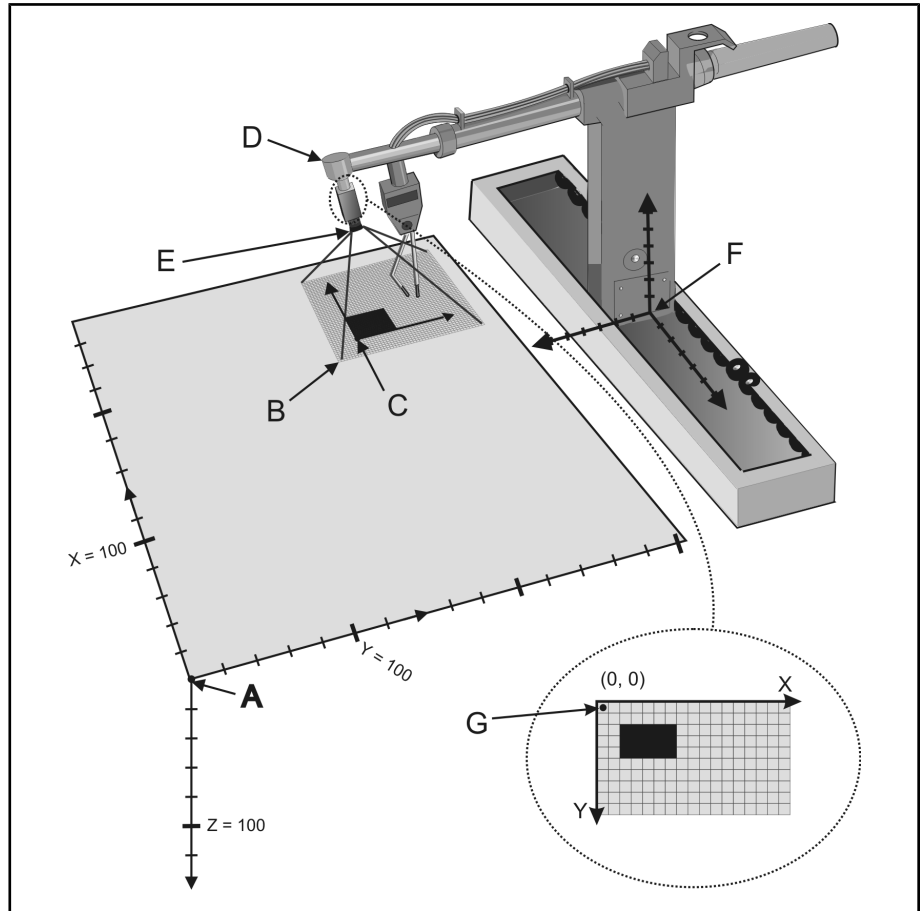
The basic concepts and vocabulary conventions for the Camera Calibration module are:

- **Axis.** The axes of a coordinate system are used to specify the location of objects within a coordinate system. A coordinate system typically has as many axes as there are dimensions in the space it describes; these axes are typically perpendicular to each other.
- **Calibration plane.** Calibration plane is the world plane used to perform the initial calibration with a grid or a list of points.
- **Calibration points.** Calibration points are points in an image with known real-world coordinates and used to establish the real-world coordinates of all other points in the image based on the selected calibration mode.
- **Coordinates.** Coordinates are used to define an object's location in space. Coordinates are typically expressed as a set of numbers. Each number in the set represents the object's distance from the origin along a corresponding axis.

- **Coordinate system.** Coordinate systems are used to position objects. Coordinate systems are characterized by their origin and their axes.
- **Distortion.** Distortion is an optical effect which causes objects in an image to appear deformed.
- **Field of view.** Field-of-view refers to the largest world region visible in an image at a given resolution. It is the extent of the observable world that is seen at any moment.
- **Origin.** The origin of a coordinate system is its reference point. At the origin, the position along all the axes of the coordinate system is zero; positions in the coordinate system are typically returned with respect to the origin.
- **Pixel units.** Pixel units are the most basic unit in an image. Pixel units are used to measure sizes, distances, and positions in an uncalibrated image.
- **Pose.** A pose is the position and orientation of an object in space.
- **Real-world units.** Real-world units are used to measure sizes, distances, and positions as they exist in the real world.
- **Robot encoders.** Robot encoders are the parts of a robot that measure the movement of the robot's arm and return it in human-readable units.
- **Transformation.** Transformations are used to move, rotate, or scale objects within a coordinate system. Typically, these transformations are represented using matrices. Transformations can also be used to convert measurements from pixel units to real-world units or vice versa.
- **Working area.** Working area is the area of the image from which you want real-world results.

Coordinate systems

MIL supports different coordinate systems to describe positions in your camera setup. The following diagram shows these coordinate systems using a sample camera setup. Notice that the tool's position is assigned to the arm of the robotic manipulator.



Legend:

Marker	Description	Coordinates (as measured in the absolute coordinate system)
A	Origin of the absolute coordinate system.	(0, 0, 0)
B	Camera field of view from which image is captured	(180, 130, 0)
C	Origin of the relative coordinate system	(190, 145, 0)
D	Origin of the tool coordinate system	(190, 155, -75)
E	Origin of the camera coordinate system	(190, 155, -60)
F	Origin of robot-base coordinate system	(160, 360, -20)
G	Image coordinate system	-

Absolute world coordinate system

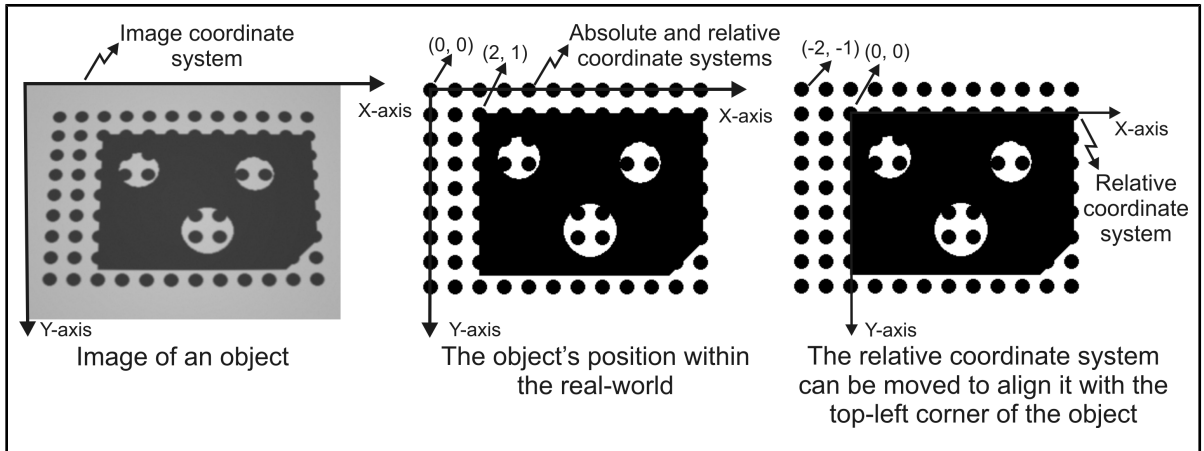
The absolute world coordinate system is implicitly defined from the calibration points when calibrating the camera setup. The absolute world coordinate system is unmovable. Its unit of measure is user-defined (for example, mm, cm, or inches). Calibration relates the image coordinate system to the absolute world coordinate system.

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

Relative world coordinate system

The relative world coordinate system is the coordinate system used to locate and/or measure objects in the real world. By default, after calibration, positional results will be returned within the relative world coordinate system, rather than the image coordinate system. Initially, the relative world coordinate system has the same position and orientation as 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.

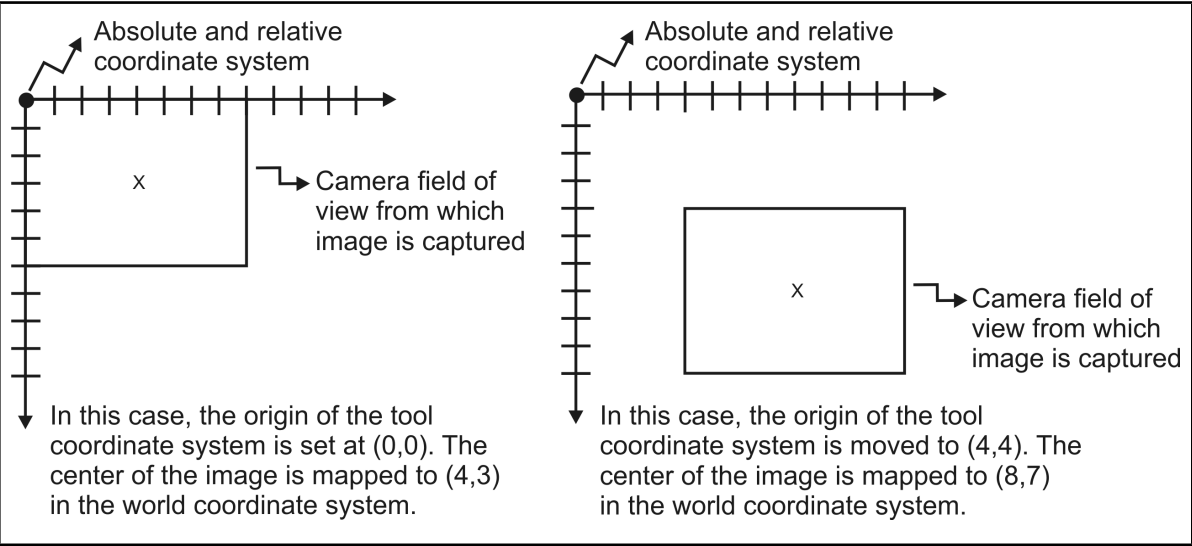


Note that this image is for illustrative purposes only. In general, the object should not be placed over a grid because if the grid's circles and the object are not differentiated when performing the processing operation, erroneous results will be returned. Refer to the *Calibrating your camera setup* section in *Chapter 5: Camera calibration* for more information on calibrating with a grid.

Tool coordinate system

The tool coordinate system relates to the tool that holds the camera in your camera setup. For example, in a robotic arm application, the origin of the tool coordinate system would be the end point of the tool's arm holding the camera. By default, the tool coordinate system has the same position and orientation as the absolute coordinate system.

The tool position can be any arbitrary point that moves with the camera. Moving the origin of the tool coordinate system is an indirect means of positioning the camera coordinate system. Since moving the camera coordinate system implies a different field of view, moving the origin of the tool coordinate system affects positional results taken from a calibrated image. The world coordinates of an image pixel taken before and after moving the tool coordinate system are not the same.



Relative camera position

Adjusting the tool position can be useful when analyzing an object that cannot fit in a single image. For more information, see the *Calibrating a camera setup that analyzes large objects* section in *Chapter 5: Camera calibration*.

Camera coordinate system

The camera coordinate system is the coordinate system with an origin positioned at the center of the camera's lens and is only useful for 3D calibration modes. The 2D calibration modes do not estimate the position of the camera coordinate system; the movement of the camera coordinate system is accomplished by moving the tool coordinate system. The 3D calibration modes estimate the orientation and distance between the camera and the calibration plane as part of their algorithmic calculations; so when using this mode, the camera position is the position of the modeled pinhole camera calculated using Tsai-based algorithm.

Moving the camera coordinate system is usually performed by moving the tool coordinate system. See the *Changing your camera setup* section in *Chapter 5: Camera calibration* for more information on moving your camera. Moving the origin of the camera coordinate system affects positional results taken from a calibrated image. The world coordinates of an image pixel taken before and after moving the camera coordinate system are not the same.

Robot-base coordinate system

The robot-base coordinate system is the coordinate system with an origin positioned at the base of the robot holding the camera. It is used by the robot software to allow setting the tool coordinate system with respect to the robot's base and can be used to provide the calibration module with values returned by the robot encoder. This coordinate system is only available for robotics calibration modes.

The robot base coordinate system simplifies defining the tool coordinate system. Even if its position and orientation with respect to the absolute coordinate system is originally unknown, the tool coordinate system can still be defined with respect to the robot base coordinate system. After performing full calibration, the relation between the robot base coordinate system and absolute coordinate system is established, as well as the relation between tool and absolute coordinate systems.

Usually, you cannot move the robot-base coordinate system, unless the robotic encoders are set on a mobile robot base.

Image coordinate system

The image coordinate system is the coordinate system used to locate and/or measure objects in a non-calibrated 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.

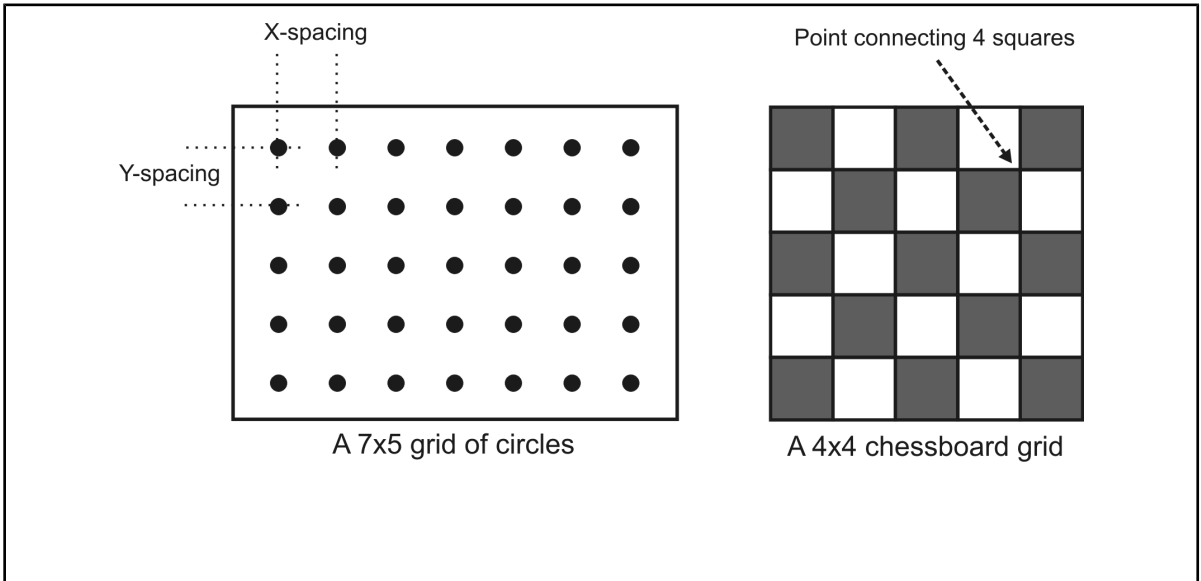
Calibrating your camera setup

To calibrate your camera setup, you must specify points in the image and map them to real-world coordinates. These are known as calibration points. You can use an image of a user-defined grid of circles or a grid of squares and have MIL automatically extract the calibration points. Alternatively, you can explicitly specify a list of pixels and their real-world coordinates. To use a grid, call **McalGrid()**. To use a list of coordinates, call **McalList()**.

Real-world grid

McalGrid() determines the calibration points from an image of a user-defined grid and the world description of this grid. The world description includes the number of rows and columns, as well as the point-to-point distance between these rows and columns, in real-world units. The number of rows and columns in your grid are used to specify the number of calibration points that **McalGrid()** extracts.

The Camera Calibration module supports two types of grids: grids of circles and chessboard grids (that is, grids of squares). In a grid of circles, the center of each circle in the real-world grid is a grid point. Whereas in a chessboard grid, points connecting four squares are grid points. Grid points are mapped to pixels in the grid's image to establish calibration points. Regardless of its type, your grid should contain at least 4 grid points for 2D-based calibration objects and 6 points for 3D-based calibration objects.



- ❖ You can speed up the time it takes to create a subpixel mapping for calibration points by using `McalGrid()` with `M_FAST` combination constant. In this case, the accuracy of the calibration might decrease.

General rules for constructing a grid

Image of the grid

You can create a pixel-to-world mapping from many grids using `McalGrid()` with `M_CHESSBOARD_GRID` or `M_CIRCLE_GRID`. However, to create an accurate (subpixel) mapping, the image of your grid should meet a set of guidelines (at the working resolution), depending on the type of your grid.

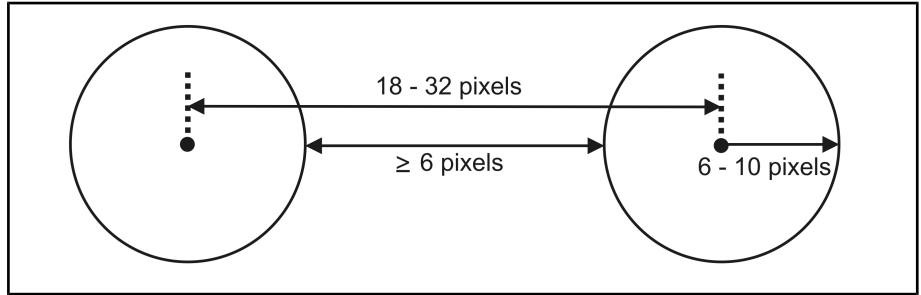
General rules for constructing a grid

Grid of circles

The image of your grid of circles should meet the following guidelines:

- 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.

General rules for constructing a grid

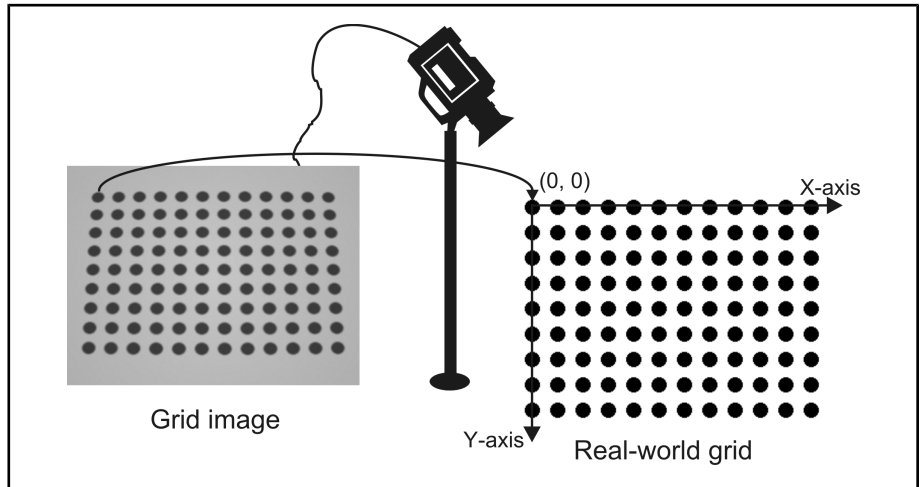
Chessboard grid

Any image of a chessboard grid can create a correct pixel-to-world mapping using `McalGrid()`, as long as it comprises the minimum-number of grid points mentioned previously. You should shed enough light on the grid to obtain high contrast in its image.

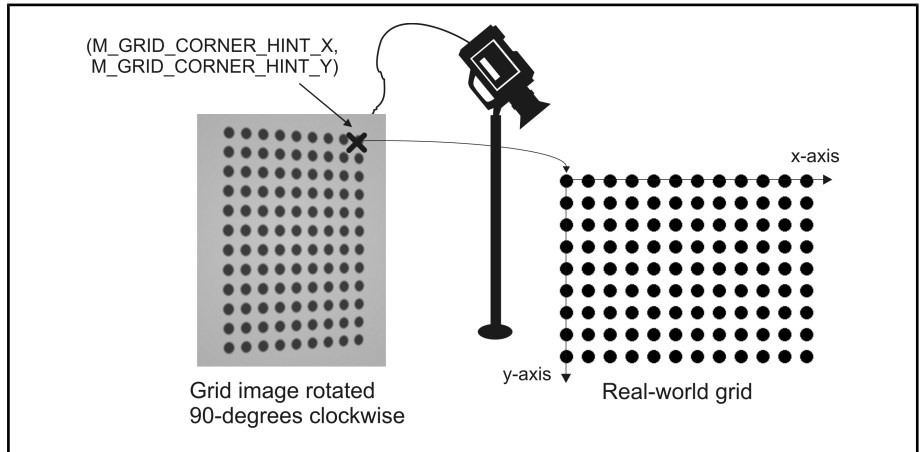
The world

Specifying the top-left corner of your grid

By default, the circle in the top-left corner of the grid image is associated to the origin, $(0, 0)$, of the absolute 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).



If the image of the calibration grid is rotated, such that the top-left circle of the grid in the real-world is not located at the top-left corner of the image, you can specify the approximate image coordinates of the circle that corresponds to the top-left circle of the grid in the real-world, using **McalControl()** with **M_GRID_CORNER_HINT_X** and **M_GRID_CORNER_HINT_Y**. In this case, when you call **McalGrid()**, MIL will use the corner circle closest to the specified coordinates, as the top-left corner of the grid.

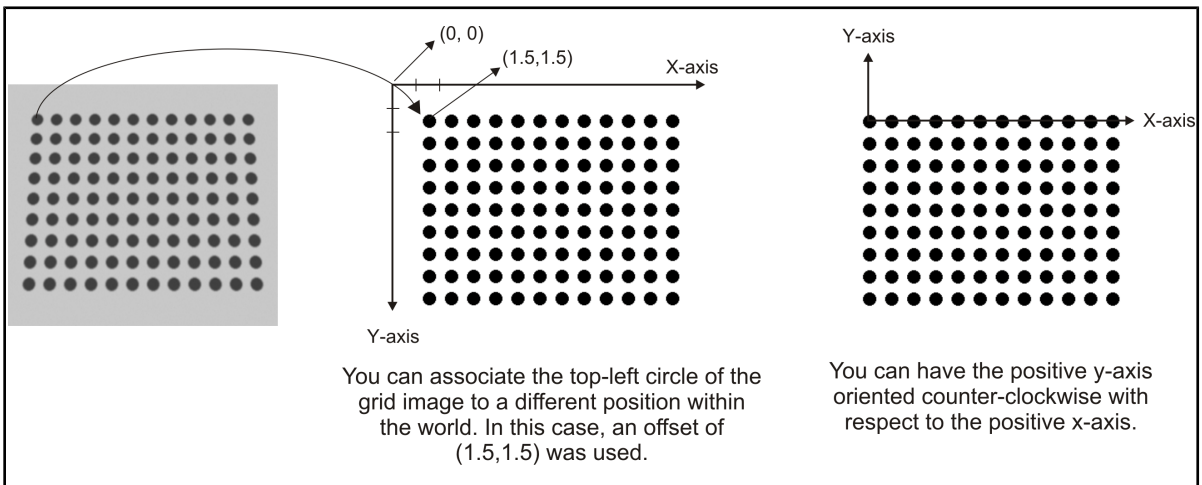


Offset and Y-axis

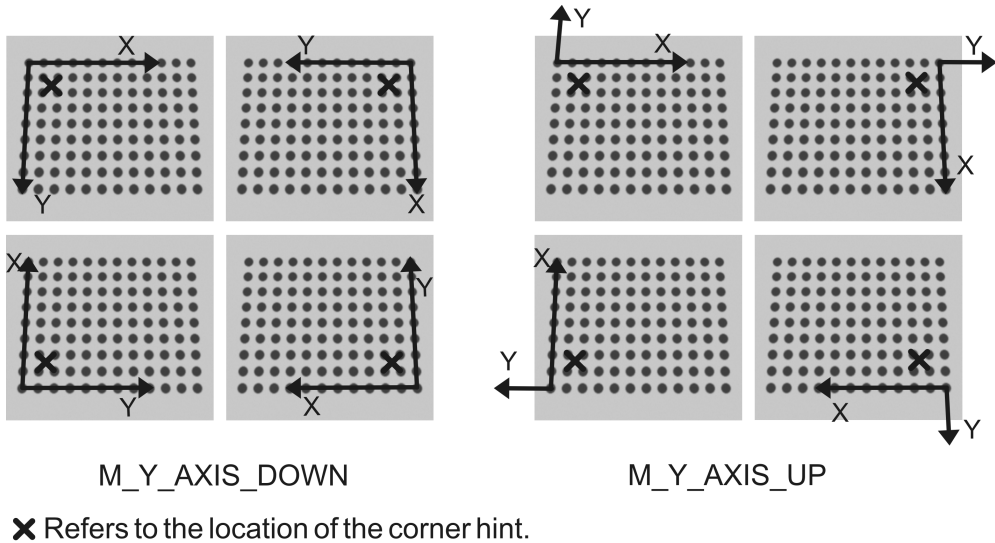
You can use **McalGrid()** to associate the top-left circle of the grid image to a different position within the absolute coordinate system. This offset is specified in real-world units. The origin of the absolute coordinate system does not have to be within the field-of-view. If the top-left circle corresponds to the location at

which you want to position the relative coordinate system, you can specify the world description of the grid in the relative coordinate system by first calling **McalControl()** with **M_CALIBRATION_PLANE** set to **M_RELATIVE_COORDINATE_SYSTEM**, then calling **McalGrid()**.

By default, the Camera Calibration module assumes the Y-axis is oriented 90° clockwise with respect to the positive X-axis. You can have the positive Y-axis oriented 90° counter-clockwise with respect to the positive X-axis using **McalGrid()** with **M_CIRCLE_GRID+M_Y_AXIS_UP**.



Note that the orientation of the absolute coordinate system in the image of the grid is affected by the location of the corner hint specified using **McalControl()** with **M_GRID_CORNER_HINT_X** and **M_GRID_CORNER_HINT_Y**.



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**, **M_CALIBRATION_IMAGE_POINTS_Y**, **M_CALIBRATION_WORLD_POINTS_X**, and **M_CALIBRATION_WORLD_POINTS_Y**. The coordinates correspond to the center of the circles.

Note that a call to **McalGrid()** also associates the calibration object to the source image buffer.

List of coordinates

McalList() allows you to explicitly specify the calibration points as a list of pixel coordinates and their associated real-world coordinates. 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 radial distortion, the Camera Calibration module requires a larger number of coordinates (for example, more than 30) distributed over the image.

Calibration modes

When you use `McalAlloc()` to create the calibration object, you have to specify the calibration mode. MIL supports the following calibration modes:

- Piecewise linear interpolation mode.
- Perspective transformation mode.
- Tsai-based mode.
- Robotics mode.

The following table provides a side-by-side feature comparison of the different calibration modes supported by MIL:

Feature	Linear interpolation	Perspective transformation	Tsai-based	Robotics
Supports camera movement parallel to the image plane	X	X	X	X
Supports full 3-dimensional camera movement			X	X
Supports working with coordinates that lie outside the calibration grid	(less accurate)	X	X	X
Calculates the position of the robot base				X
Calibrates with a single image only	X	X	X	
Allows calibrating with the camera perpendicular to the calibration plane (0° angle of incidence)	X	X		X
Supports one-dimensional distortions obtained from line-scan cameras	X	X		
Corrects rotation, scale, and perspective distortion	X	X	X	X
Corrects lens distortion	X		X	X
Corrects all types of distortion	X			
Corrects lens distortion only and keeps other distortions uncorrected			X	X
Models the camera explicitly			X	X
Estimates the position of the camera and other known objects using known world points			X	X
Allows retrieval of the position and orientation of the robot's base coordinate system				X
Allows proper calibration of cameras used for 3D reconstruction by <code>M3dmapCalibrate()</code> and <code>M3dmapTriangulate()</code> functions			X	X

Piecewise linear interpolation mode

Piecewise linear interpolation mode can compensate for any kind of distortion. In general, you should use this mode. 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 specified set of images coordinates and their real-world equivalents. It performs a mapping of each point in the image to a point in the real world using bilinear interpolation between adjacent points.

To allocate a piecewise linear interpolation calibration object, use **McalAlloc()** with **M_LINEAR_INTERPOLATION**.

- ❖ Note that piecewise linear interpolation mode does not support changing the distance between the camera and the calibration plane. Re-calibration is required in this case.

Perspective transformation mode

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. It finds a projection of an image in a field of view into a 2D plane.

To allocate a perspective transformation calibration object, use **McalAlloc()** with **M_PERSPECTIVE_TRANSFORMATION**.

- ❖ Note that perspective transformation mode does not support changing the distance between the camera and the calibration plane. Re-calibration is required in this case.

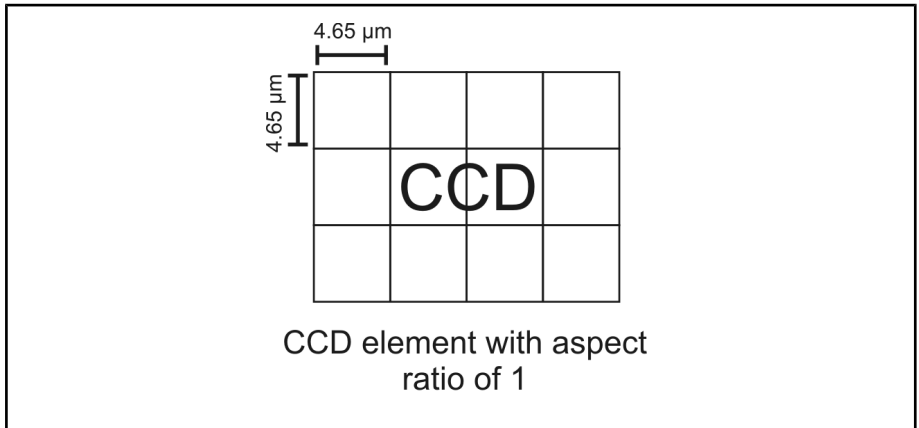
Tsai-based mode

Tsai-based mode is a calibration mode based on Roger Y. Tsai's algorithm. This mode models your camera as a pinhole camera and can compensate for rotation, translation, scale, perspective, and radial lens distortions. Tsai-based mode is a true 3D calibration technique which actually determines the distance and orientation between the modeled camera and the image plane. This allows for results to be inquired with respect to any defined plane. This mode allows for camera rotation about any axis without re-calibrating; you only need to specify the camera movement which has occurred. The mode requires a minimum of 6 points to perform a successful calibration and accuracy increases with the number of points. To allocate a Tsai-based calibration object, use **McalAlloc()** with **M_TSAI_BASED**.

Tsai-based mode calculates several intrinsic and extrinsic attributes of the pinhole camera that is used to model the real-world camera. Intrinsic attributes characterize properties of the camera itself (such as the focal length and the radial distortion), whereas extrinsic attributes characterize the camera's position and orientation.

Tsai-based mode does not estimate the following two intrinsic attributes that are needed for accurate calibration; you can adjust their values set by default if not correct:

- **Aspect ratio of the camera's charge-coupled device (CCD) elements.** This refers to the ratio of the horizontal dimension of a single CCD element to its vertical dimension. Tsai-based mode sets this ratio to 1 by default. If the aspect ratio given by the camera manufacturer is not 1, specify its actual aspect ratio using **McalControl()** with **M_CCD_ASPECT_RATIO**.



- **Principal point.** This refers to the point of intersection of the camera's optical axis with the image plane. You can specify the principal point using **McalControl()** with **M_PRINCIPAL_POINT_X** and **M_PRINCIPAL_POINT_Y**, prior to calibrating. If **McalGrid()** is used for calibration, the principal point will default to the central point of the image plane. If **McalList()** is used for calibration, it is mandatory to set the principal point prior to calibrating. In this case, it is recommended to set the principal point to the central pixel of the captured image.

To successfully perform a full calibration using Tsai-based mode, you must ensure that the optical axis of your camera is at least 30° away from the axis perpendicular to the calibration plane. This angle provides the image perspective needed by Tsai-based mode to estimate some intrinsic and extrinsic attributes. For camera setups that require a camera whose optical axis is perpendicular to the calibration plane, you must perform two calibrations: the first calibration must be performed with the camera inclined with respect to the calibration plane, using **McalGrid()** or **McalList()** with **M_FULL_CALIBRATION**; the second calibration must be

performed after repositioning the camera, such that its optical axis is perpendicular to the calibration plane, using **McalGrid()** or **McalList()** with

M_DISPLACE_CAMERA_COORD. Instead of calibrating a second time, you can, alternatively, move the camera coordinate system using **McalSetCoordinateSystem()** if the movement is known with great precision. If you want to move the camera coordinate system by moving the tool coordinate system, you must first move the tool coordinate system as described in the *Special considerations concerning the tool and camera coordinate systems* subsection in the *Changing your camera setup* section in *Chapter 5: Camera calibration*.

Robotics mode

Robotics mode is a true 3D calibration technique that supports the robot base coordinate system. In addition to estimating camera's intrinsic and extrinsic attributes, this mode also estimates the position and orientation of the robot base coordinate system with respect to the absolute coordinate system. Like the Tsai-based mode, robotics mode models the camera as an ideal pinhole camera. Therefore, the robotics mode estimates the same camera attributes estimated by Tsai-based mode. It also allows for results to be inquired with respect to any defined plane.

To allocate a calibration object in robotics mode, use **McalAlloc()** with **M_3D_ROBOTICS**. Robotics mode requires at least three calls to **McalGrid()** or **McalList()** with **M_ACCUMULATE** before performing a full calibration with **M_FULL_CALIBRATION**. Before each call, you must move the camera to a different position, and set the position of the tool coordinate system with respect to the robot base coordinate system using **McalSetCoordinateSystem()**.

Inquiring about your calibration

After calibration, you can inquire about the status and accuracy of your calibration, the intrinsic and extrinsic attributes of the pinhole camera used to model your camera when performing **M_TSAI_BASED** calibration, and other calibration information.

Calibration status and errors

After calibrating with **McalGrid()** or **McalList()**, you can check the success of your calibration using **McalInquire()** with **M_CALIBRATION_STATUS**. If calibration was successful, it will return **M_CALIBRATED**, otherwise an error code is returned. For example, if you calibrate using **McalGrid()** and the grid does not meet the required guidelines, **M_CALIBRATION_STATUS** returns **M_GRID_NOT_FOUND**. If you try to perform a full calibration in **M_TSAI_BASED** mode, and the camera is placed perpendicular to the grid, **M_PLANE_ANGLE_TOO_SMALL** is returned.

M_NOT_INITIALIZED is returned if **M_CALIBRATION_STATUS** is inquired before calling **McalGrid()** or **McalList()**.

Calibration accuracy

After a successful calibration, you can retrieve the accuracy of your calibration by inquiring the maximum or the average value of the calibration error. The calibration error is defined as the distance, in pixel units, between the initial calibration points and their projected points in an image. To retrieve the calibration error in pixel units, use **McalInquire()** with **M_AVERAGE_PIXEL_ERROR** or **M_MAXIMUM_PIXEL_ERROR**. Whereas, to retrieve the calibration error in world units, use **McalInquire()** with **M_AVERAGE_WORLD_ERROR** or **M_MAXIMUM_WORLD_ERROR**.

For robotics calibration mode, you can use **McalInquireSingle()** to inquire about the calibration accuracy of each call to **McalGrid()** or **McalList()** made with **M_ACCUMULATE**.

Position and orientation of coordinate systems

You might need to inquire about the position and orientation of one coordinate system with respect to another. You can do so using **McalInquire()** for two-dimensional calibration modes or **McalGetCoordinateSystem()** for three-dimensional calibration modes.

For two dimensional calibration modes, you can inquire about the position and orientation of the relative coordinate system and the position of the tool coordinate system:

- To retrieve the position of the relative coordinate system, use **McalInquire()** with **M_RELATIVE_ORIGIN_X**, **M_RELATIVE_ORIGIN_Y**, and **M_RELATIVE_ORIGIN_Z**. To inquire about the rotation of the relative coordinate system, use **McalInquire()** with **M_RELATIVE_ORIGIN_ANGLE**; this will return the counter-clockwise angle of rotation about the Z-axis as measured with respect to the absolute coordinate system.
- To retrieve the position of the tool coordinate system, use **McalInquire()** with **M_TOOL_POSITION_X**, **M_TOOL_POSITION_Y**, and **M_TOOL_POSITION_Z**.

For 3D-based calibration modes, use **McalGetCoordinateSystem()** to inquire about the position and orientation of the absolute, relative, tool, or camera coordinate systems. For robotics calibration mode, you can also inquire about the position and orientation of the robot base coordinate system.

McalGetCoordinateSystem() can return the relationship between these coordinate systems in three-dimensional space, in terms of a specified type of transformation. For example, after a successful Tsai-based calibration, you might need to know the calculated position and orientation of your camera with respect to the absolute coordinate system. To obtain this information as a translation and rotation of the camera coordinate system from the absolute coordinate system, you can use **McalGetCoordinateSystem()** with **M_CAMERA_COORDINATE_SYSTEM** as the target coordinate system and **M_ABSOLUTE_COORDINATE_SYSTEM** as the reference coordinate system. The following code snippet shows how to retrieve this information using two consecutive calls to **McalGetCoordinateSystem()**. The

first call returns the X, Y, and Z coordinates of the camera coordinate system's origin, and the second call returns the three rotation angles of the camera coordinate system's axes around the X, Y, and Z axes of the absolute coordinate system.

```

/*
    Get the position and orientation of the camera
    with respect to the absolute coordinate system.
*/
MIL_DOUBLE CameraPosX, CameraPosY, CameraPosZ;
MIL_DOUBLE AngleY, AngleX, AngleZ;

McalGetCoordinateSystem(CalibrationID,
                        M_CAMERA_COORDINATE_SYSTEM,
                        M_ABSOLUTE_COORDINATE_SYSTEM,
                        M_TRANSLATION,
                        M_NULL, &CameraPosX, &CameraPosY, &CameraPosZ, M_NULL);

McalGetCoordinateSystem(CalibrationID,
                        M_CAMERA_COORDINATE_SYSTEM,
                        M_ABSOLUTE_COORDINATE_SYSTEM,
                        M_ROTATION_YXZ,
                        M_NULL, &AngleY, &AngleX, &AngleZ, M_NULL);

```

For robotics calibration mode, you can retrieve the position and orientation of any coordinate system with respect to the robot base coordinate system using **McalGetCoordinateSystem()**. The values returned can then be provided to a robot controller software to move the robot.

- ❖ Note that the image coordinate system cannot be used with **McalGetCoordinateSystem()**. Refer to the *Getting results in realworld units* section in *Chapter 5: Camera calibration* for information on transforming values from world to pixel units.

Refer to the *Changing your camera setup* section in *Chapter 5: Camera calibration* for more information on coordinate system transformations.

Pinhole camera

You can retrieve the values of several intrinsic and extrinsic attributes of the pinhole camera used to model your camera when working in Tsai-based or robotics calibration mode:

- To retrieve the calculated effective focal length of the modeled pinhole camera, use **McalInquire()** with **M_FOCAL_LENGTH**. The returned value is expressed in horizontal pixel units. You can use the following relation to convert the focal length from pixel units into real-world units:

$$f_{pixel} = \frac{f}{\Delta x}$$

Where Δx specifies the distance between the centers of two horizontally adjacent CCD elements of your camera. Note that this is the focal length of the modeled pinhole camera; it will not necessarily match the focal length given by the lens manufacturer.

- To retrieve the second order radial distortion coefficient, use **McalInquire()** with **M_DISTORTION_RADIAL_1**. Radial distortion refers to image distortions caused by the camera's lens, namely, pincushion and barrel distortions. It is calculated by measuring the distance from the principal point to any other point in the image plane. Radial distortion is modeled by the following equations:

$$\delta x = x * (\kappa_1 r^2)$$

$$\delta y = y * (\kappa_1 r^2)$$

Where δx and δy represent the distortion in the X- and Y-directions, respectively, κ_1 represents the second order radial distortion coefficient, and r represents the distance from the principal point.

- To locate the camera's position in space, use **McalGetCoordinateSystem()** with **M_CAMERA_COORDINATE_SYSTEM** as a target coordinate system.

Getting results in real-world units

Transforming
coordinates

Transforming coordinates or results

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 for better efficiency.

Transforming
results

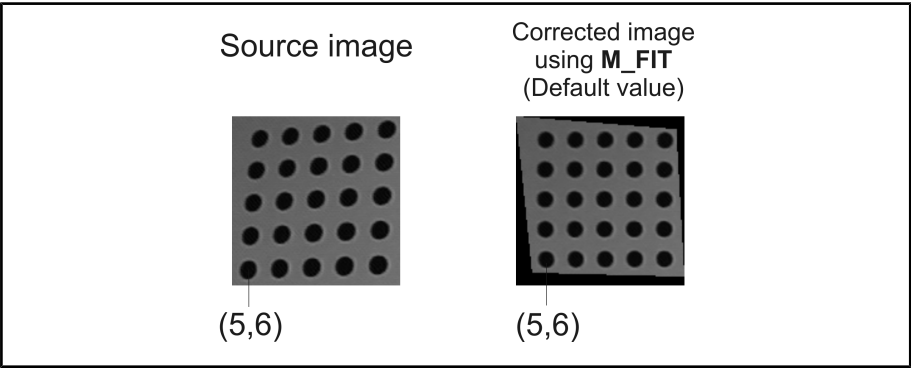
You can use `McalTransformResult()` to convert a specific non-positional result (a length, angle, or area) from its pixel to its world value (or vice-versa). Note, however, that this function uses the average pixel size to perform the conversion; results will be more accurate if you first correct the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

Corrected image

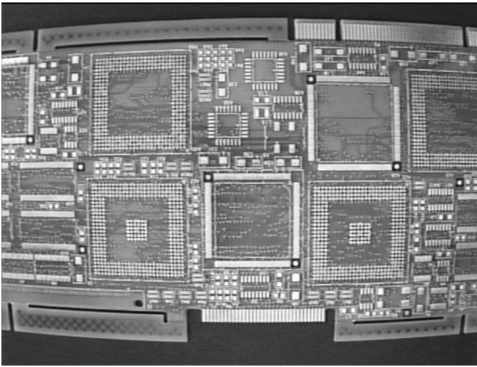
Physically correcting an image

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 Camera 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.

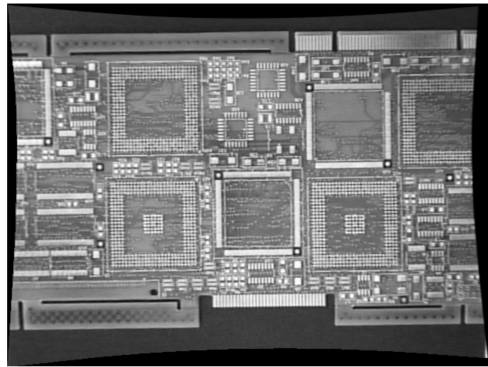
For example, after calibrating the source image below, the world coordinates of the bottom-left circle are (5,6). When you correct your image, the world coordinates of the bottom-left circle in the corrected image are also (5,6), even though it is evident that the pixel coordinates are different for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



When working in Tsai-based calibration mode or robotics calibration mode, it is possible to physically correct only lens distortions of an image, that is, barrel distortion and pin cushion distortion, without removing other types of distortions. To correct these two distortions only, use **McalTransformImage()** with **M_CORRECT_LENS_DISTORTION_ONLY**.



Barrel distortion caused by a camera lens



Distortion corrected using
M_CORRECT_LENS_DISTORTION_ONLY

Note that images are physically corrected using a geometric warping.

Accelerating through a cache

Generating look up tables to correct images

Instead of generating a corrected image, it is possible to generate look up tables (LUTs) using **McalTransformImage()** with **M_EXTRACT_LUT_X** and **M_EXTRACT_LUT_Y**. Then, you can use these LUTs to correct images using **MimWarp()**. It is recommended to use this method in one of the following situations:

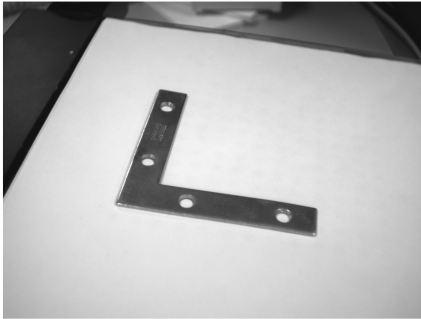
- You want to correct images using Image Processing module only.
- You are using hardware that yields faster processing with LUT inputs.

Scale and position of corrected image

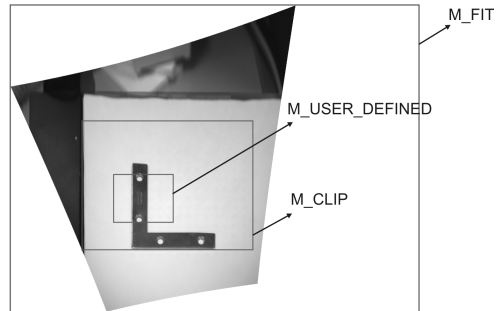
Each pixel in the physically corrected image has a constant size in real-world units. You can specify the size of pixels and the position of the origin of the relative coordinate system in the physically corrected image using **McalControl()** with **M_TRANSFORM_IMAGE_FILL_MODE** set to **M_FIT**, **M_CLIP**, or **M_USER_DEFINED** before using **McalTransformImage()**.

Using **M_FIT** specifies that the size and position are calculated automatically by fitting all the image in the physically corrected image, while **M_CLIP** automatically maps every pixel in the corrected image to a valid source pixel, removing any overscan region. Alternatively, you can manually select the size and position of the pixels in the physically corrected image by setting **M_TRANSFORM_IMAGE_FILL_MODE** to **M_USER_DEFINED** and then setting

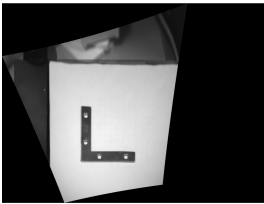
the values of `M_TRANSFORM_IMAGE_WORLD_POS_X`,
`M_TRANSFORM_IMAGE_WORLD_POS_Y`,
`M_TRANSFORM_IMAGE_PIXEL_SIZE_X`, and
`M_TRANSFORM_IMAGE_PIXEL_SIZE_Y`.



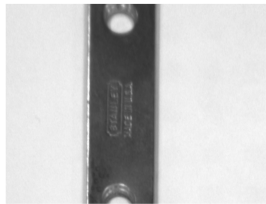
Original image



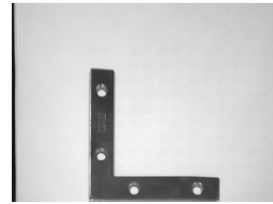
Three different fill modes, of the corrected image, overlaid.



`M_FIT`



`M_USER_DEFINED`



`M_CLIP`

Accelerating
through a cache

Transformation 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 **McalControl()** with **M_TRANSFORM_CACHE**. The cache consists of two 32-bit buffers that have the same size as the destination buffer of **McalTransformImage()**.

The information in the cache is flushed whenever you change the size of the source or destination buffers of **McalTransformImage()**, the angle of the relative coordinate system in the calibration object, or the size and position of pixels in the corrected image.

Associating

Automatically getting results in real-world units

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, use **McalAssociate()** with **M_NULL**.

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. For information on the implications this has on processing, see the *Processing calibrated images* section in *Chapter 5: Camera calibration*.

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

Calibrated digitizer

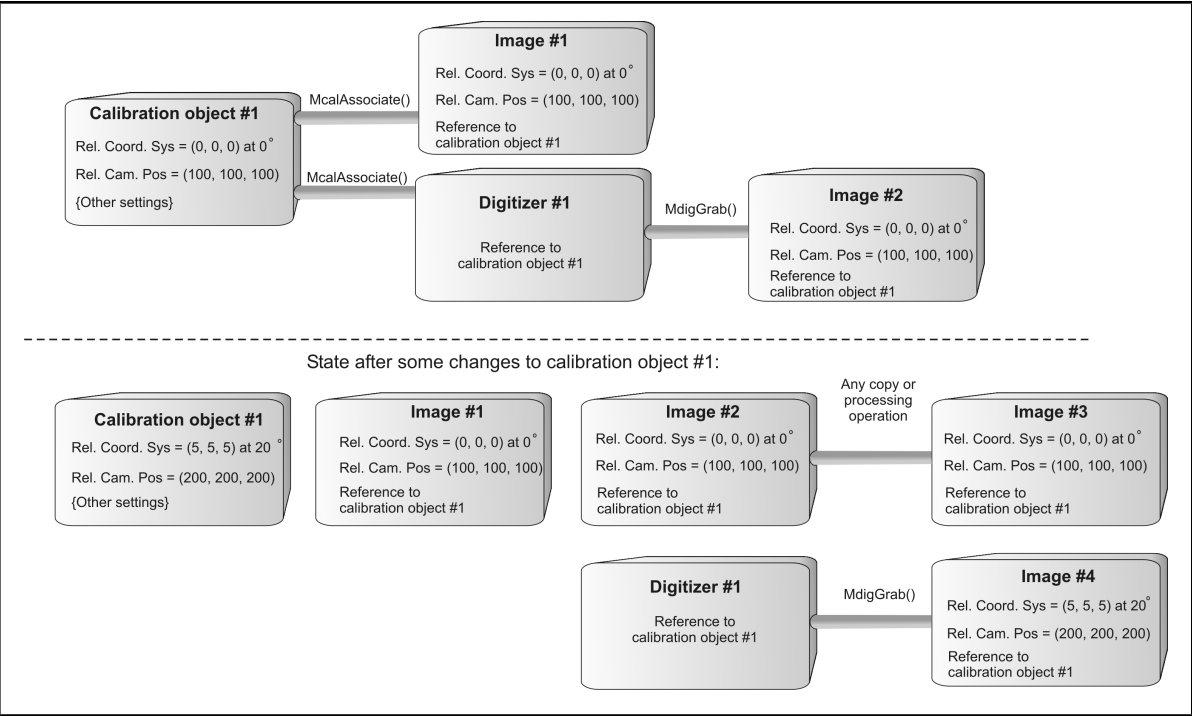
A digitizer with an associated calibration object is known as a calibrated digitizer. When you grab an image with a calibrated digitizer, the calibration object currently associated to the digitizer gets associated to the grabbed image. Therefore, the grabbed image becomes a calibrated image.

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 a calibration object to image versus 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 position and orientation of all coordinate systems saved within the calibration object and a reference to the calibration object for all other settings including the intrinsic attributes. This means that, if you change the relative coordinate system

or camera position after association, the change will not affect the image. 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 with 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.

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 `McalControl()` with `M_OUTPUT_UNITS`. By default, results are in real-world 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 function description. While results can be returned in pixel or real-world units, in general, any value which you pass to a MIL function must be in pixel units. If a value must be passed in world-units when working with a calibrated image, it will be stated in the function's description.

Changing your camera setup

Once your camera setup is calibrated, you might need to manipulate your calibration coordinate systems for different reasons. For example, you might need to reposition the tool and the camera coordinate system because the tool holding your camera might have moved, or you might need to reposition the relative coordinate system to measure, in real-world units, an object located higher than the calibration plane. To do so, use **McalSetCoordinateSystem()** for 3D-based calibration objects (**M_TSAI_BASED** or **M_3D_ROBOTICS**), or **McalRelativeOrigin()** and **McalControl()** for other calibration objects. Using these functions, you can update your camera setup without re-calibration. Refer to the *Calibrating a camera setup that analyzes large objects* section in *Chapter 5: Camera calibration* for some useful applications of these functions.

- ❖ Note that you cannot move the coordinate systems of a calibrated image. If needed, you must move the coordinate systems of your calibration object prior to associating the calibration object with an image. This allows the change to take effect in the results obtained from the calibrated image.

Types of transformation

MIL can reposition a coordinate system (target) with respect to another coordinate system (reference), using two main types of transformations: translation and rotation.

Translation

A translation of a coordinate system is equivalent to the displacement of all points along the reference coordinate axes and is specified using a translation vector. For calibration objects using a perspective-transformation mode or linear-interpolation mode (two-dimensional calibration modes), you can move the relative coordinate system using **McalRelativeOrigin()**. For 3D-based calibration objects, you can apply a translation along the X, Y, and Z-axes using **McalSetCoordinateSystem()** with **M_TRANSLATION**.

Rotation

A rotation of a coordinate system is equivalent to the movement of all points in a circular motion around some axis in a reference coordinate system.

To rotate the relative coordinate system of a calibration object that uses a two-dimensional calibration mode, use **McalRelativeOrigin()** with the **AngularOffset** parameter set to the rotation angle, in degrees.

To rotate a coordinate system of a 3D-based calibration object, use **McalSetCoordinateSystem()**. Rotations in three-dimensional space can be specified in multiple ways using this function:

- **Axis and angle rotation.** Expresses the rotation of the target coordinate system as a rotation of all its points by a certain angle of rotation (θ), around an arbitrary axis of rotation defined by a unit vector in the reference coordinate system.

$$(axis, angle) = \left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}, \theta \right)$$

A pair describing an axis and angle of rotation by θ degrees

- **Quaternion rotation.** Expresses the rotation of the target coordinate system as a quaternion from the angle of rotation (θ) and the normalized axis of rotation (\vec{v}), as defined in the following equation.

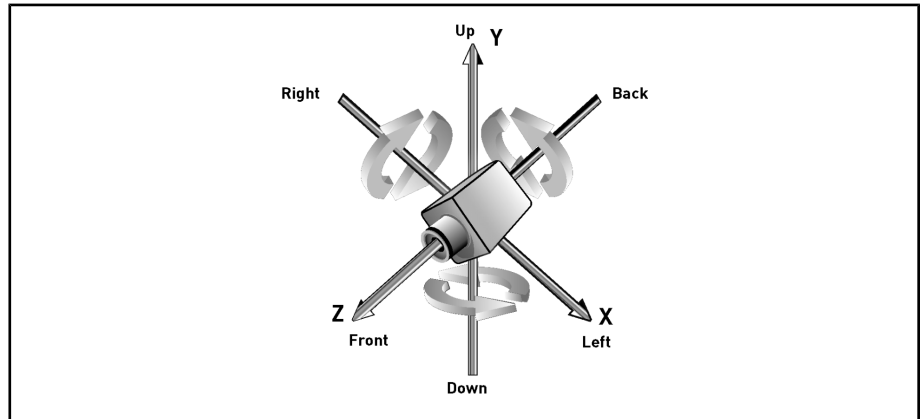
$$\vec{q} = \langle \cos(\frac{\theta}{2}), \sin(\frac{\theta}{2})v_x, \sin(\frac{\theta}{2})v_y, \sin(\frac{\theta}{2})v_z \rangle$$

- **Matrix rotation.** Expresses the rotation of the target coordinate system as a 3x3 square matrix transformation.

$$R = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

A 3x3 matrix for θ -degrees rotation around the Z-axis.

- **X-Y-Z rotation.** Expresses the rotation of the target coordinate system as a rotation of all points along the three axes of the reference coordinate system, with all possible X-, Y- and Z-component combinations. It is also known as Roll-Pitch-Yaw rotation.



- **Homogeneous matrix rotation.** Expresses the rotation of the target coordinate system as a homogeneous matrix transformation. Homogeneous matrices can express both a translation and a rotation of a coordinate system. It is a 4x4 matrix made up of a 3x3 rotation matrix and a translation vector. If the rotation matrix is an identity matrix, only a translation takes effect.

$$Q = \begin{bmatrix} \begin{bmatrix} & & \\ & R & \\ & & \end{bmatrix} & \begin{bmatrix} \\ \vec{t} \\ \end{bmatrix} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

*A 4x4 homogeneous matrix
made of a 3x3 rotation matrix (R)
and a translation vector (\vec{t}).*

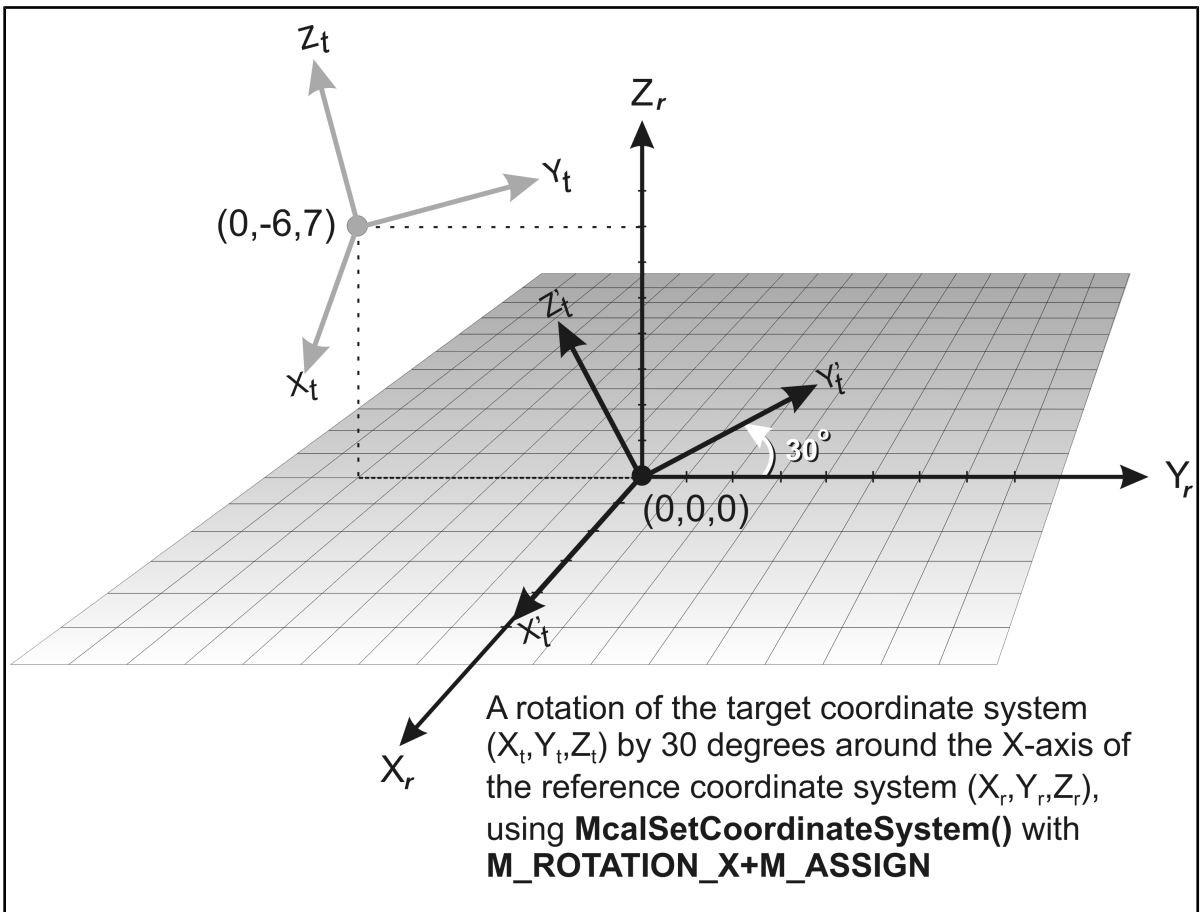
Transforming coordinate systems

For two-dimensional calibration modes, you can only transform the relative and tool coordinate systems.

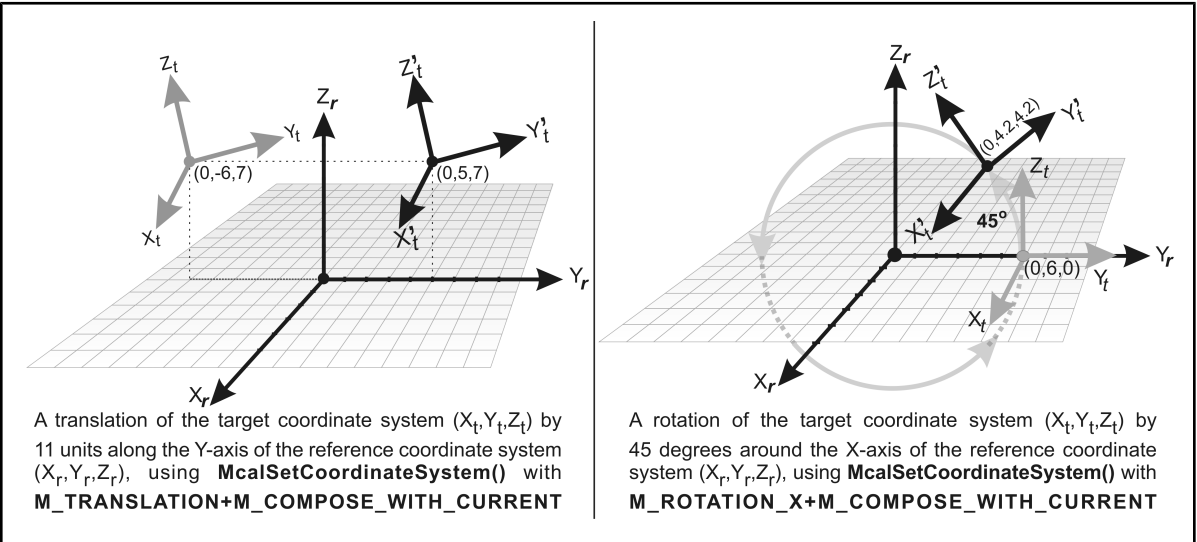
- To define, move, or rotate the relative coordinate system, use **McalRelativeOrigin()**.
- To specify a new position for the tool coordinate system, you can use **McalControl()** with **M_TOOL_POSITION_X**, **M_TOOL_POSITION_Y**, and **M_TOOL_POSITION_Z**.

For three-dimensional calibration modes, use **McalSetCoordinateSystem()** to transform a three-dimensional coordinate system in terms of another. This function can apply the transformation of a specified target coordinate system in two distinct ways:

- From the origin and orientation of the reference coordinate system, using **McalSetCoordinateSystem()** with **M_ASSIGN**. This method is useful to explicitly define the origin of the relative coordinate system or the tool coordinate system when they have not been previously defined. It is also useful when, for example, all movements of the target coordinate system are only known from the origin of the reference coordinate system.



- From its current position in a reference coordinate system using **M_COMPOSE_WITH_CURRENT**. The transformation is still defined in terms of the reference coordinate system. Essentially, the specified transformation is composed with the current position and orientation of the target coordinate system. This method is useful, for example, to translate the tool coordinate system, when you know the precise movement of a robotic arm in the absolute coordinate system.



❖ Note that the image coordinate system cannot be used with **McalSetCoordinateSystem()**.

The following code snippet shows how to use **McalSetCoordinateSystem()** to update your calibration object after rotating a robotic arm by 30 degrees. The robotic arm is the tool holding the camera; this means that the tool coordinate system should be rotated. Note that this will also rotate the camera if **M_LINK_TOOL_AND_CAMERA** is set to **M_ENABLE**, which is the default behavior.

```
/*
 * A robotic arm is holding the camera.
 * The arm rolls 30 degrees around its own axis (the Z axis)
 */

McalSetCoordinateSystem(CalibrationId, M_TOOL_COORDINATE_SYSTEM, M_TOOL_COORDINATE_SYSTEM,
                        M_ROTATION_Z + M_COMPOSE_WITH_CURRENT, M_NULL, 30.0,
                        M_DEFAULT, M_DEFAULT, M_DEFAULT);
```

Special considerations concerning the tool and camera coordinate systems

Calibration objects allocated in three-dimensional modes automatically define the camera coordinate system after a successful call to **McalGrid()** or **McalList()** with **M_FULL_CALIBRATION** and link it to the tool coordinate system such that moving one will move both. To move the camera coordinate system by assigning the tool coordinate system to known positions in another coordinate system, it is preferable to specify the actual real-world starting position and orientation of the tool coordinate system before calibrating (before calling **McalGrid()** or **McalList()**). If you have a camera mounted on a robotic arm, you can set the arm's position as the origin of the tool coordinate system. After a successful **M_TSAI_BASED** or **M_3D_ROBOTICS** calibration, you can move the tool coordinate system according to the movements of the arm and be assured that the camera coordinate system will move accordingly.

You can disable the link using **McalControl()** with **M_LINK_TOOL_AND_CAMERA** set to **M_DISABLE**. In this case, any change applied to the camera position will affect only the camera coordinate system and not the tool coordinate system and vice versa.

When working in robotics mode, you can specify the position and orientation of the robot tool, returned by a robot controller software, to the calibration object using **McalSetCoordinateSystem()**. Since the camera and robot tool are linked together, the camera is automatically positioned and oriented accordingly. The images grabbed by the camera at the new position and orientation are therefore correctly calibrated. The following code snippet shows how to provide data to the calibration object using **McalSetCoordinateSystem()**:

```
/*
 * Set the position and orientation of the tool coordinate system according
 * to the robot encoders.
 */

/* First, obtain the tool's position and orientation using the robot's software.*/

/* Second, call McalSetCoordinateSystem() twice.*/

McalSetCoordinateSystem(CalibrationId, M_TOOL_COORDINATE_SYSTEM, M_ROBOT_BASE_COORDINATE_SYSTEM,
    M_TRANSLATION + M_ASSIGN, M_NULL, ToolPosX, ToolPosY, ToolPosZ, M_DEFAULT);

McalSetCoordinateSystem(CalibrationId, M_TOOL_COORDINATE_SYSTEM, M_TOOL_COORDINATE_SYSTEM,
    M_ROTATION_XYZ + M_ASSIGN, M_NULL, AngleX, AngleY, AngleZ, M_DEFAULT);
```

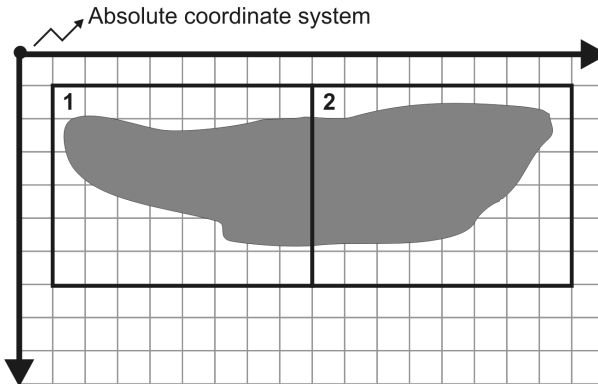
Calibrating a camera setup that analyzes large objects

The Camera 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 movable tool (manipulator): tool coordinate system 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 tool coordinate system has to be updated each time the tool is moved and before associating the calibration object with each image. To change the tool coordinate system, use **McalControl()** with **M_TOOL_POSITION_X**, **M_TOOL_POSITION_Y**, and **M_TOOL_POSITION_Z** for calibration objects using a two-dimensional calibration mode. For **M_TSAI_BASED** calibration objects, use **McalSetCoordinateSystem()** with **M_TOOL_COORDINATE_SYSTEM** as the target coordinate system.



1 = First field of view

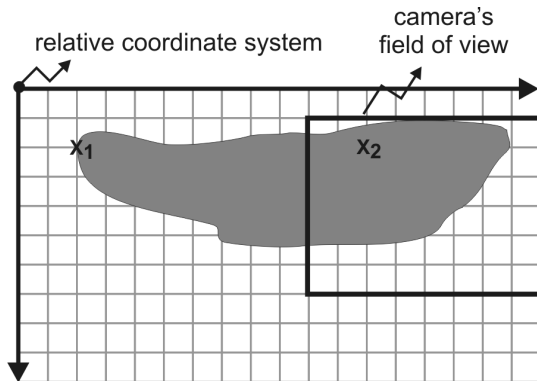
2 = Second field of view

For the first grab, the origin of the tool coordinate system was (1,1)

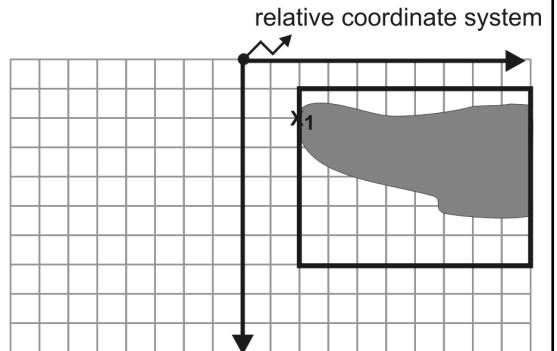
For the second grab, the origin of the tool coordinate system became (9,1)

Single camera and movable 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 relative coordinate system, the relative coordinate system must be moved each time the object is moved and before associating the calibration object with each image. To move the relative coordinate system, use **McalRelativeOrigin()** for calibration objects using a two-dimensional calibration mode. For **M_TSAI_BASED** calibration objects, use **McalSetCoordinateSystem()** with **M_RELATIVE_COORDINATE_SYSTEM** as the target coordinate system.



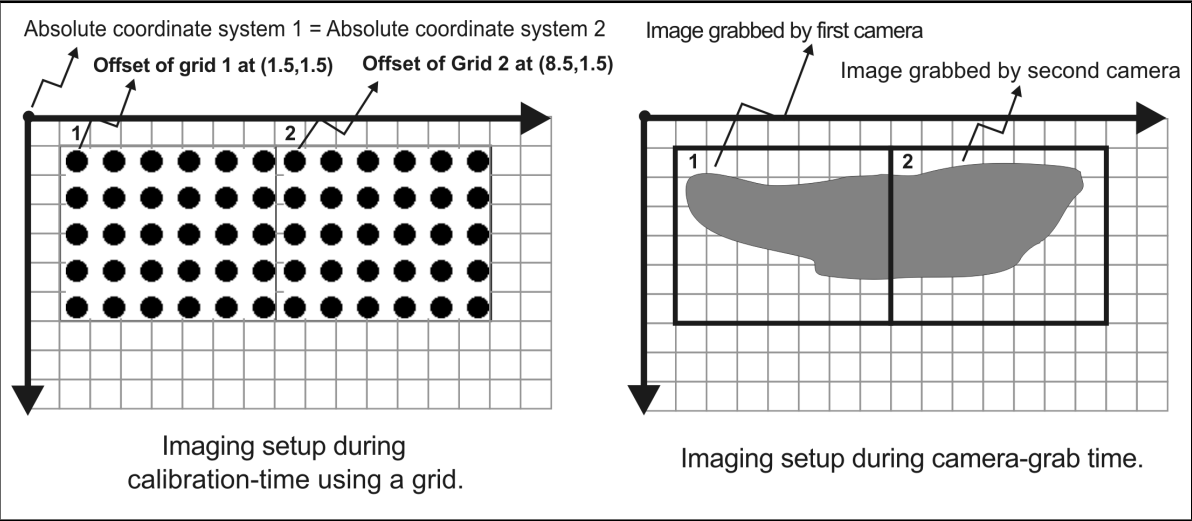
To measure the distance between x_1 and x_2 , the relative coordinate system must be moved along with the object. In this first grab, position x_2 maps to (12, 2).



The object and relative coordinate system are moved 8 world units to the right. Position x_1 maps to (2, 2) so the distance between x_1 and x_2 is 10 world units. Note that, if the relative coordinate system were not moved, x_1 would have mapped to (10, 2).

Several fixed cameras and fixed object: grid offset 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()**, all real-world coordinates used in each mapping must be coordinates from the same absolute coordinate system.



Processing calibrated images

This section provides information on possible operations that you can perform on calibrated images.

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 with the destination image only if all source images have the same calibration object; otherwise, no calibration object gets associated with the destination image.

- Geometric 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 considered uncalibrated always produce uncalibrated images. These functions include, for example, **MbufImport...** and **MbufLoad()**.

Saving and reloading a calibrated image

When saving a calibrated image using **MbufSave()** or **MbufExport()**, the image's calibration object is not saved with it. You must save the associated calibration object in a separate file using **McalSave()** or **McalStream()**.

To save a calibrated image and its Calibration object, do the following:

1. Obtain the identifier of the image's calibration object using **McalInquire()** with **M_ASSOCIATED_CALIBRATION**.
2. Save the calibration object using **McalSave()**.
3. Save the image buffer using **MbufSave()**.

To reload the image and associate it with its calibration object, do the following:

1. Restore the image buffer using **MbufRestore()**.
2. Restore the calibration object using **McalRestore()**.
3. Associate the restored image with the restored calibration object using **McalAssociate()**.

❖ Note that by using **McalInquire()** with **M_ASSOCIATED_CALIBRATION**, you can know whether an image is calibrated or not. If **M_NULL** is returned, the image is not calibrated.

Saving and reloading the child buffer of a calibrated image

When you associate a calibration object to an image, its child images are also associated with the calibration object; their offset from the parent image is automatically taken into account. If you save a child image into a separate file from its parent, it becomes an ordinary image. To reload a child image, you should associate it with the saved calibration object and then specify its original offset from the parent that was calibrated using **McalControl()** with **M_CALIBRATION_CHILD_OFFSET_X** and **M_CALIBRATION_CHILD_OFFSET_Y**.

- ❖ Note that if you create a calibration object for a child image, associate it to the child image, and then save the image separately from its parent, you will not need to specify its offset from its parent.

Chapter

6

Blob analysis

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

Blob analysis overview

Blob definition

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, maximum diameter at a given angle (Feret diameter), 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.

MIL also allows you to combine computation results and merge them together into one result. It is also sometimes desirable to merge blobs with specific characteristics. MIL supports this feature as well.

Supported buffers

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

Steps to performing blob analysis

The following steps provide a basic methodology for using the MIL Blob Analysis module:

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 significantly affecting real blobs. 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:
 - a. 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.
 - b. Specifying certain features as sorting keys. **MblobSelectFeature()**, **MblobSelectMoment()**, and **MblobSelectFeret()** each allow you to specify a selected feature as a sorting key. The calculated results are sorted in accordance with the sorting key selection.
 - c. Calculating results for the selected features, using **MblobCalculate()**. For this function, 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.
 - d. 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.

- e. Getting the number of blobs currently included, using **MblobGetNumber()**, and retrieving the results from the blob result buffer, using **MblobGetResult()**. You can also use **MblobGetResultSingle()** to obtain results for a single blob and use **MblobGetLabel()** and **MblobGetRuns()** to obtain more specific results. Note that trying to retrieve a result which is not available generates an error.

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.

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 typically 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 can be interpreted by MIL as a single blob. In addition, any part of a blob that is in the background pixel state, because of lighting or reflection, is considered as background during analysis.

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

Segmenting the blob image

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

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

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

Preprocessing

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

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

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

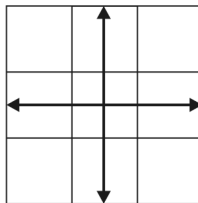
Adjusting blob analysis processing controls

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

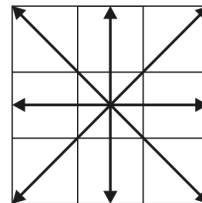
- Which pixel values are considered to be in the foreground (**M_FOREGROUND_VALUE**).
- Whether two pixels touching at their corners are considered part of the same blob, by appropriately defining the image lattice (**M_LATTICE**).
- Whether non-zero pixels can have any value or must be set to the maximum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

Controlling the image lattice

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

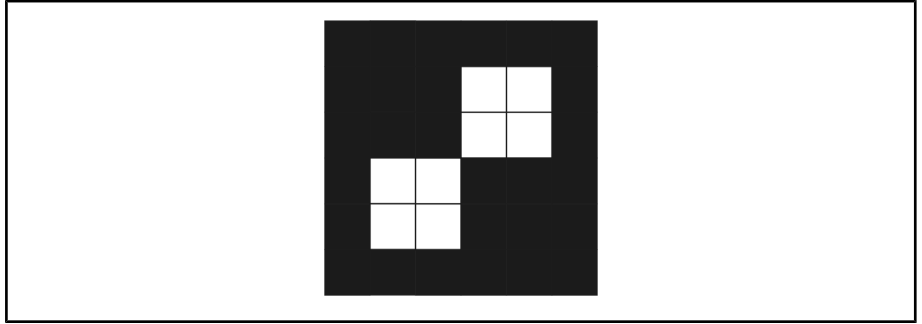


Lattice with 4 neighbor
connections
(**M_4_CONNECTED**)



Lattice with 8 neighbor
connections
(**M_8_CONNECTED**)

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



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:

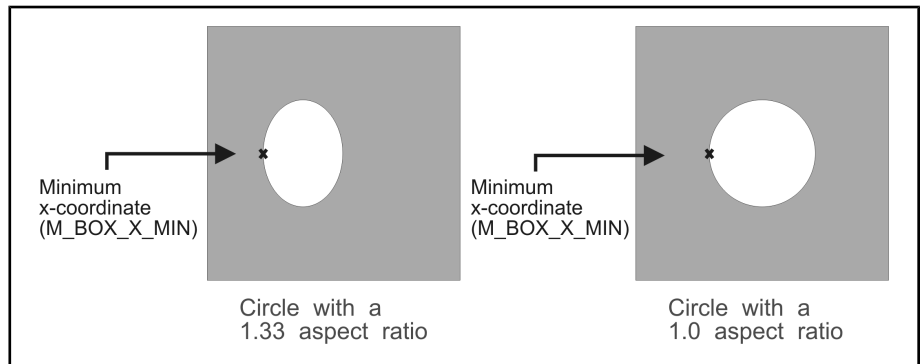
- You can adjust your image, using **MimResize()**, and then make the required blob analysis feature extractions.
- You can have calculations take the actual aspect ratio into consideration without modifying the image, by specifying the ratio, using **MblobControl()**.

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 ($\text{M_FERET_Y} / \text{M_FERET_X}$).

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 function, 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()** with the blob identification mode setting (**M_BLOB_IDENTIFICATION**), you can control how blobs in the blob identifier image are treated during calculations:

- All blobs are measured individually (**M_INDIVIDUAL**).
- All blobs are grouped together (**M_WHOLE_IMAGE**).
- Different blobs with the same label, or touching blobs with different labels, are grouped together (**M_LABELED**).
- Touching blobs with different labels are treated separately (**M_LABELED_TOUCHING**).

When using **M_LABELED** or **M_LABELED_TOUCHING**, **M_FOREGROUND_VALUE** cannot be set to **M_ZERO**, and the identifier image cannot be binary (1-bit). However, you can have the same behavior as **M_LABELED** or **M_LABELED_TOUCHING** with a binary (1-bit) identifier image, by setting **M_BLOB_IDENTIFICATION** to **M_WHOLE_IMAGE** and **M_IDENTIFIER_TYPE** to **M_BINARY**.

M_LABELED_TOUCHING is not supported with the following:

- **MblobCalculate()**, when chain features are selected (**MblobSelectFeature()** or **MblobSelect()** with **M_NUMBER_OF_CHAINED_PIXELS**, **M_CHAINS**, **M_CHAIN_X**, **M_CHAIN_Y**, **M_CHAIN_INDEX** or **M_ALL_FEATURES**).
- **MblobFill()** with **M_CONTOUR**.
- **MblobSelect()** with **M_MERGE**.

Calculations on blobs

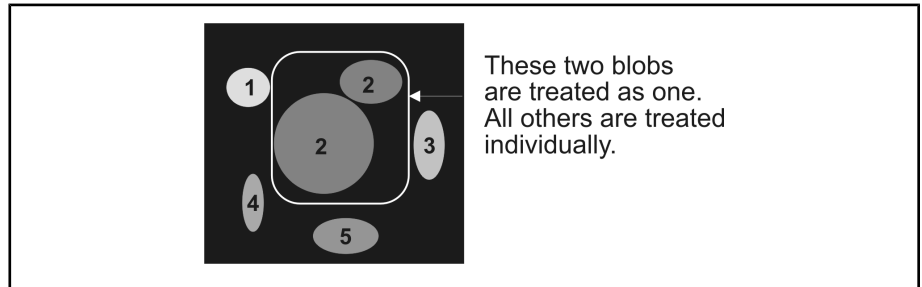
When using the blob analysis package, you usually want to make feature calculations on each blob. For example, if you want to find the area of each cell in a tissue image, set the blob identification mode to **M_INDIVIDUAL**.

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 each
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, or touching blobs with different labels. 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. If the blobs that you would like to group do not have the same label, you can use **MblobSelect()** with **M_MERGE** to group blobs together and assign a common label to them.



To treat touching blobs as distinct, use the blob identification mode **M_LABELED_TOUCHING**. With this mode, you can perform calculations on individual blobs even though they're touching, provided each blob has a different label.

Selecting blobs

Once all blobs are clearly identifiable by the Blob Analysis module, 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. You might choose to preprocess these blobs out of your image, but you might lose a significant amount of information by doing so. In some other cases, some blobs might have been incorrectly segmented as two or more blobs. You would want to group those blobs together and calculate their features as one blob. You might also want to do a blob calculation on the whole image but excluding the noise particles.

MblobSelect

The **MblobSelect()** function deals with these cases. This function allows you to select (on the basis of calculations already made) a subset of blobs for which to make further calculations, and to group blobs with certain characteristics and consider them as one blob.

When selecting a subset of blobs, your approach for selecting depends on whether you have few or many unwanted blobs.

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

If you are not able to exclude or delete a sufficient number of blobs using the second method, use the first.

When using **MblobSelect()** to specify a criterion for selecting blobs, you can set **CondLow** and **CondHigh** parameters as limits to your criterion selection. If the blobs were taken from a calibrated image, these limits can be set in pixel units or world units depending on **M_INPUT_SELECT_UNITS** control type setting in **MblobControl()**.

To group blobs that meet your criteria as one blob, use **MblobSelect()** with **M_MERGE**. Once grouped, the blobs are treated as one blob (a grouped blob) with a unique blob label. For example, if you want to calculate the area of the grouped blob, it will be the sum of the areas of the individual blobs within the group. Upon merging, only features that were calculated for all the individual blobs in the group are re-calculated for the new grouped blob; calculations are made using the results of the individual blobs in the group.

In any case, you can make as many calls as necessary to **MblobSelect()** and **MblobCalculate()** to arrive at the set of blobs with the right characteristics. 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.

Making feature extractions

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()** function 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()** function to add features to this feature list. You can, however, use the more specialized **MblobSelectMoment()** or **MblobSelectFeret()** functions to select a specific moment or Feret diameter, respectively.

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 for calculation, you must also provide the **MblobCalculate()** function with a grayscale image. The blob identifier image will identify the blobs, and the grayscale image will supply the actual blob pixel values.

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.

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.
- 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 functions fill blobs with their own label values (**MblobLabel()**) or with a user-specified value (**MblobFill()**).

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 the *Preparing for graphics* section in *Chapter 21: Generating graphics*) to control the drawing color, or use the default graphics context (**M_DEFAULT**). You can draw into any supported MIL image buffer. By drawing into its display's overlay buffer, you can also annotate the image non-destructively. For more information, see the *Annotating the displayed image nondestructively* section in *Chapter 20: Displaying an image*.

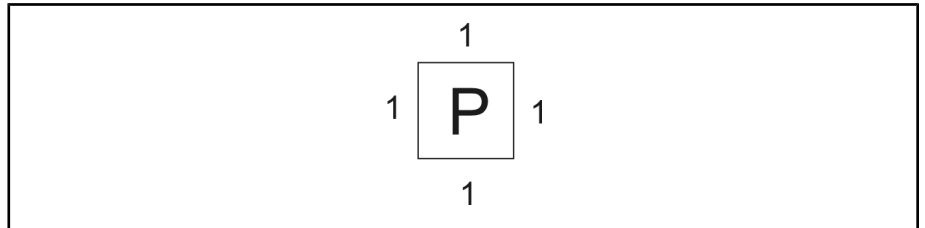
To get results, you can use **MblobGetResult()** or **MblobGetResultSingle()**. **MblobGetResult()** retrieves results for all blobs, whereas **MblobGetResultSingle()** retrieves results for the specified blob. To specify a specific blob to **MblobGetResultSingle()**, use the blob's label obtained from **MblobGetLabel()** or **MblobGetResult()**. The Blob Analysis module automatically calculates label values for included blobs when a call to **MblobCalculate()** is made. You can obtain the

label value for a specific blob using **MblobGetLabel()**, with the blob's coordinate. If you exclude some blobs from **MblobCalculate()**, the labels of the excluded blobs can still be returned until they are deleted; once deleted, the blob label returned will be 0.

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_SORTn_UP** or **M_SORTn_DOWN** to the features selected with the following blob functions: **MblobSelectFeature()**, **MblobSelectMoment()**, **MblobSelectFeret()**. Replace *n* with 1, 2, or 3 to indicate the sorting precedence of the feature(s).

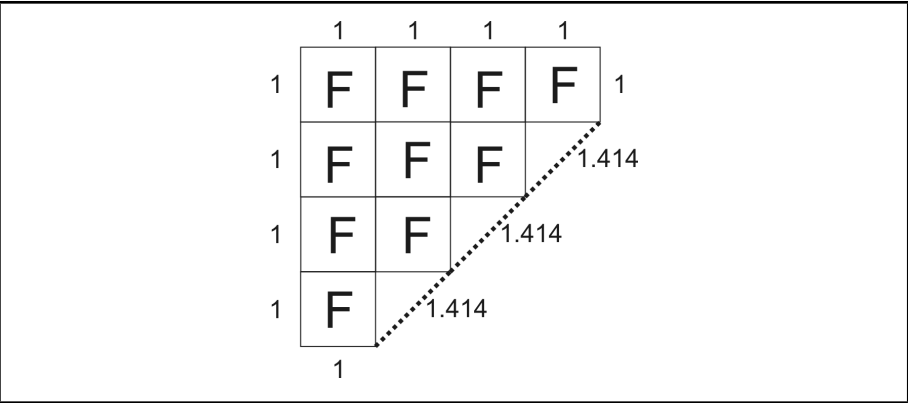
The area and perimeter

Each pixel in your image represents a real width and height (for example, in millimeters). However, all results from the blob analysis functions (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:

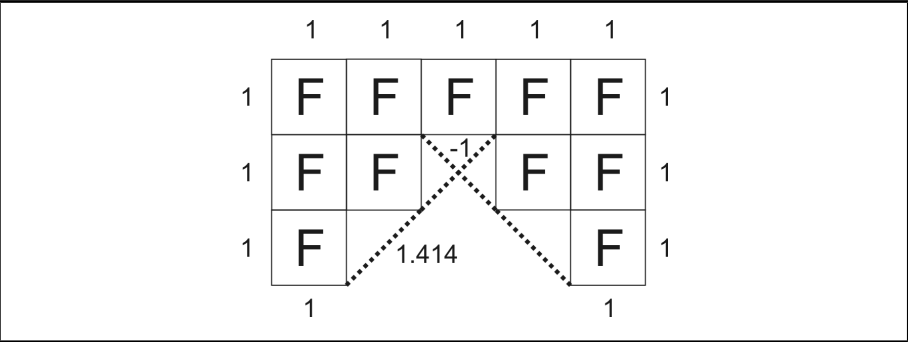


A pixel ratio of 1.0 implies that the area (**M_AREA**) of a single pixel blob is equal to 1 and the perimeter (**M_PERIMETER**) is equal to 4. When calculating the area and perimeter of a larger blob, the area would then equal the number of pixels in the blob (excluding holes), and the perimeter would equal the total number of

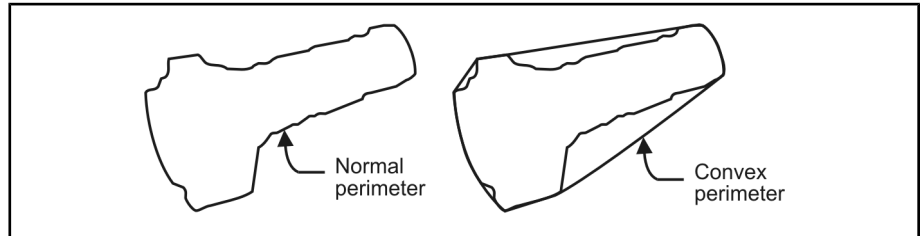
pixel sides along the blob edges (including the edges of holes). Note, an allowance is made for the staircase effect that occurs in a digital image when representing diagonals and curves. For example, in the following blob (where F represents foreground pixels), the area is 10 and the perimeter is 14.242.



When two diagonals intersect, as in the image below, the perimeter is calculated as the sum of the straight edges and of the diagonals (calculated as described above) with a correction of -1 for each intersection of diagonals. For example, in the following blob, the area is 11 and the perimeter is 17.656.

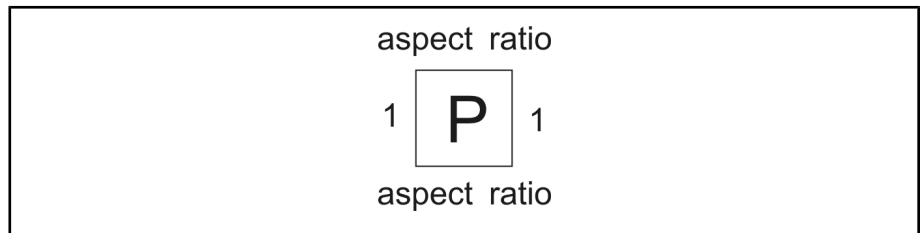


You can also calculate an approximation of the convex perimeter (**M_CONVEX_PERIMETER**) of the blobs. The convex perimeter is the perimeter of the convex hull (see below).



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()** function. The greater the number of Feret diameters used, the more accurate the approximation.

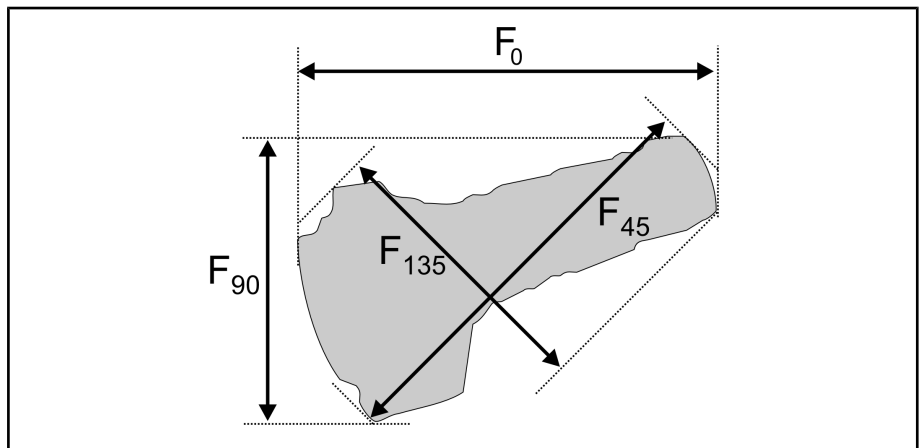
When the pixel aspect ratio has been set to anything other than 1.0, using **MblobControl()**, the aspect ratio is applied to the pixel width during calculations. Each pixel is now represented as:



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

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 .



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.

You can use **MblobControl()** with **M_NUMBER_OF_FERETS** to change the default number of angles; these angles will start at 0° and increase in increments of $180^\circ/n$, where n is the number of Feret diameters. Increasing the number of angles that are tested increases the accuracy of the results, but also increases processing time.

- ❖ Note that, since memory usage increases with the number of Feret diameters, using a large number of Feret diameters in an image with many blobs might result in memory allocation errors.

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 function.

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

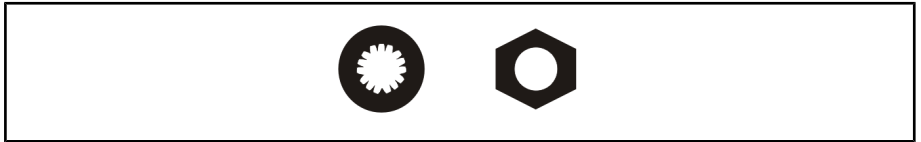
- **M_LENGTH**: an extraction of the true length of a blob.
- **M_BREADTH**: an extraction of the true breadth of a blob.
- **M_ELONGATION**: the ratio of the length to the breadth.

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.

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

- Compactness (**M_COMPACTNESS**).
- Roughness (**M_ROUGHNESS**).

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

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

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

For a program that, among other things, distinguishes between nuts, bolts and washers by analyzing the compactness of their holes, see the *Blob analysis example* section later in this chapter.

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 *BoltsNutsWashers.mim* image by counting blob holes. However, this is not a very robust measure since a single noise pixel in a bolt blob would count as a hole.

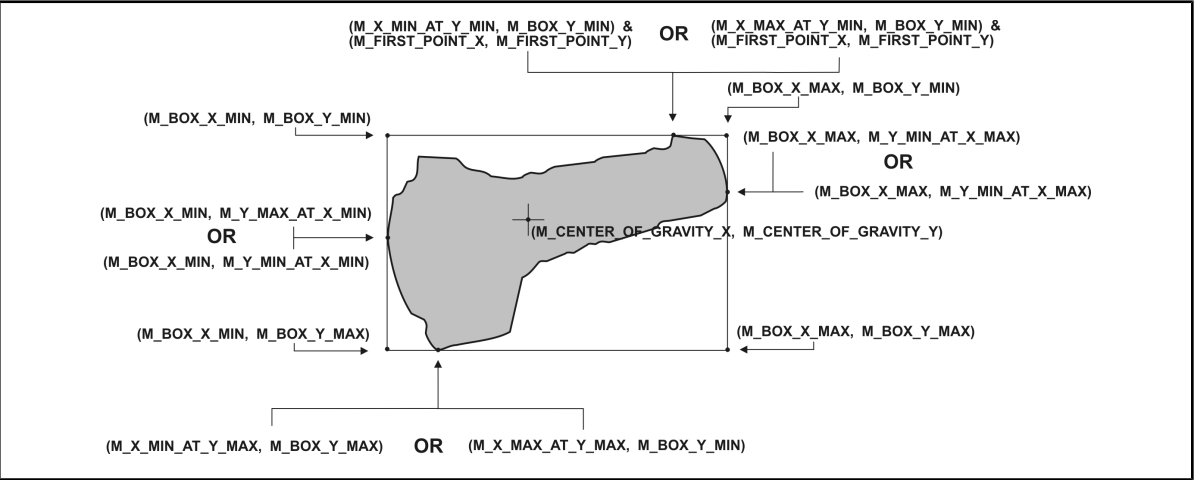
Finding the blob location

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.

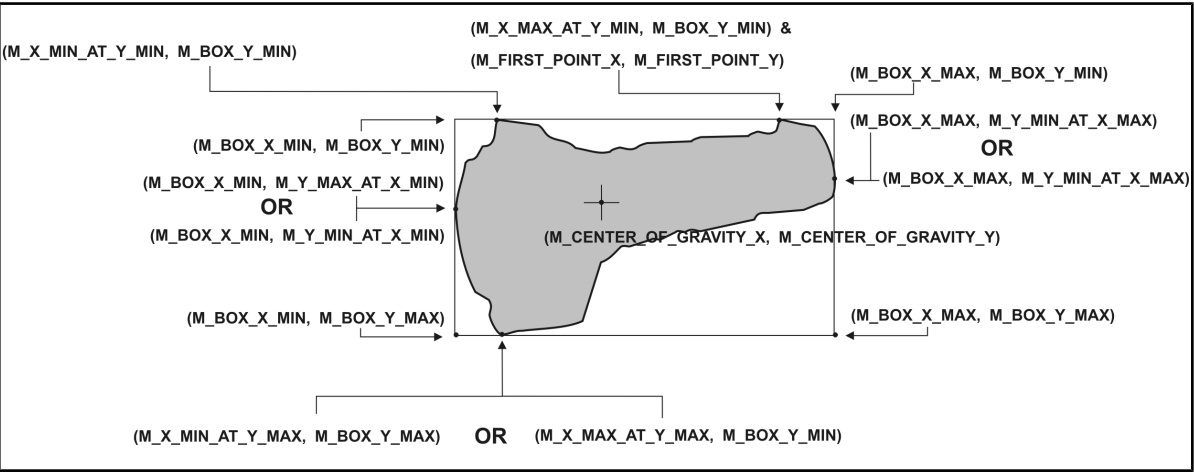
You can also use the blob location to determine if a blob touches any side of the image border. If there are any such blobs, you might want to adjust the camera's field of view so that all items are completely represented in the image, or you might want to exclude these blobs.

Blob points

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



Notice that there are two possible ways to call each blob point of contact. This is because there is only one point of contact on each side of the image border. In contrast, if there are multiple points of contact on one side of the image border, the names of the points of contact on that side will be different, as shown in the image below.



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

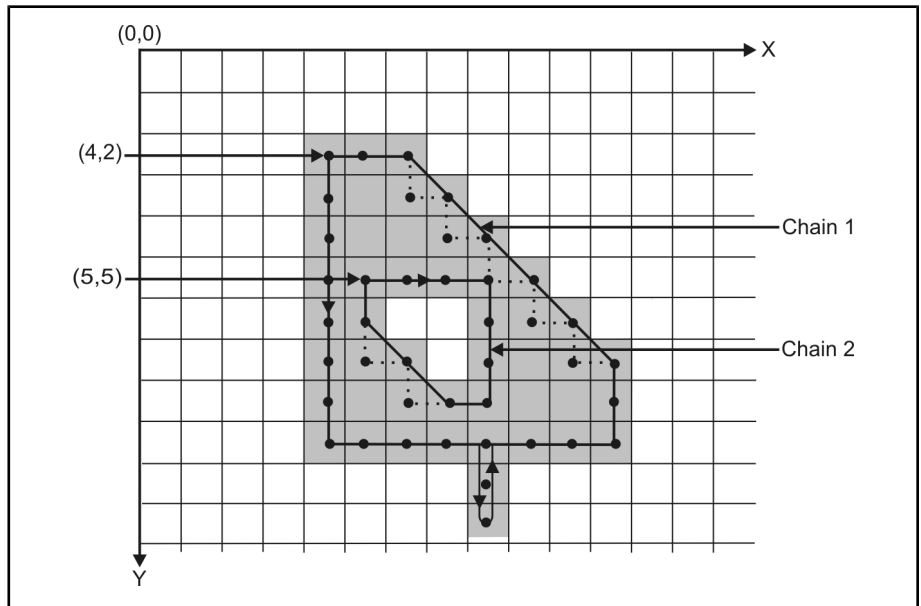
Chained pixels

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 indices.



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

When retrieving results for chain features, get the results for the number of chained pixels first (**M_NUMBER_OF_CHAINED_PIXELS**). This allows you to allocate an array that is large enough for the other chain results.

For the blob shown above, you would get the following results:

Chain index	Chain-X	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

As mentioned, blobs with holes have multiple chains. To determine the chain to which a pixel coordinate (**M_CHAIN_X** and **M_CHAIN_Y**) belongs, use **MblobGetResultSingle()** with **M_CHAIN_INDEX**.

Note that the results of the features **M_CHAIN_INDEX**, **M_CHAIN_X**, and **M_CHAIN_Y** are only available with **MblobGetResultSingle()**. All other feature results are available with both **MblobGetResultSingle()** and **MblobGetResult()**.

Moments

Using the Blob Analysis module, you can also calculate the moments used to find the center of gravity, as well as other grayscale and binary moments. The **MblobSelectMoment()** function allows you to add any moment to the feature list, whereas **MblobSelectFeature()** allows you to add only the more common moments (for example, **M_MOMENT_X0_Y2**). These common moments are calculated faster if selected by **MblobSelectFeature()**.

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.

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. You can obtain results using **MblobGetResult()** or **MblobGetResultSingle()**.

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

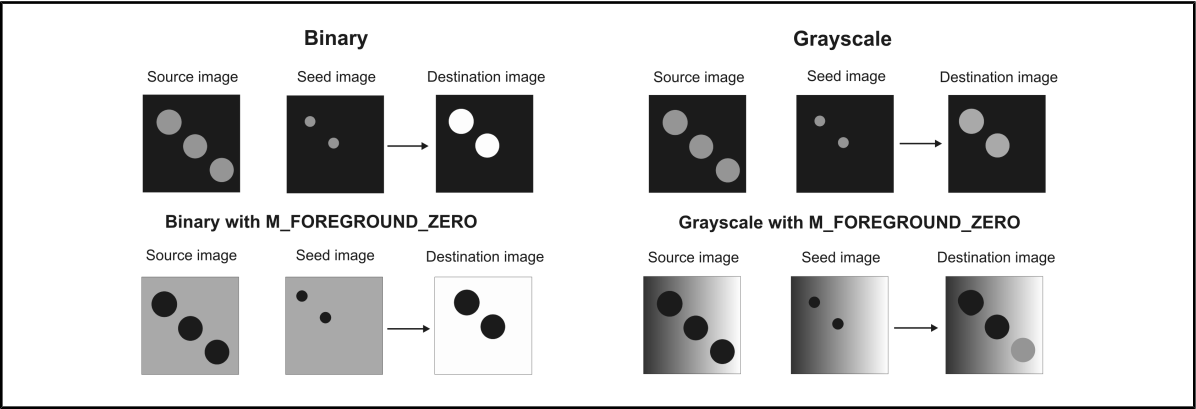
Blob reconstruction

Although the Blob Analysis module is used mainly for blob feature calculation purposes, some of the **Mblob...** functions can be used to perform blob image reconstruction. An example of this is the **MblobReconstruct()** function.

Blob reconstruction allows you to remove small particles in an image without distorting the shape of the remaining blobs, which can occur when successive erosion or dilation operations are performed on an image.

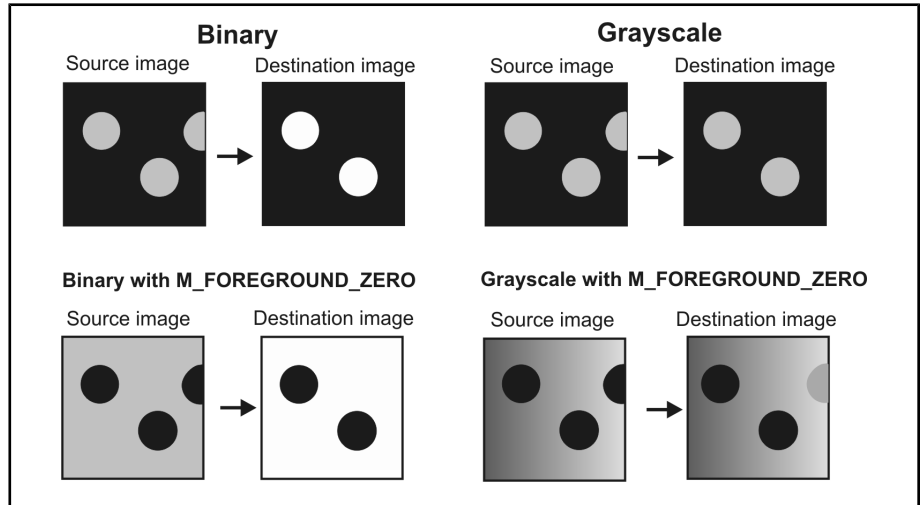
Using the different operations of the **MblobReconstruct()** function, you can:

- **Reconstruct blobs from a seed image.** The **M_RECONSTRUCT_FROM_SEED** operation copies to the destination buffer only those blobs that have a corresponding seed pixel in the seed buffer. Blobs that are not seeded are replaced according to the selected processing mode.



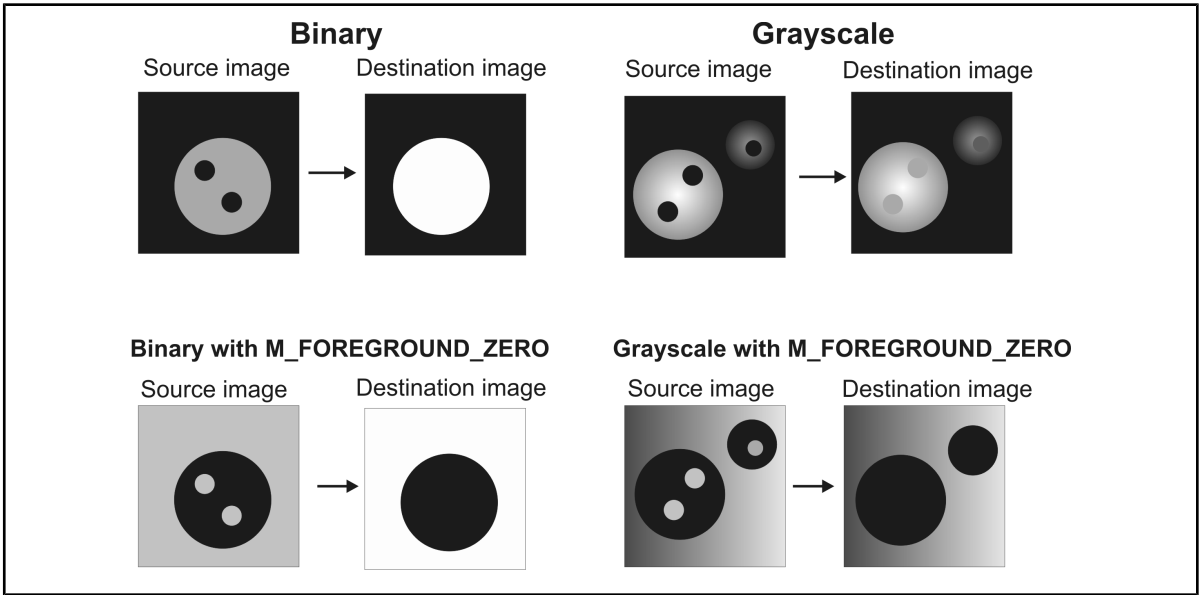
Note that when performing this operation in grayscale processing mode and with **M_FOREGROUND_ZERO** set, blobs in the source image which are not seeded are filled with the average grayscale value of the background.

- **Delete blobs that touch a border of the image.** The **M_ERASE_BORDER_BLOBS** operation copies only those blobs that do not touch the border to the destination buffer.



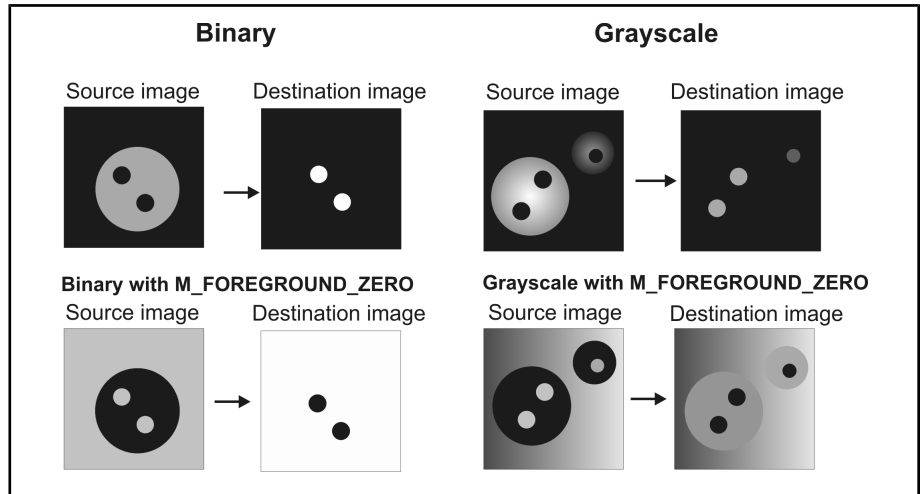
Note that when performing this operation in grayscale processing mode and with **M_FOREGROUND_ZERO** set, border blobs are filled with the average grayscale value of the background.

- **Fill holes in blobs.** The **M_FILL_HOLES** operation copies the source buffer to the destination buffer, filling those blobs that contain holes according to the selected processing mode. Holes that touch the image border are not considered to be holes.



Note that when performing this operation in grayscale processing mode, the holes are filled with the average grayscale value of the blob.

- **Extract holes from blobs.** The **M_EXTRACT_HOLES** operation copies only the holes within the blobs found in the source buffer. Holes that touch the image border are not considered to be holes.

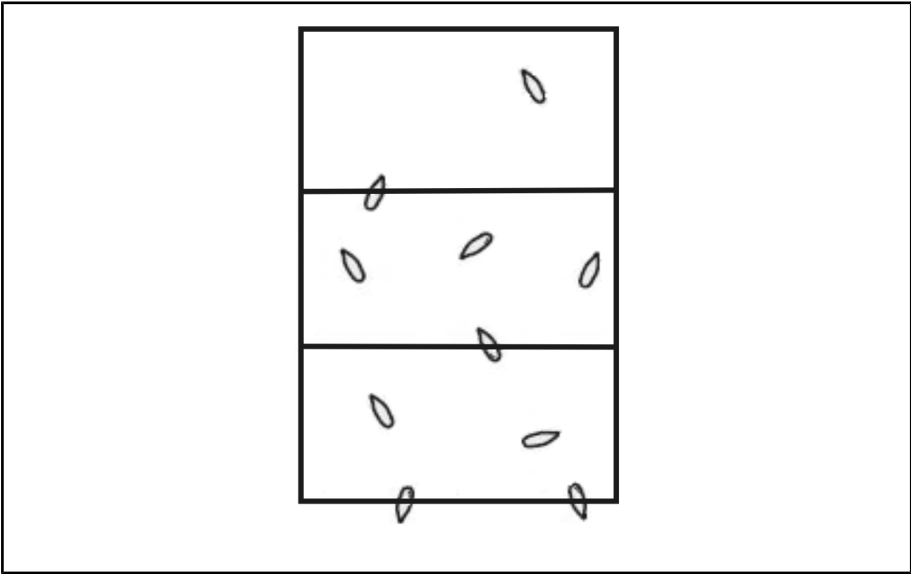


Note that when performing this operation in grayscale processing mode, the copied blob holes are filled with the average grayscale value of the blob in the source buffer. Also, when using the grayscale mode with **M_FOREGROUND_ZERO** set, the blobs are filled with the average grayscale value of the background.

Finally, you can perform other types of blob reconstruction by calling specific MIL functions, one after the other. For example, to fill all blobs composed of pixels with a required value, use **MblobSelect()** in conjunction with **MblobLabel()** and **MblobFill()**.

Merging results

In some cases, you might need to merge results from previous blob analysis calculations. For example, this might be required in an application where a line-scan camera captures a series of images of a continuous stream of rice falling off a conveyor belt as shown below. When grabbing images of the continuous stream, some objects might be split between two images. Performing blob analysis operations on these images produces erroneous results for these objects since an object split between two images is considered as two separate blobs. Results for one portion of the object will be in one result buffer and results for the other portion will be in another result buffer. To obtain valid results for these objects, you would need to merge the results of the objects that are split between two result buffers.



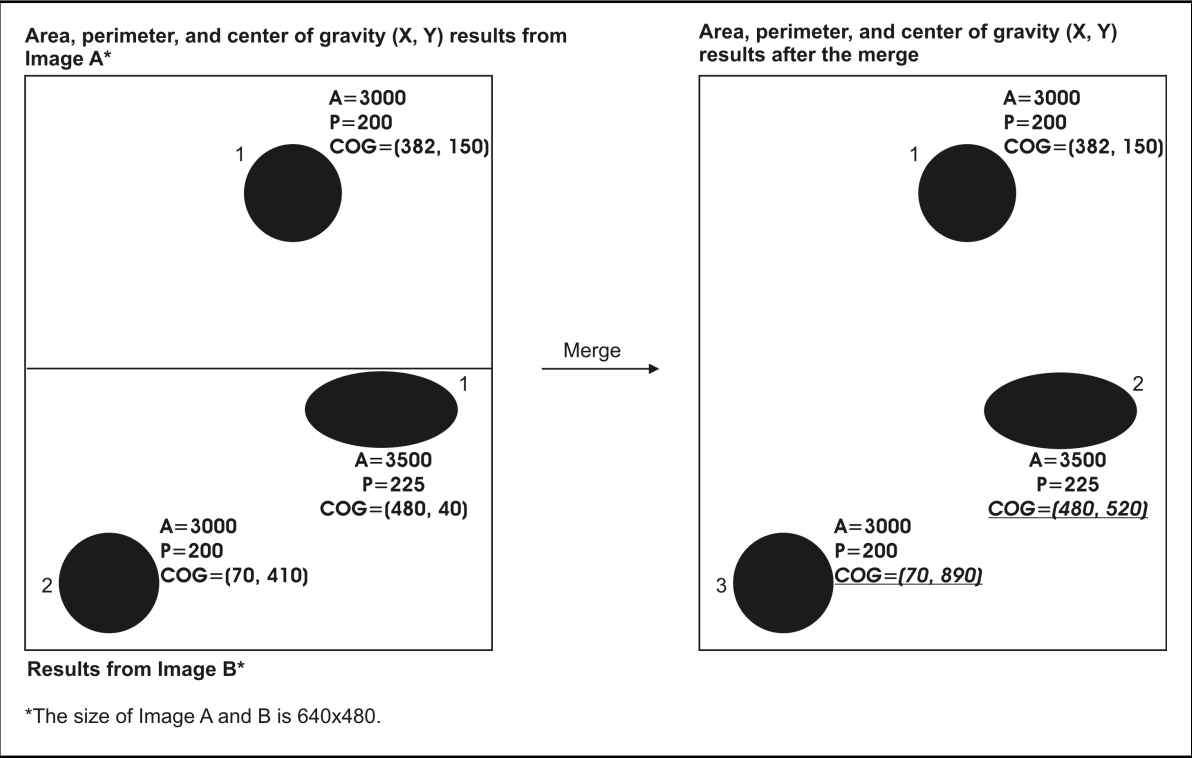
You can use **MdigProcess()** to grab sequential frames of the continuous stream into a list of image buffers (or child buffers of a large parent buffer), perform the required blob analysis operation on the frames as they are being grabbed, and store the results in different blob analysis result buffers. You can then merge the results from the required result buffers using **MblobMerge()**.

MblobMerge() merges two source result buffers at a time as if they were taken from two vertically adjacent child buffers of one image. By merging the result buffers as such, the destination result buffer uses the coordinate system of the first result buffer and positional results from the second result buffer are translated in the Y-direction to take into account this coordinate system change. In addition, border blobs that would touch if they were in one image are grouped into one grouped blob, assigned a single label, and recalculated as if the grouped blob were one blob. The merged results can be copied or moved (depending on the **ControlFlag** parameter of **MblobMerge()**) into a destination result buffer.

Call **MblobMerge()** function as many times as necessary to merge the results of all the required result buffers to obtain an ultimate blob analysis result buffer for your continuous stream.

A simple merge

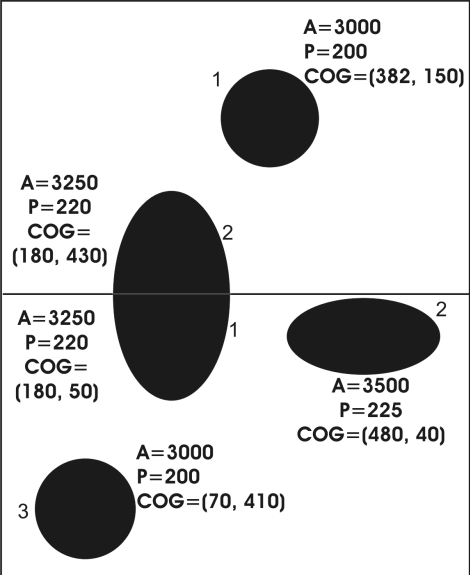
A simple merge operation is illustrated below. In this example, the area, perimeter, and center of gravity of blobs in two images have been calculated. After merging the results, the area and perimeter remain the same, but the center of gravity is recalculated with respect to the coordinate system of the destination result buffer. Also note that the labels of the blobs are changed after the merge to properly differentiate the blobs in the merged result buffer.



Border blobs

Most probably, you will have border blobs in your images that would touch if they were in one image. When merging results, these blobs are grouped into one grouped blob, assigned a single label, and recalculated as if the grouped blob were one blob. For example, in the illustration below, border blob 2 in image A and border blob 1 in image B have separate results before the merge. After the merge, however, these border blobs are considered as one blob with its own results in the new coordinate system.

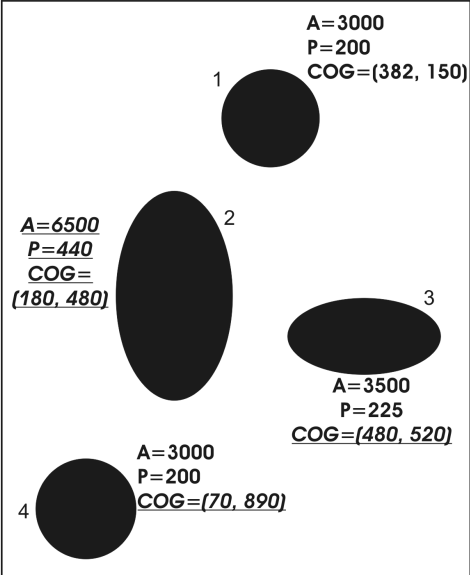
Area, perimeter, and center of gravity (X, Y) results from Image A*



Results from Image B*

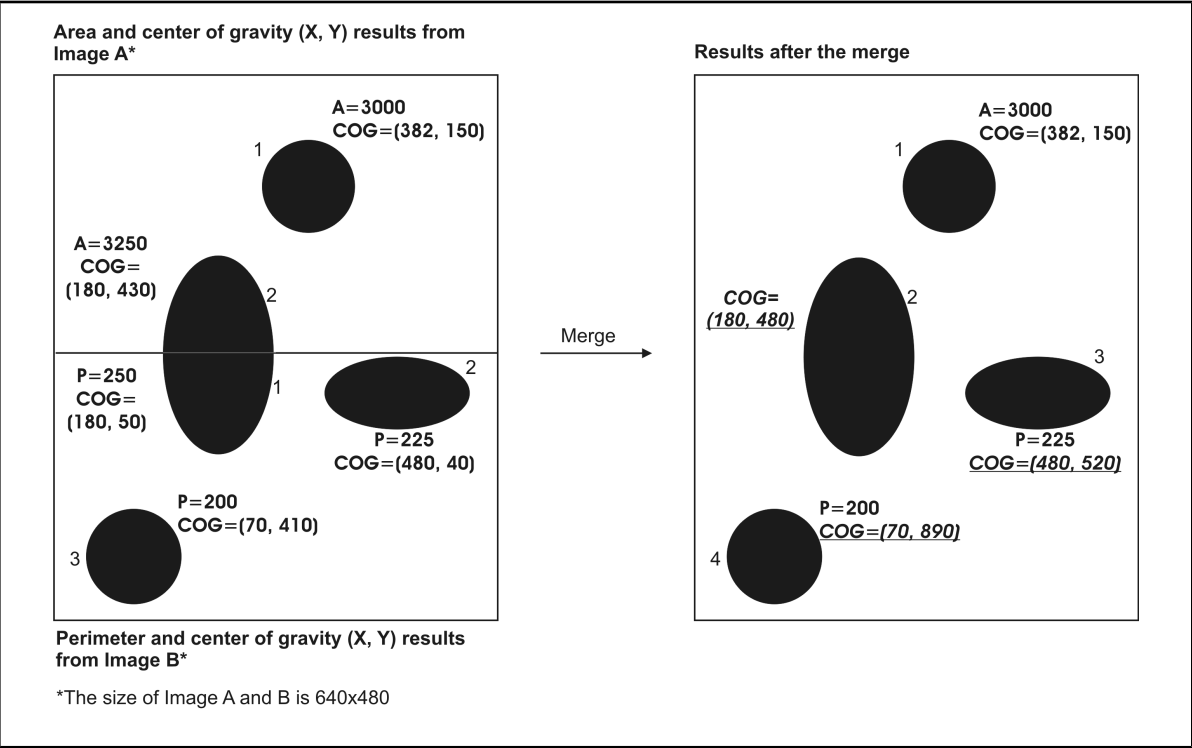
*The size of Image A and B is 640x480.

Area, perimeter, and center of gravity (X, Y) results after the merge



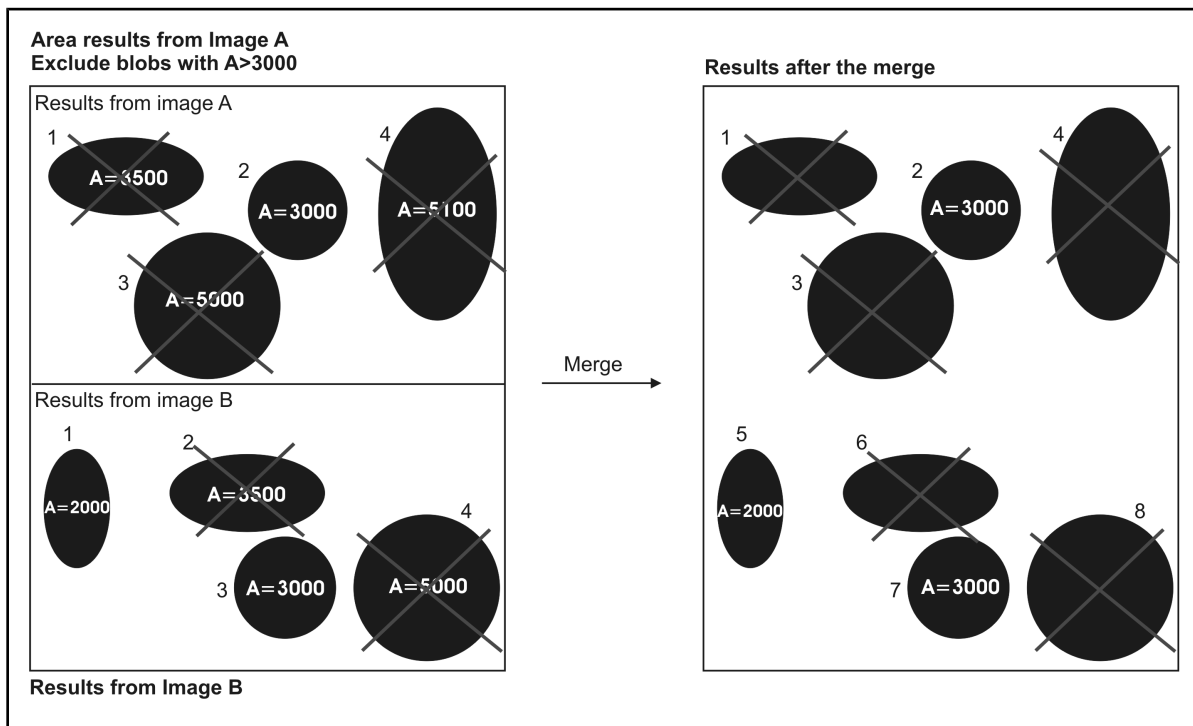
In the event that different blob analysis calculations are performed on the two images, only features that were calculated for all the individual blobs in the grouped blob are recalculated, and these are recalculated using the results of the individual blobs in the group. This case is illustrated below. The area and center of gravity are computed for image A, but the perimeter and center of gravity are computed for image B. Upon merging the results, only the center of gravity is computed for the grouped blob (blob 2).

Note that trying to retrieve a result in the destination result buffer (using **MblobGetResult()**) which is not available generates an error.

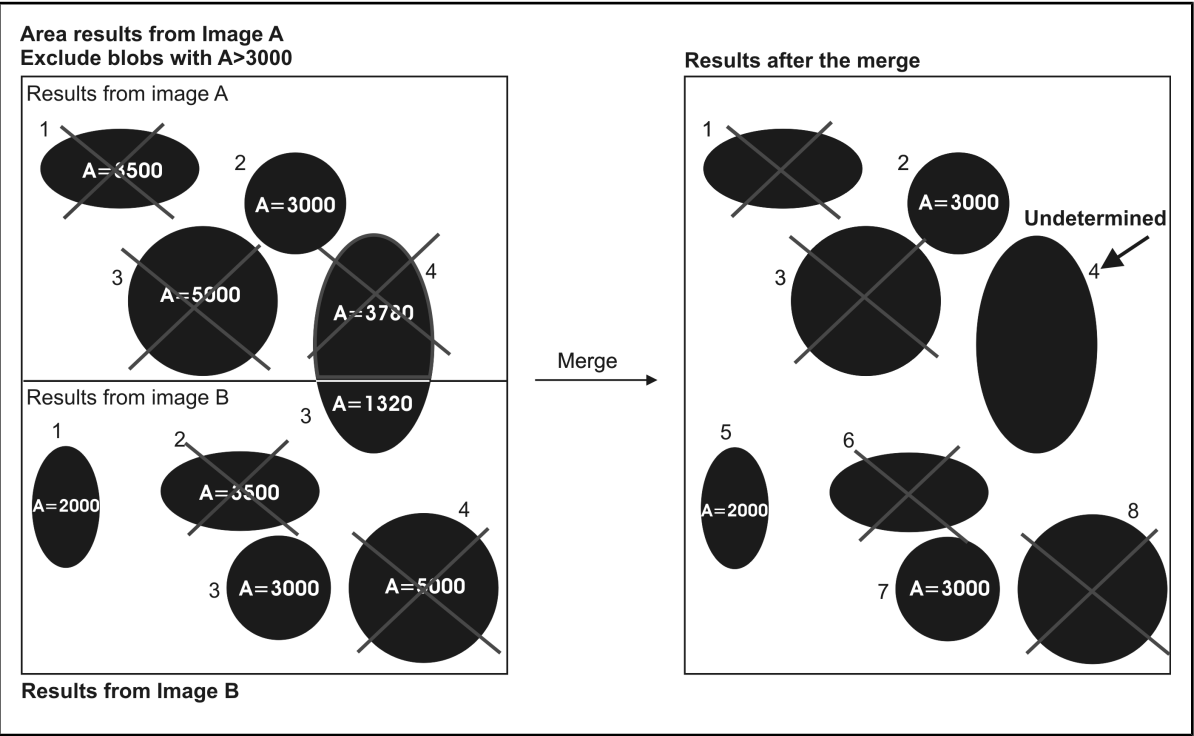


Inclusion state

In most cases, the inclusion state of the blobs remains the same after the merge. Consider the example below. The included blobs remain included after the merge, and the excluded ones remain excluded after the merge.



If an included border blob is grouped with an excluded blob, the inclusion state of the merged blob is undetermined, as illustrated below. For this reason, it is not recommended that you use the merge operation under these circumstances.



Note that after merging the results, you can perform a **MblobSelect()** operation to further include or exclude blobs in your destination result buffer, but if you perform an **MblobCalculate()** operation after the merge, all the results in the destination result buffer will be discarded.

Other remarks

For the merge operation to work properly, the results in the source result buffers must satisfy some constraints. The results in the source result buffers must:

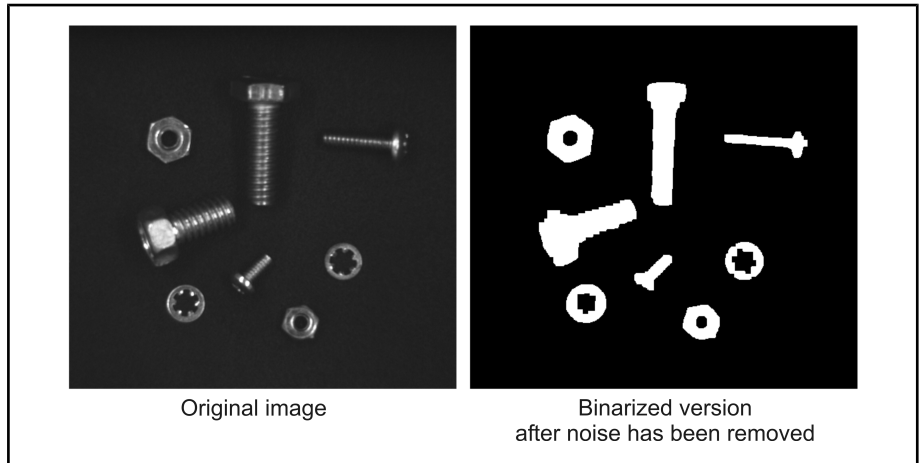
- Both have their run information saved or unsaved (**M_SAVE_RUNS** set to **M_ENABLE** or **M_DISABLE**).
- Have the same pixel aspect ratio (**M_PIXEL_ASPECT_RATIO**).
- Have the same lattice (**M_LATTICE**).
- Have the same blob identification mode (**M_BLOB_IDENTIFICATION**). If the blob result buffers have **M_BLOB_IDENTIFICATION** set to **M_LABELED**, they cannot be merged.
- Have the same number of Ferets (**M_NUMBER_OF_FERETS**).

Note that if you add a specified moment to the feature list (using **MblobSelectMoment()** with **M_GRAYSCALE+M_CENTRAL**, or with **M_BINARY+M_CENTRAL** and **M_SAVE_RUNS** disabled), **M_MERGE** will not be able to perform the moment calculation.

Furthermore, if you specify a sorting key for result retrieval, the sorting order will not be respected after the merge.

Blob analysis example

The following illustrates the source image before and after binarization and noise removal operations:



The example below binarizes an image, determines the center of gravity for each blob, and counts the number of bolts, nuts and washers. 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.cpp
 *
 * Synopsis:  This program loads an image of some nuts, bolts and washers,
 *            determines the number of each of these, finds and marks
 *            their center of gravity using the Blob analysis module.
 */
#include <mil.h>
#include <malloc.h>

/* Target MIL image file specifications. */
#define IMAGE_FILE          M_IMAGE_PATH MIL_TEXT("BoltsNutsWashers.mim")
#define IMAGE_THRESHOLD_VALUE 26L

/* 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

int MosMain(void)
{
    MIL_ID      MilApplication,          /* Application identifier.      */
               MilSystem,                /* System identifier.           */
               MilDisplay,               /* Display identifier.          */
               MilImage,                 /* Image buffer identifier.      */
               MilOverlayImage,          /* Overlay image.               */
               MilBinImage,              /* Binary image buffer identifier. */
               MilBlobResult,            /* Blob result buffer identifier. */
               MilBlobFeatureList;        /* Feature list identifier.      */
    MIL_INT     TotalBlobs,              /* Total number of blobs.       */
               BlobsWithHoles,           /* Number of blobs with holes.  */
               BlobsWithRoughHoles,      /* Number of blobs with rough holes. */
               n;                        /* Counter.                     */
    MIL_DOUBLE  *CogX,                   /* X coordinate of center of gravity. */
               *CogY;                   /* Y coordinate of center of gravity. */

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

    /* Restore source image into image buffer. */
    MbufRestore(IMAGE_FILE, MilSystem, &MilImage);

    /* Display the buffer and prepare for overlay annotations. */
    MdispSelect(MilDisplay, MilImage);
    MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
    MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

    /* Allocate a binary image buffer for fast processing. */
    MbufAlloc2d(MilSystem,
               MbufInquire(MilImage, M_SIZE_X, M_NULL),
               MbufInquire(MilImage, M_SIZE_Y, M_NULL),
               1+M_UNSIGNED, M_IMAGE+M_PROC, &MilBinImage);

    /* Pause to show the original image. */
    MosPrintf(MIL_TEXT("\nBLOB ANALYSIS:\n"));
    MosPrintf(MIL_TEXT("-----\n\n"));
    MosPrintf(MIL_TEXT("This program determines the number of bolts, nuts and washers\n"));
    MosPrintf(MIL_TEXT("in the image and finds their center of gravity.\n"));
    MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
    MosGetch();

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

```

```

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

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

/* Enable the Area and Center Of Gravity feature calculation. */
MblobSelectFeature(MilBlobFeatureList, M_AREA);
MblobSelectFeature(MilBlobFeatureList, M_CENTER_OF_GRAVITY);

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

/* Calculate selected features for each blob. */
MblobCalculate(MilBinImage, M_NULL, MilBlobFeatureList, MilBlobResult);

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

/* Get the total number of selected blobs. */
MblobGetNumber(MilBlobResult, &TotalBlobs);
MosPrintf(MIL_TEXT("There are %ld objects "), TotalBlobs);

/* Read and print the blob's center of gravity. */
if ((CogX = (MIL_DOUBLE *)malloc(TotalBlobs*sizeof(MIL_DOUBLE))) &&
    (CogY = (MIL_DOUBLE *)malloc(TotalBlobs*sizeof(MIL_DOUBLE)))
    )
{
    /* Get the results. */
    MblobGetResult(MilBlobResult, M_CENTER_OF_GRAVITY_X, CogX);
    MblobGetResult(MilBlobResult, M_CENTER_OF_GRAVITY_Y, CogY);

    /* Print the center of gravity of each blob. */
    MosPrintf(MIL_TEXT("and their centers of gravity are:\n"));
    for(n=0; n < TotalBlobs; n++)
        MosPrintf(MIL_TEXT("Blob #%ld: X=%5.1f, Y=%5.1f\n"), n, CogX[n], CogY[n]);
    free(CogX);
    free(CogY);
}
else
    MosPrintf(MIL_TEXT("\nError: Not enough memory.\n"));

/* Draw a cross at the center of gravity of each blob. */
MgraColor(M_DEFAULT, M_COLOR_RED);
MblobDraw(M_DEFAULT, MilBlobResult, MilOverlayImage, M_DRAW_CENTER_OF_GRAVITY,
            M_INCLUDED_BLOBS, M_DEFAULT);

/* Reverse what is considered to be the background so that
 * holes are seen as being blobs.
 */

```

```

MblobControl(MilBlobResult, 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(MilBlobFeatureList, M_COMPACTNESS);

/* Calculate selected features for each blob. */
MblobCalculate(MilBinImage, M_NULL, MilBlobFeatureList, MilBlobResult);

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

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

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

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

/* Print results. */
MosPrintf(MIL_TEXT("\nIdentified objects:\n"));
MosPrintf(MIL_TEXT("%ld bolts\n"), TotalBlobs-BlobsWithHoles);
MosPrintf(MIL_TEXT("%ld nuts\n"), BlobsWithHoles - BlobsWithRoughHoles);
MosPrintf(MIL_TEXT("%ld washers\n\n"), BlobsWithRoughHoles);
MosPrintf(MIL_TEXT("Press <Enter> to end.\n\n"));
MosGetch();

/* Free all allocations. */
MblobFree(MilBlobResult);
MblobFree(MilBlobFeatureList);
MbufFree(MilBinImage);
MbufFree(MilImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);

return 0;
}

```


Chapter

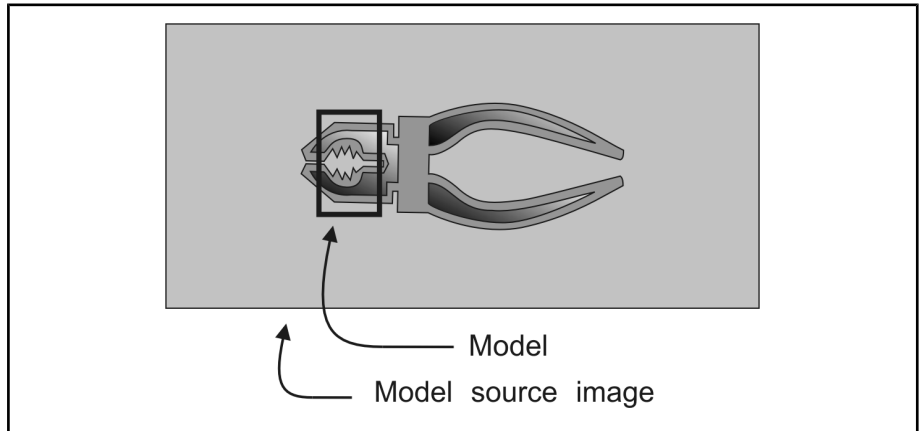
7

Pattern matching

This chapter explains how to use the MIL Pattern Matching module to perform normalized grayscale correlation (NGC) pattern matching.

Pattern matching - in general

The MIL package includes the Pattern Matching module that uses normalized grayscale correlation (NGC) to help solve machine vision problems such as alignment, measurement, and inspection of objects. 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 source image**.



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

This chapter describes various pattern matching techniques. It also provides guidelines on how to define a search model, find the occurrences of this model in the target image, and understand the search algorithm.

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

The Pattern Matching module also provides some support for calibration. Although the results are calculated in pixels, if the target image has been associated with a calibration object, positional results can be returned in calibrated real-world units.

Steps to performing a pattern search

The following steps provide a basic methodology for using the MIL Pattern Matching module:

1. Load or grab a model source image.
2. Define the model from the model source image. For information on defining a model, see the *Defining a model* section later in this chapter.
3. Optionally, specify a range for angular search. For information on specifying a range, see the *Rotation* section later in this chapter.
4. Optionally, mask any irrelevant areas of the model using **MpatSetDontCare()** to set the model's "don't care" pixels. For more information, see the *Masking the model* section later in this chapter.
5. Specify the model's search constraints. For more information, see the *Search constraints* section later in this chapter.
6. Preprocess the model, using **MpatPreprocModel()**. For information on preprocessing the model, see the *Preprocess the search model* section later in this chapter.
7. Allocate a result buffer to store the results of your search, using **MpatAllocResult()**.
8. Grab a target image. Optionally, process it to improve its quality.

9. Find instances of the model in the target image using **MpatFindModel()**. The search is performed according to the defined search constraints.

You can also search for several models that are of the same size and that use the same search region in the same image, using **MpatFindMultipleModel()**. This function finds occurrences of the specified models in the given image, and returns the position of each occurrence for each model or of the best matches from the group of models. Note that in the former case, you have to allocate and specify a result buffer for each model that is being sought. In the latter case, you have to allocate and specify a single result buffer. If you have to search for several different models, this is more efficient.

10. Read the search results. To read results, use **MpatGetNumber()** and **MpatGetResult()** to get the number of model occurrences found in the target, and the required results, respectively. You can use **MpatDraw()** to draw the occurrences' bounding box and/or a cross at the occurrences' reference position.

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 source image, use **MpatInquire()** with **M_ORIGINAL_X** and **M_ORIGINAL_Y**.

11. Free all your allocated objects, using **MpatFree()**.

In general, the first seven steps are performed once, while steps 8 through 11 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()**.

For examples on how to perform a search with a user-defined model, see the *Pattern matching examples* section later in this chapter.

Defining a model

To define a model, you can use **MpatAllocModel()** to define which portion of the model source image to use as your model, or you can use **MpatAllocAutoModel()** to have MIL automatically extract a unique model for you.

The model should be unique and not defined from a flat region (consistent pixel values with no edges present). Also, the model source image should be of the best quality possible. If noisy, try to clean the image, using the image processing techniques discussed previously in this manual.

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.

For information on defining a model that might appear at different angles in the target image, see the *Rotation* section later in this chapter.

Upon allocation, the model is extracted from the selected region in the model source image buffer and stored in a non-displayable model buffer. The model 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 *Chapter 21: Generating graphics*) to control the drawing color, or use the default graphics context (**M_DEFAULT**). You can also choose to draw in the display's overlay buffer. By drawing into the display's overlay buffer, you can annotate an image non-destructively. For more information, see *Chapter 20: Displaying an image*.

Models are usually saved on disk for future use, using **MpatSave()**. You can also set and save the model's search constraints. For information on setting a model's search constraints, see the *Steps to performing a pattern search* section earlier in this chapter.

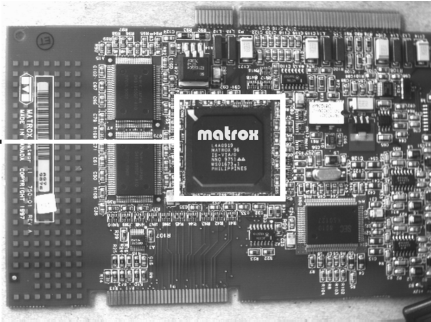
Rotation

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

Model

An `M_NORMALIZED` model internally rotated

An `M_NORMALIZED+M_CIRCULAR_OVERSCAN` model internally rotated

■ = resulting `M_DONT_CARE` pixels

Creating rotated versions of the models

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

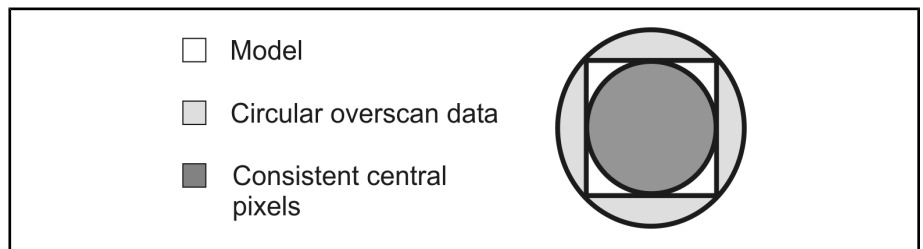
To implement the first type of search, allocate an **M_NORMALIZED** model with **MpatAllocModel()**. Then, enable and specify the angular range in which it can appear with **MpatSetAngle()**. When you call **MpatPreprocModel()**, it will internally create different models by rotating the original model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

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 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.

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. Therefore, it is recommended that an **M_NORMALIZED** + **M_CIRCULAR_OVERSCAN** model be as square as possible: the longer the rectangle, the smaller the number of consistent central pixels in every model.



As mentioned, a larger region than the one defined will be fetched from the model source image. Therefore, the model must not be extracted from a region too close to the edge of the model source image.

The pixels surrounding the model should be relevant to the positioning of the pattern (that is, the model should appear in the target image with the same overscan data). An example is the image of an integrated circuit.

Both methods find the position and match score of the model in a target image.

Finally, it should be noted that MIL's implementation of **MpatFindModel()** with a **M_CIRCULAR_OVERSCAN** type model is significantly faster than that of a **M_NORMALIZED** model when performing an angular search.

Setting the angle of search

By default, the angle of search is 0°. However, you can specify a rotational range of up to 360°, as well as the required precision of the resulting angle and the interpolation mode used for the rotated model. These settings can influence the speed of the search significantly. The accuracy of the search can also be influenced.

When an angular range has been specified with **MpatSetAngle()**, **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

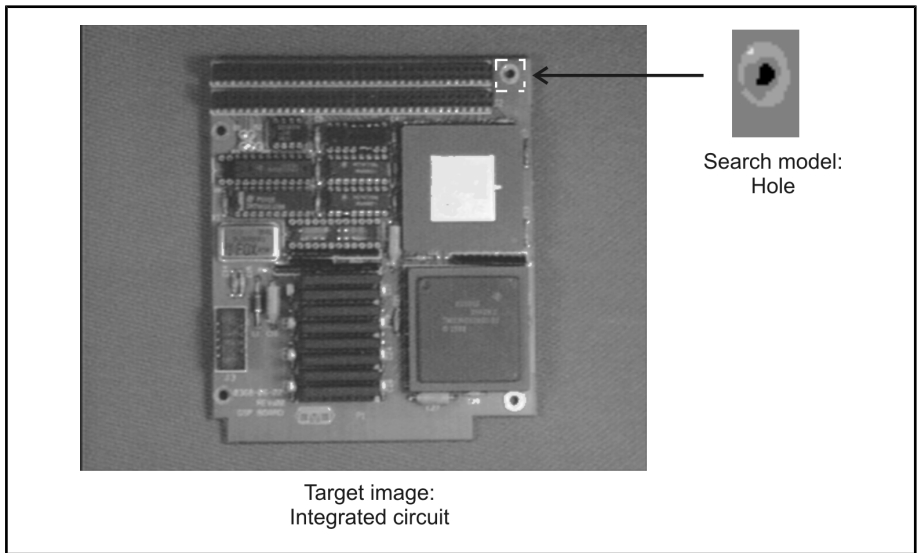
Every model has its own particular rotation tolerance. This tolerance is dependent on the individual model characteristics and surrounding target image features. To determine the rotation tolerance of a model:

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°), 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 and 2, 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.

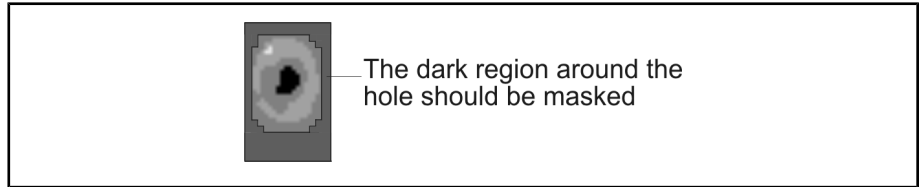
For instance, in a machine guidance application, a mechanical device might need to know where mounting holes are located on a circuit board so that screws can be properly inserted. In this case, a mounting hole would be the search model and the circuit board would be the target image.



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 can be masked by setting pixel values in certain regions to *"don't care"* pixels. 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. Allocate an image that is of the same size as the model image. This new image is the **mask image**.
2. Clear the mask image to one pixel value. This is the background color of the mask image.
3. Set the *"don't care"* pixels in the mask image to a specific value, different from the background. This is the foreground color of the mask image. There are numerous ways of doing this. For example, you can use one of the **Mgra...** functions.

To establish which pixels to set to *"don't care"*, you can refer to a copy of the model image obtained using **MpatCopy()** with **M_DEFAULT**.

4. Call **MpatSetDontCare()**, specifying the mask image along with the foreground color used to draw the *"don't care"* pixels. This function sets the model's *"don't care"* pixels.
5. View the mask using **MpatDraw()** with **M_DRAW_DONT_CARES**. Alternatively, you can call **MpatCopy()** with **M_DONT_CARE**.

When you change the *"don't care"* pixels of a model, you should preprocess the search model again.

Search constraints

When a model is defined (whether manually or automatically), it is assigned a set of default search constraints. You can change the following constraints:

- 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 the required number of occurrences to one (the default value) and avoid unnecessary searches for further matches. If a correlation has a match score greater than or equal to the certainty level, it is automatically considered an occurrence (default 80%), the remaining occurrences will be the best of those greater than or equal to the acceptance level.

When you ask for a specific number of matches (using **MpatSetNumber()**), the **MpatFindModel()** function 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 acceptance level

The level at which the correlation (match score) between the model and the pattern in the image is considered a match is called the acceptance level.

You can set the acceptance level for the specified model, using **MpatSetAcceptance()**. If the correlation between the target image and the model is less than this level, they are not considered a match. A perfect match is 100%, a typical match is 80% or higher (depending on the image), and no correlation is 0%. If your images have considerable noise and/or distortion, you might have to set the level below the default value of 70%. 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.

Setting the certainty level

The certainty level is the match score (usually higher than that of the acceptance level) above (or equal to) 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, using **MpatSetCertainty()**. 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.

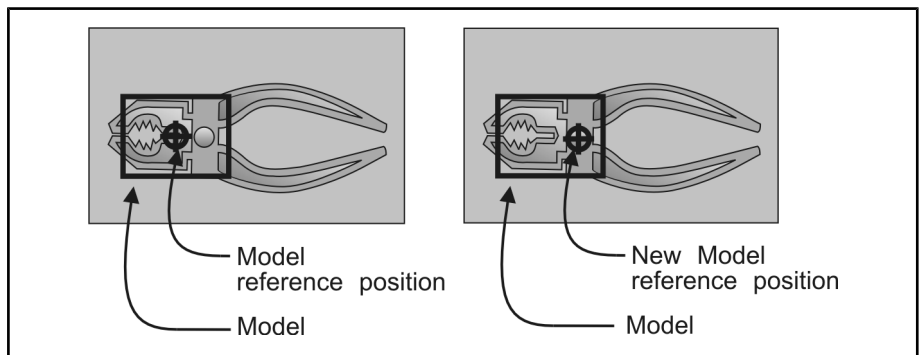
Often, you know that the pattern you want is unique in the image, so anything that reaches the acceptance level must be the match you want; therefore, you can set the certainty and acceptance levels to the same value.

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. However, you cannot set the certainty level to 50% because 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

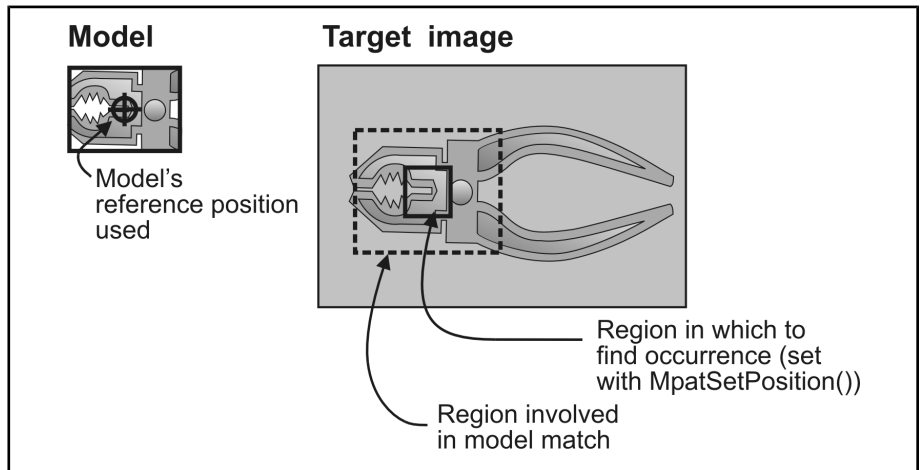
The coordinates returned by `MpatGetResult()`, after a call to `MpatFindModel()`, are the coordinates of the model's reference position (in pixel or real-world units, depending on whether the camera setup is calibrated; for more information, see *Chapter 5: Camera calibration*). By default, this reference position is defined to be at the geometric center. Note that, when using pixel units, results are returned relative to the top-left corner of the target image.

If there is a particular spot from which you would like results returned, you can change the model's reference position, using `MpatSetCenter()`. For example, if your model has a hole and you want to find results with respect to this hole, change the reference position of the model accordingly. Note that you can define the reference position to be outside of the model's boundary.



Selecting the search region

Instead of searching the entire region of an image, you can limit the search region with **MpatSetPosition()**. This function specifies the region in which to find the model's reference position. Therefore, the search region can even be smaller than the model. If you have redefined the model's reference position (with **MpatSetCenter()**), make sure that the search region defined by **MpatSetPosition()** covers this new reference position and takes into account the angular search range of the model.



In general, you should not use a child buffer to delimit the search region to a portion of an image; this might cause the search routine to have border or edge effects and be less accurate (the routine does not assume that there is valid data outside of the buffer).

Search time is roughly proportional to the region searched; always set the search region to the minimum required when speed is a consideration.

Positional accuracy

You can set the positional accuracy for your search. Use **MpatSetAccuracy()** to set the required positional accuracy. It can be set to:

- **M_LOW** (typically ± 0.20 pixel).
- **M_MEDIUM** (typically ± 0.10 pixel).
- **M_HIGH** (typically ± 0.05 pixel).

Note, the actual precision achieved is dependent on the quality of the model and of the image (the tolerances listed above are typical for high-quality, low-noise images).

A less precise positional accuracy will speed up the search. Positional accuracy is also slightly affected by the search speed parameter (**MpatSetSpeed()**).

Setting the speed parameter

You can specify the algorithm's search speed, using **MpatSetSpeed()**. When the search speed is set to **M_VERY_HIGH** or **M_HIGH**, the search algorithm takes more shortcuts, and the search is performed faster. However, as you increase the speed, the robustness of the search operation (the likelihood of finding a model) can decrease. For more information on search speed, see the *Speeding up the search* section later in this chapter.

Preprocess the search model

When 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 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 setting

For any search model, it is possible to set the speed of the search. Increasing the search speed reduces the search time, however, as you increase the speed, the robustness of the search operation (the likelihood of finding a model) can decrease. When you call **MpatPreprocModel()**, MIL analyzes the pattern in the model, and determines appropriate shortcuts; 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 can reduce the positional accuracy very slightly.

You adjust the search speed setting, using **MpatSetSpeed()**. This function 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° 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 settings are 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 the *Custom spatial filters* section in *Chapter 4: Advanced image processing*). 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*");

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

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:

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

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:

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

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

The sums used to compute the correlation function can be retrieved using **MpatGetResult()**. The retrievable sums are the following:

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

The number of pixels N is also available using **MpatGetResult()**. The above sums are only available if they are saved in the result buffer using

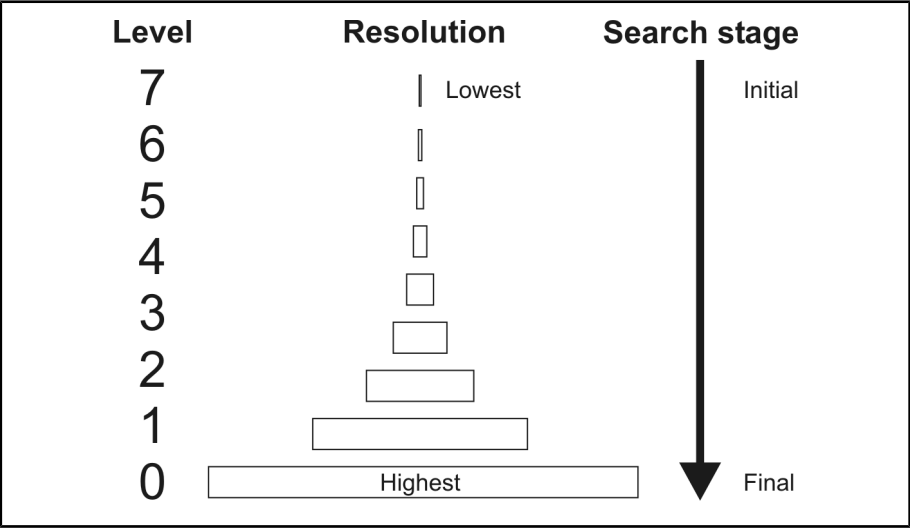
MpatSetSearchParameter() with **M_SAVE_SUMS** set to **M_ENABLE**. These sums can be used, for example, to assess the model or target intensity level and contrast.

- ❖ Note that the correlation function calculated with the retrieved sums, will probably differ from the score returned by **MpatGetResult()**. Although the score is derived from the correlation function, it also depends on the first and last subsampling levels chosen, optimization schemes and subpixel interpolation.

On a typical computer, 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 computer, 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 512×512 at level 0, then at level 1 it is 256×256 , at level 2 it is 128×128 , 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 (*128x128* model and *512x512* image), it might start the search at level 4, which would mean using an *8x8* version of the model and a *32x32* version of the target image.

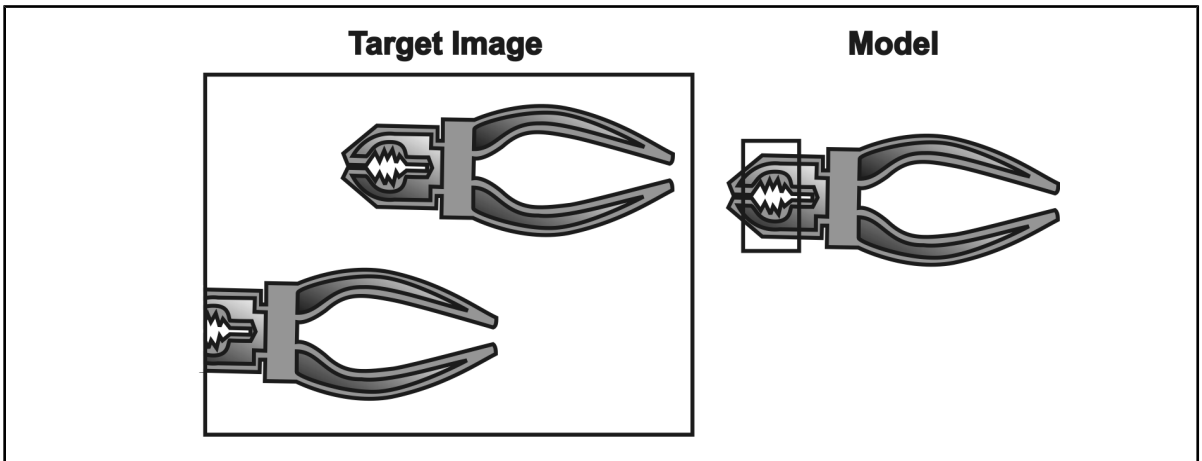
To determine the default first and last resolution levels, use **MpatInquire()** with **M_PROC_FIRST_LEVEL** and **M_PROC_LAST_LEVEL**, respectively. If required, you can change the first and last resolution levels, using **MpatSetSearchParameter()** with **M_FIRST_LEVEL** and **M_LAST_LEVEL**, respectively.

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 region

A zero-overscan technique is used to search for partially occluded occurrences near the border of the target image. However, this can lead to lower score and lower accuracy. In the example below, the left-most occurrence, which overlaps the border of the target image, would yield a lower score than the right-most occurrence.



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.

Subpixel accuracy

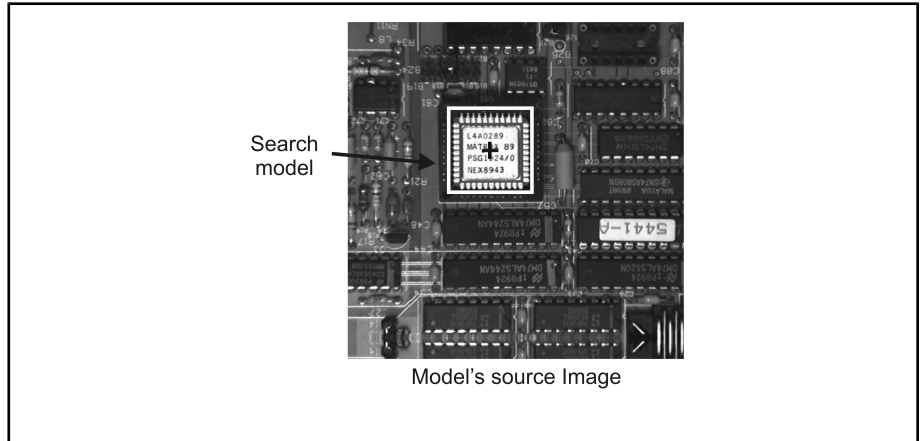
The highest match score occurs at only one point, and drops off around this peak. The exact (subpixel) position of the model can be estimated from the match scores found on either side of the peak. A surface is fitted to the match scores around the peak and, from the equation of the surface, the exact 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-accuracy search, 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).

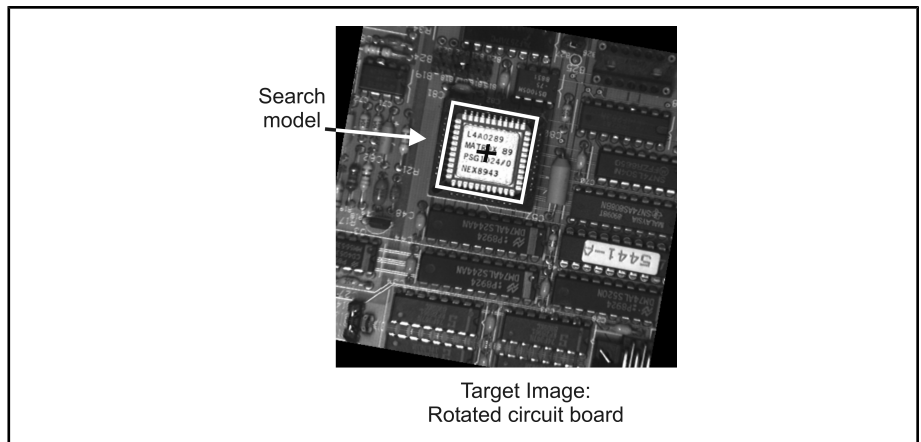
Pattern matching examples

MPat.cpp contains 3 examples that use the NGC Pattern Matching module. The first and second examples of *MPat.cpp* use the following image as source image:



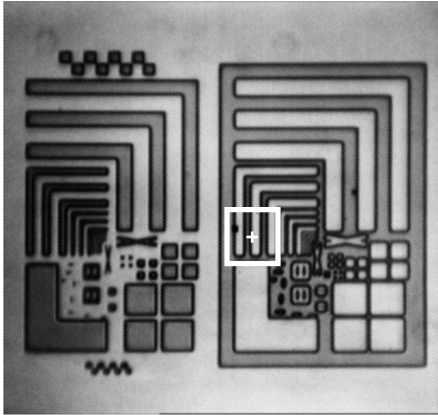
This first example defines a model and searches for it in a shifted version of the image (without rotation). It also demonstrates the subpixel accuracy of **MpatFindModel()**. Refer to the **SearchModelExample** function for this first example.

The second example of *MPat.cpp* uses the following image as target image:

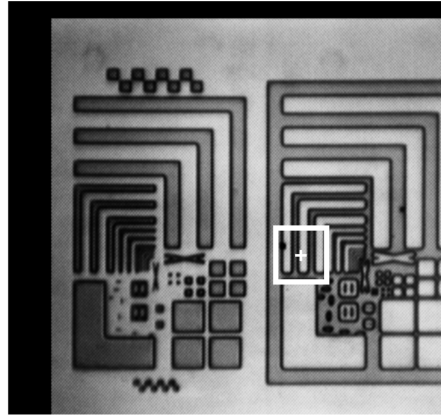


This example defines a model and searches for it in a rotated version of the image. Refer to the **SearchRotatedModelExample** function for this second example.

The third example of *MPat.cpp* uses the following images:



Model's source image



Target image

This example automatically allocates a model in a wafer image and finds its horizontal and vertical displacement. Refer to the **AutoAllocationModelExample** function for this third example.

```

/*****
/*
* File name: MPat.cpp
*
* Synopsis: This program contains 3 examples of the pattern matching module:
*
* The first example defines a model and then searches for it in a shifted
* version of the image (without rotation).
*
* The second example defines a model and then searches for it in a
* rotated version of the image.
*
* The third example automatically allocates a model in a wafer image and finds
* its horizontal and vertical displacement.
*/
#include <mil.h>
#include <math.h>

```

```

/* Example functions declarations. */
void SearchModelExample(MIL_ID MilSystem, MIL_ID MilDisplay);
void SearchRotatedModelExample(MIL_ID MilSystem, MIL_ID MilDisplay);
void AutoAllocationModelExample(MIL_ID MilSystem, MIL_ID MilDisplay);

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

    MosPrintf(MIL_TEXT("\nGRAYSCALE PATTERN MATCHING:\n"));
    MosPrintf(MIL_TEXT("-----\n\n"));

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

    /* Run the search at 0 degree example. */
    SearchModelExample(MilSystem, MilDisplay);

    /* Run the search over 360 degrees example. */
    SearchRotatedModelExample(MilSystem, MilDisplay);

    /* Run the automatic model allocation example. */
    AutoAllocationModelExample(MilSystem, MilDisplay);

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

    return 0;
}

/*****/
/* Find model in shifted version of the image example. */

/* Source image file name. */
#define FIND_IMAGE_FILE      M_IMAGE_PATH MIL_TEXT("CircuitsBoard.mim")

/* Model position and size. */
#define FIND_MODEL_X_POS      153L
#define FIND_MODEL_Y_POS      132L
#define FIND_MODEL_WIDTH      128L
#define FIND_MODEL_HEIGHT     128L
#define FIND_MODEL_X_CENTER    (FIND_MODEL_X_POS+(FIND_MODEL_WIDTH -1)/2.0)
#define FIND_MODEL_Y_CENTER    (FIND_MODEL_Y_POS+(FIND_MODEL_HEIGHT-1)/2.0)

/* Target image shifting values. */
#define FIND_SHIFT_X          4.5
#define FIND_SHIFT_Y          7.5

```

```

/* Minimum match score to determine acceptability of model (default). */
#define FIND_MODEL_MIN_MATCH_SCORE 70.0

/* Minimum accuracy for the search. */
#define FIND_MODEL_MIN_ACCURACY 0.1

/* Absolute value macro. */
#define Absolute(x) (((x) < 0.0) ? -(x) : (x))

void SearchModelExample(MIL_ID MilSystem, MIL_ID MilDisplay)
{
    MIL_ID      MilImage,           /* Image buffer identifier. */
               MilOverlayImage,    /* Overlay image.           */
               Model,               /* Model identifier.        */
               Result;              /* Result identifier.       */
    MIL_DOUBLE  XOrg = 0.0, YOrg = 0.0; /* Original model position. */
    MIL_DOUBLE  x    = 0.0, y    = 0.0; /* Model position.          */
    MIL_DOUBLE  ErrX = 0.0, ErrY = 0.0; /* Model error position.    */
    MIL_DOUBLE  Score = 0.0;          /* Model correlation score.  */
    MIL_DOUBLE  Time  = 0.0;          /* Model search time.       */
    MIL_DOUBLE  AnnotationColor = M_COLOR_RED; /* Drawing color.          */

    /* Restore source image into an automatically allocated image buffer. */
    MbufRestore(FIND_IMAGE_FILE, MilSystem, &MilImage);

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

    /* Prepare for overlay annotations. */
    MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
    MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);
    MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

    /* Allocate a normalized grayscale model. */
    MpatAllocModel(MilSystem, MilImage, FIND_MODEL_X_POS, FIND_MODEL_Y_POS,
                   FIND_MODEL_WIDTH, FIND_MODEL_HEIGHT, M_NORMALIZED, &Model);

    /* Set the search accuracy to high. */
    MpatSetAccuracy(Model, M_HIGH);

    /* Set the search model speed to high. */
    MpatSetSpeed(Model, M_HIGH);

    /* Preprocess the model. */
    MpatPreprocModel(MilImage, Model, M_DEFAULT);

    /* Draw a box around the model in the model image. */
    MgraColor(M_DEFAULT, AnnotationColor);
    MpatDraw(M_DEFAULT, Model, MilOverlayImage, M_DRAW_BOX+M_DRAW_POSITION,
             M_DEFAULT, M_ORIGINAL);

    /* Pause to show the original image and model position. */

```

```

MosPrintf(MIL_TEXT("\nA %ldx%ld model was defined in the source image.\n"),
          FIND_MODEL_WIDTH, FIND_MODEL_HEIGHT);
MosPrintf(MIL_TEXT("It will be found in an image shifted ")
          MIL_TEXT("by %.2f in X and %.2f in Y.\n"), FIND_SHIFT_X, FIND_SHIFT_Y);
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Clear the overlay image. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

/* Translate the image on a subpixel level. */
MimTranslate(MilImage, MilImage, FIND_SHIFT_X, FIND_SHIFT_Y, M_DEFAULT);

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

/* Dummy first call for bench measure purpose only (bench stabilization,
   cache effect, etc...). This first call is NOT required by the application. */
MpatFindModel(MilImage, Model, Result);
MappTimer(M_TIMER_RESET+M_SYNCHRONOUS, M_NULL);

/* Find the model in the target buffer. */
MpatFindModel(MilImage, Model, Result);

/* Read the time spent in MpatFindModel. */
MappTimer(M_TIMER_READ+M_SYNCHRONOUS, &Time);

/* If one model was found above the acceptance threshold. */
if (MpatGetNumber(Result, M_NULL) == 1L)
{
    /* Read results and draw a box around the model occurrence. */
    MpatGetResult(Result, M_POSITION_X, &x);
    MpatGetResult(Result, M_POSITION_Y, &y);
    MpatGetResult(Result, M_SCORE, &Score);
    MgraColor(M_DEFAULT, AnnotationColor);
    MpatDraw(M_DEFAULT, Result, MilOverlayImage, M_DRAW_BOX+M_DRAW_POSITION,
             M_DEFAULT, M_DEFAULT);

    /* Calculate the position errors in X and Y and inquire original model position. */
    ErrX = fabs((FIND_MODEL_X_CENTER+FIND_SHIFT_X) - x);
    ErrY = fabs((FIND_MODEL_Y_CENTER+FIND_SHIFT_Y) - y);
    MpatInquire(Model, M_ORIGINAL_X, &XOrg);
    MpatInquire(Model, M_ORIGINAL_Y, &YOrg);

    /* Print out the search result of the model in the original image. */
    MosPrintf(MIL_TEXT("Search results:\n"));
    MosPrintf(MIL_TEXT("-----\n"));
    MosPrintf(MIL_TEXT("The model is found to be shifted by \tX:%.2f, Y:%.2f.\n"),
              x-XOrg, y-YOrg);
    MosPrintf(MIL_TEXT("The model position error is \t\tX:%.2f, Y:%.2f\n"),
              ErrX, ErrY);
    MosPrintf(MIL_TEXT("The model match score is \t\t%.1f\n"), Score);
    MosPrintf(MIL_TEXT("The search time is \t\t\t%.3f ms\n\n"), Time*1000.0);
}

```

```

    /* Verify the results. */
    if (
        (Absolute((x - XOrg) - FIND_SHIFT_X) > FIND_MODEL_MIN_ACCURACY) ||
        (Absolute((y - YOrg) - FIND_SHIFT_Y) > FIND_MODEL_MIN_ACCURACY) ||
        (Score < FIND_MODEL_MIN_MATCH_SCORE)
    )
        MosPrintf(MIL_TEXT("Results verification error !\n"));
    }
else
    MosPrintf(MIL_TEXT("Model not found !\n"));

/* Wait for a key to be pressed. */
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Clear the overlay image. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

/* Free all allocations. */
MpatFree(Result);
MpatFree(Model);
MbufFree(MilImage);
}

/*****
/* Find rotated model example. */

/* Source image file name. */
#define ROTATED_FIND_IMAGE_FILE      M_IMAGE_PATH MIL_TEXT("CircuitsBoard.mim")

/* Image rotation values. */
#define ROTATED_FIND_ROTATION_DELTA_ANGLE  10
#define ROTATED_FIND_ROTATION_ANGLE_STEP  1
#define ROTATED_FIND_RAD_PER_DEG          0.01745329251

/* Model position and size. */
#define ROTATED_FIND_MODEL_X_POS          153L
#define ROTATED_FIND_MODEL_Y_POS          132L
#define ROTATED_FIND_MODEL_WIDTH          128L
#define ROTATED_FIND_MODEL_HEIGHT         128L

#define ROTATED_FIND_MODEL_X_CENTER      ROTATED_FIND_MODEL_X_POS+ \
                                          (ROTATED_FIND_MODEL_WIDTH -1)/2.0
#define ROTATED_FIND_MODEL_Y_CENTER      ROTATED_FIND_MODEL_Y_POS+ \
                                          (ROTATED_FIND_MODEL_HEIGHT-1)/2.0

/* Minimum accuracy for the search position. */
#define ROTATED_FIND_MIN_POSITION_ACCURACY  0.10

/* Minimum accuracy for the search angle. */

```



```

#define ROTATED_FIND_MIN_ANGLE_ACCURACY          0.25

/* Angle range to search. */
#define ROTATED_FIND_ANGLE_DELTA_POS            ROTATED_FIND_ROTATION_DELTA_ANGLE
#define ROTATED_FIND_ANGLE_DELTA_NEG            ROTATED_FIND_ROTATION_DELTA_ANGLE

/* Prototypes of utility functions. */
void RotateModelCenter(MIL_ID Buffer,
                      MIL_DOUBLE *X,
                      MIL_DOUBLE *Y,
                      MIL_DOUBLE Angle);

MIL_DOUBLE CalculateAngleDist(MIL_DOUBLE Angle1,
                             MIL_DOUBLE Angle2);

void SearchRotatedModelExample(MIL_ID MilSystem, MIL_ID MilDisplay)
{
    MIL_ID      MilSourceImage,          /* Model image buffer identifier. */
    MilTargetImage,          /* Target image buffer identifier. */
    MilDisplayImage,          /* Target image buffer identifier. */
    MilOverlayImage,          /* Overlay image. */
    MilModel,                /* Model identifier. */
    MilResult;               /* Result identifier. */
    MIL_DOUBLE  RealX         = 0.0,      /* Model real position in x. */
    RealY         = 0.0,      /* Model real position in y. */
    RealAngle     = 0.0,      /* Model real angle. */
    X             = 0.0,      /* Model position in x found. */
    Y             = 0.0,      /* Model position in y found. */
    Angle         = 0.0,      /* Model angle found. */
    Score         = 0.0,      /* Model correlation score. */
    Time          = 0.0,      /* Model search time. */
    ErrX          = 0.0,      /* Model error position in x. */
    ErrY          = 0.0,      /* Model error position in y. */
    ErrAngle      = 0.0,      /* Model error angle. */
    SumErrX       = 0.0,      /* Model total error position in x. */
    SumErrY       = 0.0,      /* Model total error position in y. */
    SumErrAngle   = 0.0,      /* Model total error angle. */
    SumTime       = 0.0;      /* Model total search time. */
    MIL_INT      NbFound      = 0;        /* Number of models found. */
    MIL_DOUBLE   AnnotationColor = M_COLOR_RED; /* Drawing color. */

    /* Load target image into image buffers and display it. */
    MbufRestore(ROTATED_FIND_IMAGE_FILE, MilSystem, &MilSourceImage);
    MbufRestore(ROTATED_FIND_IMAGE_FILE, MilSystem, &MilTargetImage);
    MbufRestore(ROTATED_FIND_IMAGE_FILE, MilSystem, &MilDisplayImage);
    MdispSelect(MilDisplay, MilDisplayImage);

    /* Prepare for overlay annotations. */
    MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
    MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);
    MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

```

```

/* Allocate a normalized grayscale model. */
MpatAllocModel(MilSystem, MilSourceImage,
               ROTATED_FIND_MODEL_X_POS, ROTATED_FIND_MODEL_Y_POS,
               ROTATED_FIND_MODEL_WIDTH, ROTATED_FIND_MODEL_HEIGHT,
               M_NORMALIZED+M_CIRCULAR_OVERSCAN, &MilModel);

/* Set the search model speed. */
MpatSetSpeed(MilModel, M_MEDIUM);

/* Set the position search accuracy. */
MpatSetAccuracy(MilModel, M_HIGH);

/* Activate the search model angle mode. */
MpatSetAngle(MilModel, M_SEARCH_ANGLE_MODE, M_ENABLE);

/* Set the search model range angle. */
MpatSetAngle(MilModel, M_SEARCH_ANGLE_DELTA_NEG, ROTATED_FIND_ANGLE_DELTA_NEG);
MpatSetAngle(MilModel, M_SEARCH_ANGLE_DELTA_POS, ROTATED_FIND_ANGLE_DELTA_POS);

/* Set the search model angle accuracy. */
MpatSetAngle(MilModel, M_SEARCH_ANGLE_ACCURACY, ROTATED_FIND_MIN_ANGLE_ACCURACY);

/* Set the search model angle interpolation mode to bilinear. */
MpatSetAngle(MilModel, M_SEARCH_ANGLE_INTERPOLATION_MODE, M_BILINEAR);

/* Preprocess the model. */
MpatPreprocModel(MilSourceImage, MilModel, M_DEFAULT);

/* Allocate a result buffer. */
MpatAllocResult(MilSystem, 1L, &MilResult);

/* Draw the original model position */
MpatDraw(M_DEFAULT, MilModel, MilOverlayImage, M_DRAW_BOX+M_DRAW_POSITION,
         M_DEFAULT, M_ORIGINAL);

/* Pause to show the original image and model position. */
MosPrintf(MIL_TEXT("\nA %ldx%ld model was defined in the source image.\n"),
          ROTATED_FIND_MODEL_WIDTH, ROTATED_FIND_MODEL_HEIGHT);
MosPrintf(MIL_TEXT("It will be searched in images rotated from %ld degree ")
          MIL_TEXT("to %ld degree.\n"), -ROTATED_FIND_ROTATION_DELTA_ANGLE,
          ROTATED_FIND_ROTATION_DELTA_ANGLE);
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Dummy first call for bench measure purpose only (bench stabilization,
cache effect, etc...). This first call is NOT required by the application. */
MpatFindModel(MilSourceImage, MilModel, MilResult);

/* If the model was found above the acceptance threshold. */
if (MpatGetNumber(MilResult, M_NULL) == 1L)
{
    /* Search for the model in images at different angles. */
    RealAngle = ROTATED_FIND_ROTATION_DELTA_ANGLE;

```

```

while (RealAngle >= -ROTATED_FIND_ROTATION_DELTA_ANGLE)
{
    /* Rotate the image from the model image to target image. */
    MimRotate(MilSourceImage, MilTargetImage, RealAngle, M_DEFAULT,
              M_DEFAULT, M_DEFAULT, M_DEFAULT, M_BILINEAR+M_OVERSCAN_CLEAR);

    /* Reset the timer. */
    MapTimer(M_TIMER_RESET+M_SYNCHRONOUS, M_NULL);

    /* Find the model in the target image. */
    MpatFindModel(MilTargetImage, MilModel, MilResult);

    /* Read the time spent in MpatFindModel(). */
    MapTimer(M_TIMER_READ+M_SYNCHRONOUS, &Time);

    /* Clear the overlay image. */
    MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

    /* If one model was found above the acceptance threshold. */
    if (MpatGetNumber(MilResult, M_NULL) == 1L)
    {
        /* Read results and draw a box around model occurrence. */
        MpatGetResult(MilResult, M_POSITION_X, &X);
        MpatGetResult(MilResult, M_POSITION_Y, &Y);
        MpatGetResult(MilResult, M_ANGLE, &Angle);
        MpatGetResult(MilResult, M_SCORE, &Score);

        MgraColor(M_DEFAULT, AnnotationColor);
        MpatDraw(M_DEFAULT, MilResult, MilOverlayImage,
                 M_DRAW_BOX+M_DRAW_POSITION, M_DEFAULT, M_DEFAULT);

        MbufCopy(MilTargetImage, MilDisplayImage);

        /* Calculate the angle error and the position errors for statistics. */
        ErrAngle = CalculateAngleDist(Angle, RealAngle);

        RotateModelCenter(MilSourceImage, &RealX, &RealY, RealAngle);
        ErrX = fabs(X - RealX);
        ErrY = fabs(Y - RealY);

        SumErrAngle += ErrAngle;
        SumErrX += ErrX;
        SumErrY += ErrY;
        SumTime += Time;
        NbFound++;

        /* Verify the precision for the position and the angle. */
        if ((ErrX > ROTATED_FIND_MIN_POSITION_ACCURACY) ||
            (ErrY > ROTATED_FIND_MIN_POSITION_ACCURACY) ||
            (ErrAngle > ROTATED_FIND_MIN_ANGLE_ACCURACY))
        {
            MosPrintf(MIL_TEXT("Model accuracy error at angle %.1f !\n\n"), RealAngle);
            MosPrintf(MIL_TEXT("Errors are X:%.3f, Y:%.3f and Angle:%.2f\n\n"),

```

```

        ErrX, ErrY, ErrAngle);
    MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
    MosGetch();
}
else
{
    MosPrintf(MIL_TEXT("Model was not found at angle %.1f !\n\n"), RealAngle);
    MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
    MosGetch();
}

RealAngle -= ROTATED_FIND_ROTATION_ANGLE_STEP;
}

/* Print out the search result statistics */
/* of the models found in rotated images. */
MosPrintf(MIL_TEXT("\nSearch statistics for the model ")
    MIL_TEXT("found in the rotated images.\n"));
MosPrintf(MIL_TEXT("-----")
    MIL_TEXT("-----\n"));
MosPrintf(MIL_TEXT("The average position error is \t\tX:%.3f, Y:%.3f\n"),
    SumErrX/NbFound, SumErrY/NbFound);
MosPrintf(MIL_TEXT("The average angle error is \t\t%.3f\n"), SumErrAngle/NbFound);
MosPrintf(MIL_TEXT("The average search time is \t\t%.3f ms\n\n"),
    SumTime*1000.0/NbFound);
}
else
{
    MosPrintf(MIL_TEXT("Model was not found!\n\n"));
}

/* Wait for a key to be pressed. */
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Clear the overlay image. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

/* Free all allocations. */
MpatFree(MilResult);
MpatFree(MilModel);
MbufFree(MilTargetImage);
MbufFree(MilSourceImage);
MbufFree(MilDisplayImage);
}

/* Calculate the rotated center of the model to compare the accuracy with
 * the center of the occurrence found during pattern matching.
 */
void RotateModelCenter(MIL_ID Buffer,
    MIL_DOUBLE *X,

```

```

        MIL_DOUBLE *Y,
        MIL_DOUBLE Angle)
{
    MIL_INT BufSizeX = MbufInquire(Buffer, M_SIZE_X, M_NULL);
    MIL_INT BufSizeY = MbufInquire(Buffer, M_SIZE_Y, M_NULL);
    MIL_DOUBLE RadAngle = Angle * ROTATED_FIND_RAD_PER_DEG;
    MIL_DOUBLE CosAngle = cos(RadAngle);
    MIL_DOUBLE SinAngle = sin(RadAngle);

    MIL_DOUBLE OffSetX = (BufSizeX-1)/2.0F;
    MIL_DOUBLE OffSetY = (BufSizeY-1)/2.0F;

    *X = (ROTATED_FIND_MODEL_X_CENTER-OffsetX)*CosAngle +
        (ROTATED_FIND_MODEL_Y_CENTER-OffsetY)*SinAngle + OffSetX;
    *Y = (ROTATED_FIND_MODEL_Y_CENTER-OffsetY)*CosAngle -
        (ROTATED_FIND_MODEL_X_CENTER-OffsetX)*SinAngle + OffSetY;
}

/* Calculate the absolute difference between the real angle
 * and the angle found.
 */
MIL_DOUBLE CalculateAngleDist(MIL_DOUBLE Angle1, MIL_DOUBLE Angle2)
{
    MIL_DOUBLE dist = fabs(Angle1 - Angle2);

    while(dist >= 360.0)
        dist -= 360.0;

    if(dist > 180.0)
        dist = 360.0 - dist;

    return dist;
}

/*****
 * Automatic model allocation example. */

/* Source and target images file specifications. */
#define AUTO_MODEL_IMAGE_FILE          M_IMAGE_PATH MIL_TEXT("Wafer.mim")
#define AUTO_MODEL_TARGET_IMAGE_FILE   M_IMAGE_PATH MIL_TEXT("WaferShifted.mim")
// #define AUTO_MODEL_IMAGE_FILE        M_IMAGE_PATH MIL_TEXT("CircuitsBoard.mim")
// #define AUTO_MODEL_TARGET_IMAGE_FILE  M_IMAGE_PATH MIL_TEXT("CircuitsBoard.mim")

/* Model width and height */
#define AUTO_MODEL_WIDTH    64L
#define AUTO_MODEL_HEIGHT   64L

void AutoAllocationModelExample(MIL_ID MilSystem, MIL_ID MilDisplay)
{
    MIL_ID      MilImage,                /* Image buffer identifier. */
    MilOverlayImage,                    /* Overlay image. */

```

```

        MilSubImage,                /* Sub-image buffer identifier. */
        MilOverlaySubImage,         /* Overlay sub-image buffer identifier. */
        Model,                      /* Model identifier. */
        Result;                     /* Result buffer identifier. */
MIL_INT AllocError;                /* Allocation error variable. */
MIL_INT ImageWidth, ImageHeight;   /* Target image dimensions */
MIL_DOUBLE OrgX=0.0, OrgY=0.0;     /* Original center of model. */
MIL_DOUBLE x=0.0, y=0.0, Score=0.0; /* Result variables. */
MIL_DOUBLE AnnotationColor = M_COLOR_RED; /* Drawing color. */

/* Load model image into an image buffer. */
MbufRestore(AUTO_MODEL_IMAGE_FILE, MilSystem, &MilImage);

/* Display the image and prepare for overlay annotations. */
MdispSelect(MilDisplay, MilImage);
MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);
MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

/* Restrict the region to be processed to the bottom right corner of the image. */
MbufInquire(MilImage, M_SIZE_X, &ImageWidth);
MbufInquire(MilImage, M_SIZE_Y, &ImageHeight);
MbufChild2d(MilImage, ImageWidth/2, ImageHeight/2,
            ImageWidth/2, ImageHeight/2, &MilSubImage);
MbufChild2d(MilOverlayImage, ImageWidth/2, ImageHeight/2,
            ImageWidth/2, ImageHeight/2, &MilOverlaySubImage);

/* Automatically allocate a normalized grayscale type model. */
MpatAllocAutoModel(MilSystem, MilSubImage, AUTO_MODEL_WIDTH, AUTO_MODEL_HEIGHT,
                  M_DEFAULT, M_DEFAULT, M_NORMALIZED, M_DEFAULT, &Model);

/* Set the search accuracy to high. */
MpatSetAccuracy(Model, M_HIGH);

/* Check for that model allocation was successful. */
MappGetError(M_CURRENT, &AllocError);
if (!AllocError)
{
    /* Draw a box around the model. */
    MgraColor(M_DEFAULT, AnnotationColor);
    MpatDraw(M_DEFAULT, Model, MilOverlaySubImage, M_DRAW_BOX+M_DRAW_POSITION,
            M_DEFAULT, M_ORIGINAL);
    MosPrintf(MIL_TEXT("A model was automatically defined in the image.\n"));
    MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
    MosGetch();

    /* Clear the overlay image. */
    MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

    /* Load target image into an image buffer. */
    MbufLoad(AUTO_MODEL_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);

    /* Draw a box around the occurrence. */
    MgraColor(M_DEFAULT, AnnotationColor);
    MpatDraw(M_DEFAULT, Result, MilOverlaySubImage, M_DRAW_BOX+M_DRAW_POSITION,
                                                    M_DEFAULT, M_DEFAULT);

    /* Analyze and print results. */
    MpatInquire(Model, M_ORIGINAL_X, &OrgX);
    MpatInquire(Model, M_ORIGINAL_Y, &OrgY);
    MosPrintf(MIL_TEXT("An image misaligned by 50 pixels in X and 20 pixels ")
                                                    MIL_TEXT("in Y was loaded.\n\n"));
    MosPrintf(MIL_TEXT("The image is found to be shifted by %.2f in X, ")
                                                    MIL_TEXT("and %.2f in Y.\n"), x-OrgX, y-OrgY);
    MosPrintf(MIL_TEXT("Model match score is %.1f percent.\n"), Score);
    MosPrintf(MIL_TEXT("Press <Enter> to end.\n\n"));
    MosGetch();
}
else
{
    MosPrintf(MIL_TEXT("Error: Pattern not found properly.\n"));
    MosPrintf(MIL_TEXT("Press <Enter> to end.\n\n"));
    MosGetch();
}

/* Free result buffer and model. */
MpatFree(Result);
MpatFree(Model);
}
else
{
    MosPrintf(MIL_TEXT("Error: Automatic model definition failed.\n"));
    MosPrintf(MIL_TEXT("Press <Enter> to end.\n\n"));
    MosGetch();
}

```

```
    }

    /* Clear the overlay image. */
    MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

    /* Free child buffer and defaults. */
    MbufFree(MilSubImage);
    MbufFree(MilOverlaySubImage);
    MbufFree(MilImage);
}
```


Chapter

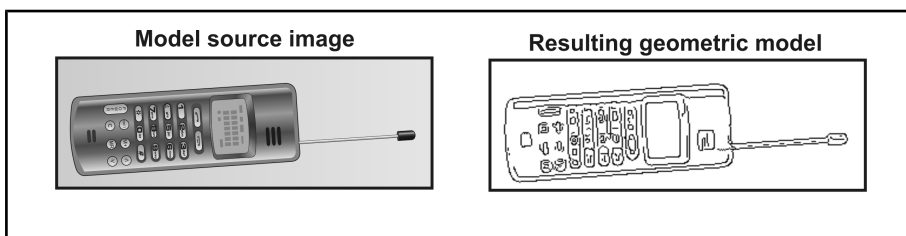
8

Geometric Model Finder

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

Geometric Model Finder module

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



The 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 scales. You can also search for different kinds of models with Model Finder including models created from images (image-type), models created from Edge Finder result buffers (Edge Finder-type), models created from Model Finder result buffers (Model Finder-type), models merged from two previous models (merge-type), and synthetic models. Synthetic models are either models that have a predefined shape, or models defined from CAD files.

The module provides complete support for calibration. Searches can be performed in the calibrated real-world such that, without physically correcting your images, occurrences can be found even in the presence of complex distortions, and results calculated in real-world units.

The module also allows you to restore a Model Finder context from a file or memory stream, or save a Model Finder context to a file or memory stream.

Steps to performing a model search

The following steps provide a basic methodology for using the 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()** or **MmodDefineFromFile()**.
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 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()**.

For information about using calibrated model source images and targets, see the *Calibration* section later in this chapter.

Basic concepts

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

- **Active edges.** Edges which are used to compose the geometric model without being masked out, and are searched for in the target.
- **Bounding box of the edges.** The smallest rectangle that fully encloses all of the edges of the model.
- **Chain.** The set of connected edgels that construct an edge.
- **Edge.** A curve that delineates a boundary, which can be established from intensity transitions in an image. In Model Finder, an edge is considered to be a chain, and its features.
- **Edge map.** The edges extracted from the model image.
- **Edgel.** Elementary point (or edge element) within an edge.
- **Model.** The information that defines the pattern of active edges to find in the target and the search settings with which to do so.
- **Model box.** The box that delimits the boundaries of the model. Essentially, the model box is the bounding box of the edges plus a margin. For image-type and Edge Finder-type models, the size of the model box is specified when defining the model. For synthetic models, the margin can be adjusted after model definition.
- **Model Finder context.** The container for all models that you want to find. The Model Finder context allows you to set global search settings that apply to the search algorithm.
- **Model image.** For an image-type and Edge Finder-type model, the model image is a copy of the region in the model source image from which the model has been defined. Note that the model source image for an Edge Finder-type model is the image source of the result buffer. For a synthetic model, the model image is the image representation (edge map) of the model.

- **Model mask.** A binary image used to define irrelevant, inconsistent, or featureless areas in the model, so that only pertinent model details are used for the search.
- **Model origin.** The point in the model's coordinate system that is considered to be (0, 0).
- **Model source.** The immediate source of the model's edges. For example, for an Edge Finder-type model, the model source is the buffer of the Edge Finder results.
- **Model source image.** The image from which you have defined an image-type model, or from which you have extracted the edges for an Edge Finder-type model.
- **Occurrence.** An instance of the model found in the target.
- **Score.** A measure of the active edges in the model found in the occurrence, weighted by the deviation in position of these common edges.
- **Synthetic models.** Models that have a predefined shape, such as a circle or a square, or are defined from a CAD-type file.
- **Target.** The image or result buffer in which to search for occurrences of the model.
- **Target score.** A measure of edges found in the occurrence that are not present in the original model, weighted by the deviation in position of the common edges.
- **Target edges.** The edges in the target.

Types of Model Finder contexts

Using **MmodAlloc()**, you can allocate one of two types of Model Finder contexts: one that uses a general geometric search algorithm (**M_GEOMETRIC**) or one that uses a controlled geometric search algorithm (**M_GEOMETRIC_CONTROLLED**). Both algorithms use edge-based geometric features of the models and the target to establish a match.

General geometric contexts

General geometric Model Finder contexts use a general geometric search algorithm to locate user-specified models. General geometric contexts handle a full range of scale and angles unlike a controlled geometric context.

Controlled geometric contexts

Controlled geometric Model Finder contexts use a controlled geometric search algorithm to locate user-specified models. Using a controlled geometric context is recommended in situations where the image has high geometric complexity and where there are only small differences scale between the occurrence and the nominal scale of the model (**M_SCALE**). In such a situation, a controlled geometric search is often faster and more robust than the general geometric search. It is especially fast when there is a difference in angle between the occurrence and the nominal angle of the model (**M_ANGLE**). Model searches use a hierarchical method to reduce the time needed to complete the search. Note that this might cause very thin lines to be missed as candidates.

The speed of the search with a controlled geometric Model Finder context also depends on the angle range of the search. The search is faster when using a smaller angle range.

There are restrictions on when you can use a controlled geometric context. Scale range searches are disabled when you use this context. As well, if you are retrieving a result relating to the entire context, you cannot get or draw the edges for the entire target; you can only get or draw the target edges in the region of an occurrence.

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.

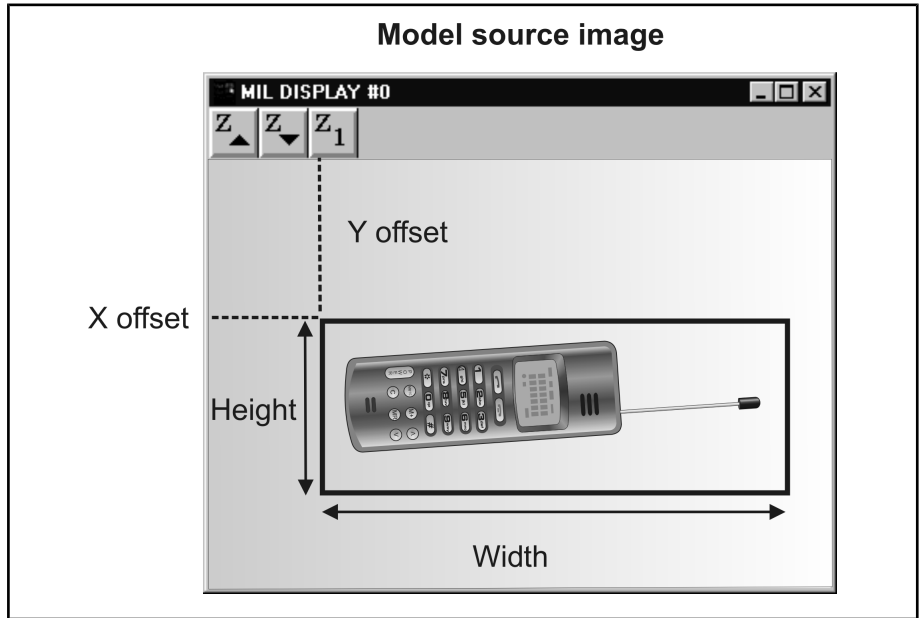
There are four different kinds of models that can be added to a Model Finder context: a model from an image, a synthetic model, a model from an Edge Finder or Model Finder result buffer, or a model created from two other models. All of these models can be mixed in the same Model Finder context, using **MmodDefine()** or **MmodDefineFromFile()**. The **MmodDefine()** function can add image-type models, result-type models, and predefined-shape synthetic models; the **MmodDefineFromFile()** function can add CAD-file synthetic models.

Image-type models

You can add models defined from an image to a Model Finder context manually or automatically, using **MmodDefine()** with **M_IMAGE** or **M_AUTO_DEFINE**, respectively. These types of models are referred to as **image-type** models.

The specified model source image must be a 1-band, 8-bit unsigned image.

When defining a model manually, you must specify the region in the image from which to define the model. The minimum size of a model is limited to 16.0 X 16.0 pixels, while the maximum size of 1024.0 X 1024.0 pixels is permitted.



When defining a model automatically, MIL searches the model source image for unique geometric features. When the most suitable geometric features are found, the function automatically defines a model from those features. To be useful, the model source image should be a typical target image; otherwise, the selected area for the model might never be present in the target. Defining a model automatically is useful, for example, when you want to perform whole image alignment, for which the displacement of a unique model from its original location specifies the displacement of the image. MIL can automatically define a model based on specified settings or on default settings. The former is useful if you know, for example, that the selected model needs to be non-ambiguous only in a small range of angles; its ambiguity needs not be checked for the entire default range. Note that MIL searches for the unique model far enough from the image borders so that if the target image is shifted from the model source image, the model can still be found; you specify the maximum displacement that can occur between the position of the model in the source image and the position of the model when found in the target image.

To define a model based on specified settings:

1. Define an empty image-type model, using **MmodDefine()** with **M_AUTO_DEFINE** set to **M_NULL**.
2. Set the model's settings, using **MmodControl()** with the index of the empty model.
3. Define a unique geometric model from the model source image using **MmodControl()** with **M_AUTO_DEFINE** and the identifier of the source image. **MmodControl()** searches for a model with unique geometric features in the model source image that takes into account the settings previously specified using **MmodControl()**.

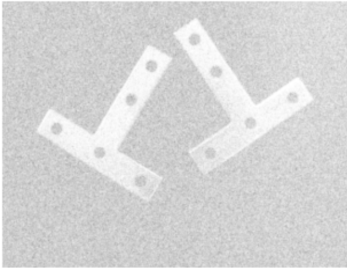
To define a model based on default settings, use **MmodDefine()** with **M_AUTO_DEFINE** and the identifier of the source image.

Extracting edges

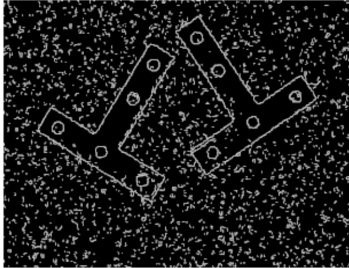
When you call **MmodDefine()**, it stores the specified region of the model source image with the model. This image is referred to as the model image. The model source image is then no longer needed.

It is actually when you call **MmodPreprocess()** that edges are extracted from the model image. Specifically, **MmodPreprocess()** extracts the contour edges from the model image. For more information on contour edges, see the *Extracting the edges* section in *Chapter 9: Edge Finder*.

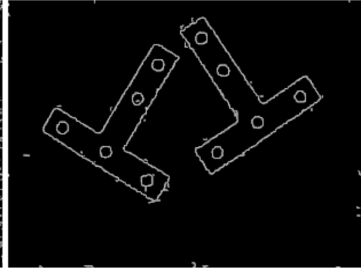
The edge extraction process involves a denoising operation to even out rough edges and remove noise. You can control the degree of smoothness (strength) of the denoising operation used for all models in the context, using **MmodControl()** with **M_SMOOTHNESS**. The range of this control type varies from 0.0 to 100.0; a value of 100.0 results in a strong noise reduction effect, while a value of 0.0 has almost no noise reduction effect. The default setting is 50.0.



Target image with considerable noise



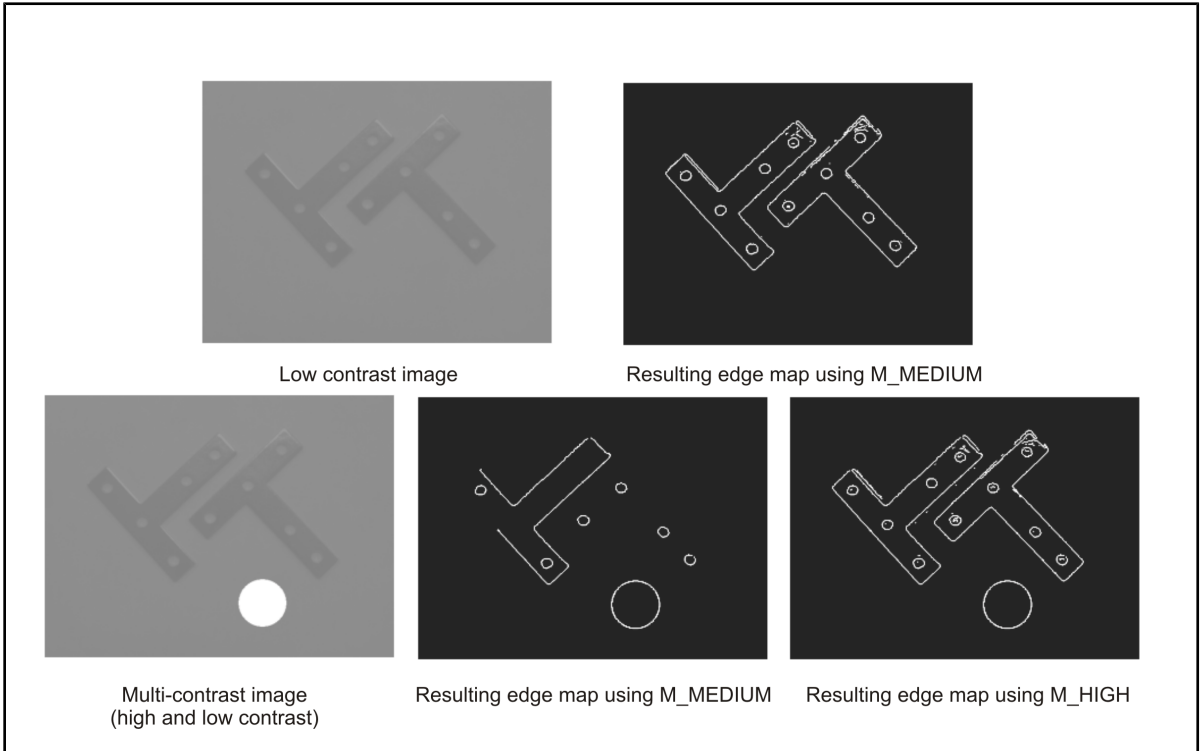
Edge map obtained with M_SMOOTHNESS set to 50



Edge map obtained with M_SMOOTHNESS set to 70

You can also control the detail level of the edge extraction process for all models in the context, using **MmodControl()** with **M_DETAIL_LEVEL**. Basically, this controls how much detail is extracted from the model image of each model in the context. The default setting (**M_MEDIUM**) offers a robust detection of 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 type.



If your images contain low-contrast and high-contrast objects, and your object of interest is low-contrast, a detail level setting of **M_HIGH** should be used to ensure the detection of the low-contrast 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 setting is very sensitive to noise.

Generally, the default settings for **M_SMOOTHNESS** and **M_DETAIL_LEVEL** are sufficient for the majority of images; you should adjust these settings only when dealing with very noisy images, extremely low or multi-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.

Note that the settings for **M_SMOOTHNESS** and **M_DETAIL_LEVEL** are also applied to the search target when it is an image.

Synthetic models

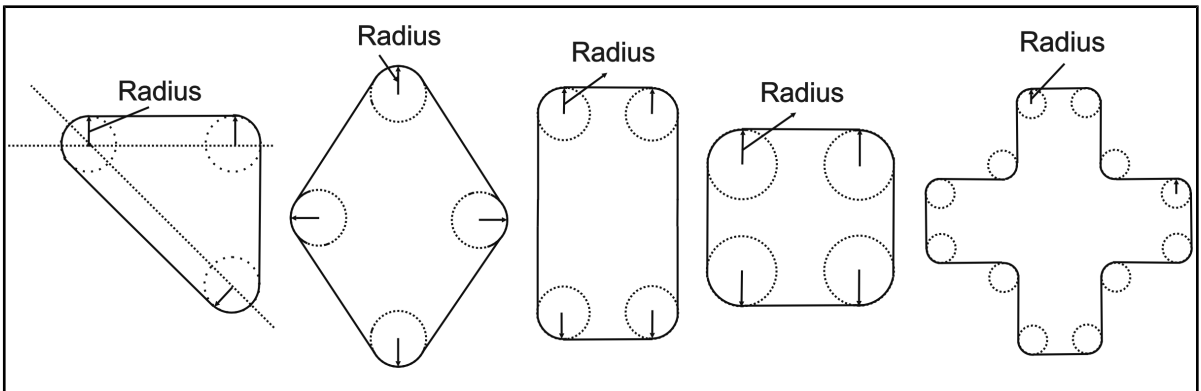
Synthetic models are models that either have a predefined shape, or have been defined from a CAD DXF file. Synthetic models allow you to search target images and Edge Finder result buffers for shapes that you expect to be in the target.

Predefined-shape models

You can use **MmodDefine()** to add predefined-shape models to the context. The model types that are available are circle, cross, diamond, ellipse, rectangle, ring, square, and triangle. These model types are described in more detail in the description of **MmodDefine()**.

Use the **MmodDefine()** parameters to define the attributes of the predefined-shape model, from the foreground color of the model to how it is constructed. The definition of each **MmodDefine()** parameter depends on the model. For example, for a circle model, **MmodDefine()** would have a parameter that defines the radius of the circle. For an ellipse model, **MmodDefine()** would have parameters that define the width and the height.

You can also round all the corners of a model that has corners (the cross, diamond, rectangle, square, and triangle models). Set the radius of the curvature for the model's corners using **MmodControl()** with **M_CORNER_RADIUS**.



Predefined-shape models use the center of the shape as the origin (0,0).

CAD-file models

You can add models that are defined from CAD DXF files to a Model Finder context, using **MmodDefineFromFile()**.

Not every entity in a CAD DXF file is supported; unsupported entities are ignored. The supported entities are Line, Polyline, Lwpolyline, Circle, Arc, Ellipse, Block (a collection of entities), and Insert (an entity containing a reference to where the Block should be drawn, and at what location and scale).

When the model is added to the Model Finder context from a CAD DXF file, the model will retain its coordinate system (the origin and the axis). Most CAD DXF files are oriented so that the Y-axis is positive going up, whereas the coordinate system MIL uses is oriented so that the Y-axis is positive going down. So if you take the edge coordinates of an object from a CAD DXF file and put them in an image, the imaged object will look flipped when compared to the original object. This is also the case for a model defined from a CAD DXF file. This means that, unless the object is symmetrical, the match will not be made. You can use **MmodControl()** with **M_CAD_Y_AXIS** to flip the Y-axis for the model so that it is positive going down.

Model size and units

The dimensions of a synthetic model should be specified in user-defined units. Specify the ratio between model units and pixel units using **MmodControl()** with **M_PIXEL_SCALE**.

By default, synthetic models are defined to have a 10% margin around the bounding box of their edges. When finding an occurrence, extra edges found in this area will reduce the target score. You can change the size of the margin using **MmodControl()** with the **M_BOX_MARGIN_...** control types.

Synthetic models and calibration

You can search for synthetic models in a calibrated or non-calibrated target. If the target is not calibrated, then **M_PIXEL_SCALE** is used for the match.

If the target is calibrated, the model units should be the same as the calibrated units. If the target is calibrated, **M_PIXEL_SCALE** will be used for draw operations, but not for the match nor the returned results.

Note that if you are searching for a synthetic model in a calibrated target, you must also associate the calibration object of the target with the model, using **M_ASSOCIATED_CALIBRATION**. MIL needs the calibration object for internal purposes at preprocessing time. Note that MIL will not use the calibration object to compensate for incongruencies between the model and the target, nor will MIL map the model's coordinate system to that of the target.

- ❖ After you have specified your pixel scale and scale settings, you can verify if your synthetic models are valid, using **MmodInquire()** with **M_VALID**. Synthetic models can be invalid if their global size in X or Y (*size of the model box* x **M_PIXEL_SCALE** x **M_SCALE**) is greater than 1024.

Models defined from result buffers

You can define models from two different types of result buffers: Edge Finder result buffers and Model Finder result buffers.

In the case of these models, the **model source image** is the image from which the results were obtained. The **model source** is the result buffer in which the results are stored. The **model image** is a copy of the region in the model source image from which the model has been defined.

Models defined from an Edge Finder result buffer

You would define models from an Edge Finder result buffer if you want to use other types of edges, such as crest edges, or if you want to re-use the edge map for another purpose. If you knew that the edges of interest had a minimum size, you could use Edge Finder to include only edges above that size. The included edges of the results are the ones that will be used to define the model. These result-type models are referred to as **Edge Finder-type** models.

Some things have to be done before you can define a model from an Edge Finder result buffer.

- The Edge Finder result buffer must have been previously allocated using **MedgeAllocResult()**.
- The Edge Finder result buffer must be compatible with the Model Finder context. Enable this using **MedgeControl()** with **M_MODEL_FINDER_COMPATIBLE** prior to calling **MedgeCalculate()**.

- You can speed up the time it takes to search for the models by setting **M_EXTRACTION_SCALE** in **MedgeControl()** prior to calling **MedgeCalculate()**. For more information, see the *Interfacing with the Geometric Model Finder module* section in *Chapter 9: Edge Finder*.
- **MedgeCalculate()** must have been already called using this result buffer.

You can add models defined from an Edge Finder result buffer to a Model Finder context using **MmodDefine()** with **M_EDGE_RESULT**.

The size of the model can be defined using the offset and size parameters of **MmodDefine()**.

Models defined from a Model Finder result buffer

Adding models defined from a Model Finder result buffer allows you to define new models from previous match occurrences. These models are referred to as **Model Finder-type** models.

There are many reasons to use a Model Finder-type model. For example, if you have a model that changes slightly over time, you can define a model from the result to update it. Updating the model prevents potentially misidentifying the model because it has changed beyond your starting tolerances.

To define a Model Finder-type model, use **MmodDefine()** with **M_MOD_RESULT** and pass the identifier of the Model Finder result buffer and the index of the occurrence to use. The model is defined with the edges of the target at the occurrence position. You have to use **MmodControl()** with **M_MOD_DEFINE_COMPATIBLE** enabled before you can use this feature; this must be done before **MmodFind()** finds the result. The result buffer must be compatible with the Model Finder context.

Models defined from two other models

Models can be defined from two previously existing models and added to the context. Use **MmodDefine()** with **M_MERGE_MODEL** and specify the two models from which to create the new model. The model will be formed by taking the first model specified and adding the edges of the second model that are not included in the edges of the first model.

There are no restrictions on the types of models you can merge. They can be of different types, as long as they are part of the same model context.

Preprocessing

After you add models to your context and before performing the search, you have to preprocess the context, using **MmodPreprocess()**, otherwise an error will occur. It does not matter the type of model, or the type of context. Calling **MmodPreprocess()** will prepare models for geometric matching. In the case of image-type models, it is also the time when edges are actually extracted from the model image.

Note that the Model Finder context must be preprocessed again if some of the models' search settings are changed. You can check if you need to preprocess after changing a control using **MmodInquire()** with **M_PREPROCESSED**.

When you save a Model Finder context, preprocessing information is not saved with the context. You must preprocess the context when the Model Finder context is restored.

Model origin

The model origin is the point in the model that is considered to be (0,0). The location of the model origin depends on the type of model: for image and Edge Finder-type models, the model origin is the center of the upper left pixel of the model image; for predefined-shape models, the center of the shape is the model origin. The model origin for a model defined from a CAD DXF file is the same as point (0,0) in the CAD DXF file.

Model indices and labels

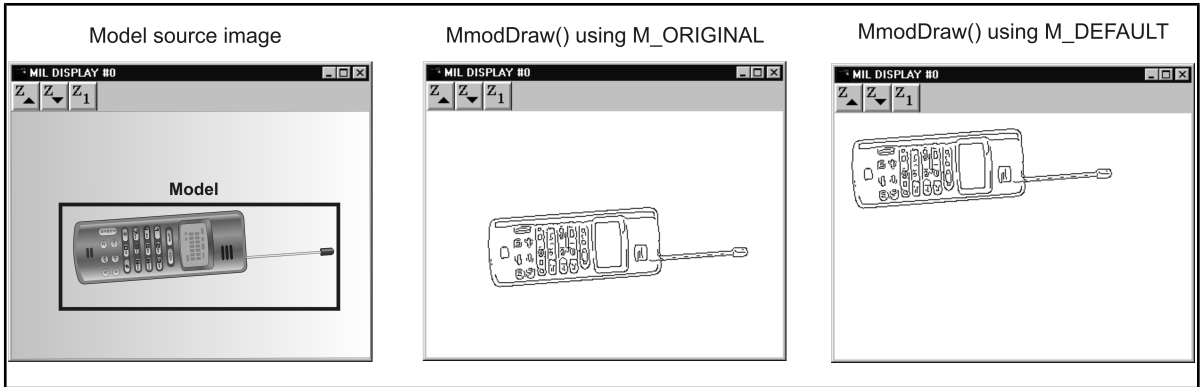
Each model added to a Model Finder context can be accessed by its index. The first model is assigned an index of 0, and for each subsequent model added, the index is increased by one. You can also apply setting changes to all models within a Model Finder context by using **M_ALL** as the index. When a model is deleted, all model indices greater than the deleted model are shifted down by 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 and inquiring the model's active edges

When testing different model source images for the best model candidate, you can use **MmodDraw()** to draw the model's active edges. This can be done for any type of model, although for synthetic models it is less useful, since the edges should already be known. The resulting edge map can reveal whether the edges you have chosen are the ones that you want, or if there are problems with the edges. You can use **MmodInquire()** with the **M_ALLOC_SIZE...** inquire types to determine the necessary size for the destination image buffer. To make corrections for image-type or Model Finder-type models, you might have to adjust the image processing control types, or mask unwanted edges. For Edge Finder-type models, you can exclude edges that you don't want from the Edge Finder result buffer and redefine the model, or you can mask unwanted edges.

If you have used offsets to define the model region in the model source, 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. This is valid for image-type or Edge Finder-type models.

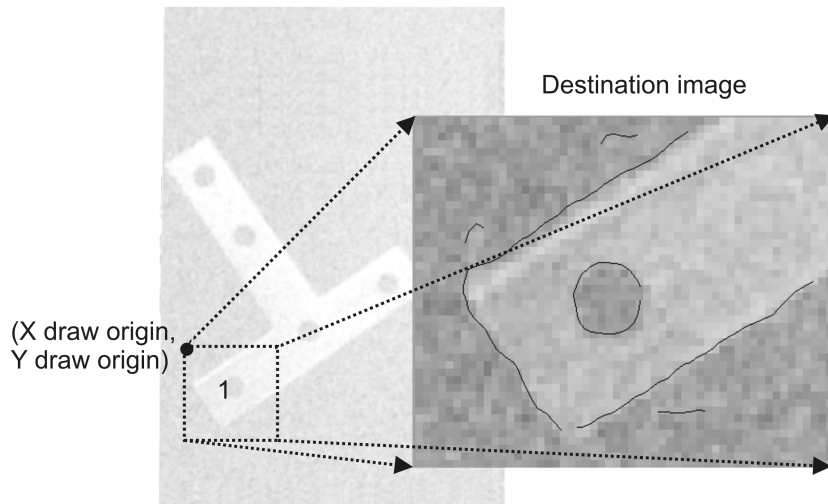


You can use **MmodInquire()** with the **M_CHAIN_INDEX**, **M_CHAIN_X**, and **M_CHAIN_Y** controls to inquire the active edges of the model.

Note that if you try to draw a model's edges and its Model Finder context has not been preprocessed, a preliminary edge extraction operation will be performed for the model.

You can also draw features from a zoomed region of the model image, as well as other items. To do so, specify the appropriate values for the **MmodControl()** **M_DRAW_RELATIVE_ORIGIN_X**, **M_DRAW_RELATIVE_ORIGIN_Y**, **M_DRAW_SCALE_X**, and **M_DRAW_SCALE_Y** control types. The relative origin values must be specified in pixels, and are relative to the coordinates of the top-left corner of the region in the model image, while the scale values specify the X- and Y-scaling factors used to fill the destination image buffer.

Drawing a zoomed region of the model image



- ¹ How much of this is drawn in the destination image is determined by the scale factor and the size of the destination image.

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.

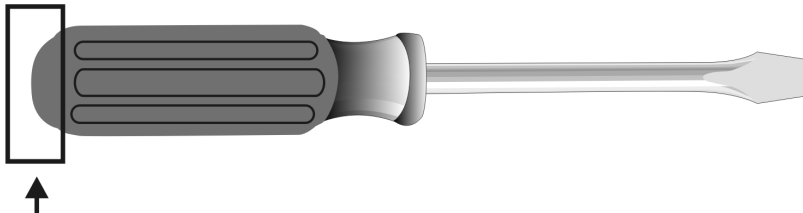
Make sure your images have enough contrast

Contrast is necessary for identifying edges in your model source and target image with subpixel accuracy. For this reason, it is recommended that you avoid models that contain only slow gradations in grayscale values.

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.

Poor geometric model

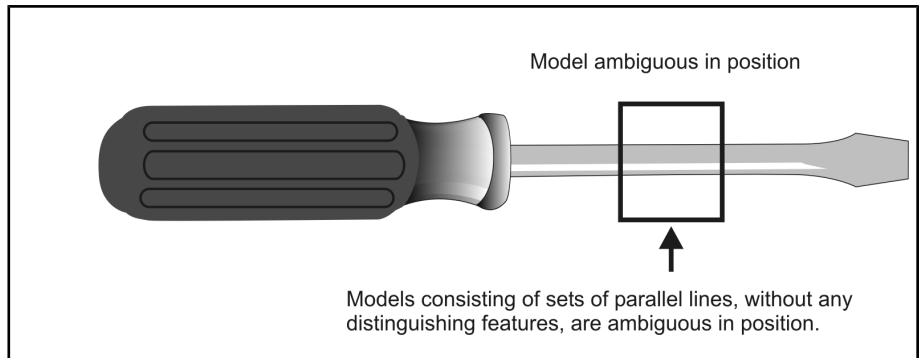


Simple curves lack distinguishing features and can produce false matches.

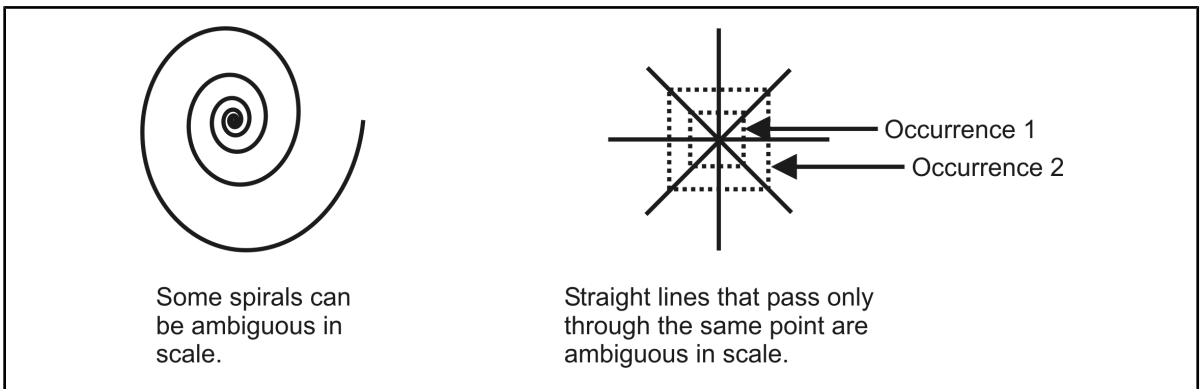
Be aware of ambiguous models

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

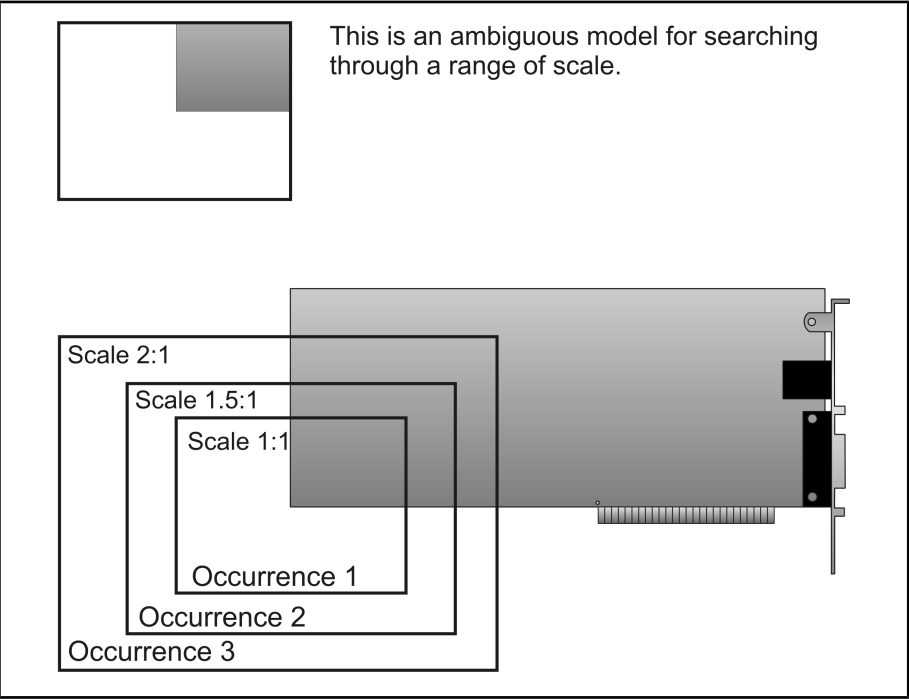
Models which are ambiguous in position are usually composed of one or more sets of parallel lines only. Such models make it impossible to establish a unique position for them. A large number of matches can be found since the actual number of line segments in any particular line is theoretically limitless.



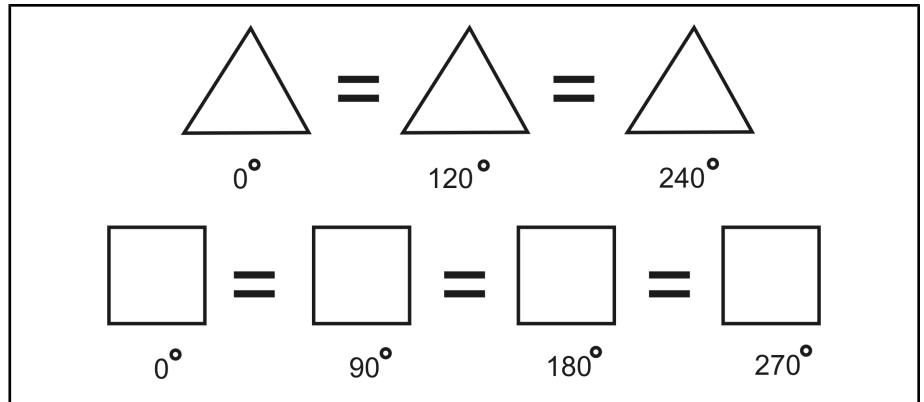
Models which are ambiguous in scale are usually composed only of straight lines which pass through the same point; some spirals are also ambiguous in scale. Models which consist of small portions of objects should be tested to verify that they are not ambiguous in scale.



For example, a model of an isolated corner is ambiguous in terms of scale because it consists of only two straight lines that pass through the same point.

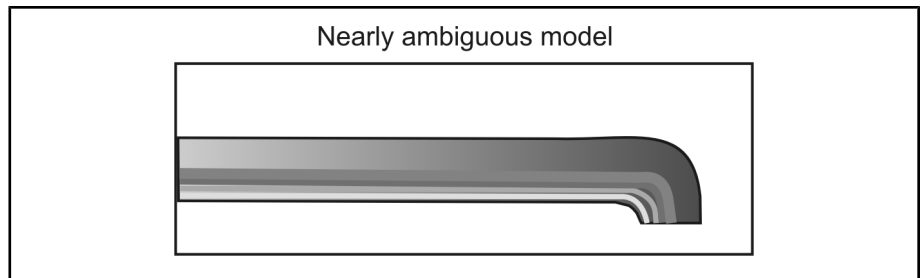


Symmetric models are often ambiguous in angle due to their similarity in features. For example, circles are completely ambiguous in terms of angle. Other simple symmetric models, such as squares and triangles, are ambiguous with respect to certain angles:



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 features. This will ensure that only correct matches are returned as results. For example, the following model can produce false matches since the greater proportion of active edges in the model is composed of parallel straight lines rather than distinguishing curves.



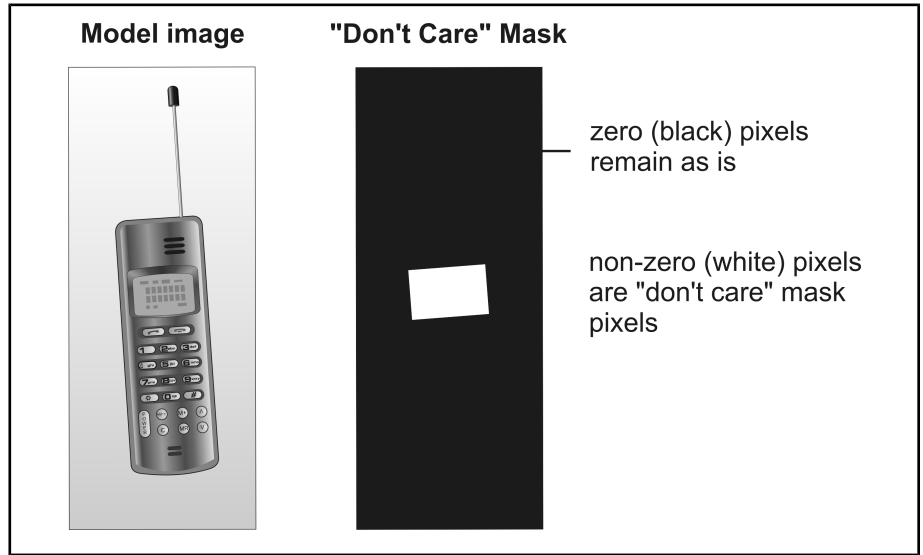
Masking your model

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()**.

You can define three types of masks: a *"don't care"*, a *"flat region"*, and a *"weighted region"* mask. You can use these types of masks with the same model. Note that *"don't care"* and *"flat region"* masks are not supported for synthetic models.

With a *"don't care"* mask (**M_DONT_CARES**), MIL ignores the masked regions when searching for occurrences of the model. Masked edges in the model edge map and edges in the corresponding regions of the target edge map will be ignored and will not contribute to either the score or the target score. These regions might be noise edges, unwanted edges, inconsistent features, or simply regions which are irrelevant to your search.

In the following example, 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.



With a *"flat region"* mask (**M_FLAT_REGIONS**), the masked regions are expected to be featureless in the found occurrence, and any edges present in these regions will reduce the target score. Masked edges in the model edge map will be ignored while edges in the corresponding regions of the target edge map will lower the target score. Features that are present in a region where none are expected can indicate that an error has occurred, for example in the manufacturing process.

With a *"weighted region"* mask (**M_WEIGHT_REGIONS**), the extracted model features have weights according to the values of the *"weighted region"* mask. The weights of the features are used to calculate the score; although the weights themselves don't affect the target score, edges corresponding to negative weights will be ignored when calculating the target score. The valid range for the weights of the mask is -127 to +127.

Positive weights indicate how significant it is for a given pixel to be part of an edge in the occurrence; the more positive the weight, the greater the influence on the score. The absence of a pixel that corresponds to a very positive weight reduces the score more than the absence of one with a lower weight. The longer the edges with a positive weight, the more significant the edges to the target coverage and to the target score.

Negative weights indicate how significant it is for a given pixel not to be part of an edge of the occurrence; the more negative the weight, the more the influence on the score. The presence of a pixel that corresponds to a very negative weight will reduce the score more than the presence of one that corresponds to a less negative weight.

The effect of positive and negative weights on the score is relative. A "*weighted region*" mask that has negative weights of -100 and positive weights of 100 will have the same score as the same mask where the negative weight is -1 and the positive weight is 1.

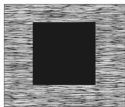
Note that weights that have no correspondence to edges in the model have no effect on the score.

"*Weighted regions*" are particularly useful when the pattern you are searching for in the target also exists as a subset of other patterns; that is, you cannot otherwise identify the required pattern uniquely. In these cases, you have to create a model that includes the unwanted extra edges and mask the extra edges with negative weights.

In the example below, the square is the shape that is being searched for, while every other shape is unacceptable. The "*weighted region*" mask will cause the one acceptable shape to be found with a high score, while ensuring that the two other shapes that are similar are found with a low score.

Model

Don't care mask

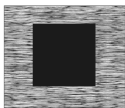


Score: high for 3 occurrences.

Target score: high for 3 occurrences.

Model

Flat region mask

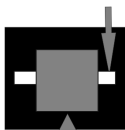
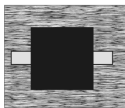


Score: high for 3 occurrences

Target score: high for 1 occurrence, lower for 2 occurrences

Model

Weighted regions mask
(negative weight)

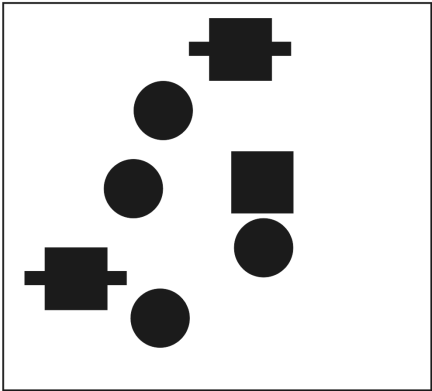


Weighted regions mask
(positive weight)

Score: high for 1 occurrence, low for 2 occurrences

Target score: high for 3 occurrences

Target Image



A separate image buffer is used to create each mask and each buffer must be of the same size as the model. Note that if you modify the margin of a synthetic model's bounding box after a mask has been applied, the mask will be discarded since it is no longer the same size as the model. You must then re-apply a mask that is of the same size as the new model box.

For a *"don't care"* or *"flat region"* mask, the buffer must be a 1-band 8-bit unsigned buffer, where any non-zero pixel value is considered a masked pixel. For a *"weighted region"* mask, the buffer must be a 1-band 8-bit signed buffer and the valid range for the weights of the mask is -127 to +127. This buffer can be calibrated; however, even if the model is calibrated, the mask is applied on a pixel basis.

Note that if you expect significant misalignment between the model and the target, you might need to compensate for this by slightly increasing the region of the *"don't care"* or *"flat region"* pixels in the mask.

If you want to verify the mask that has been applied to a model, you can use **MmodDraw()** to draw the model's masked pixels. If necessary, you can clear a model's current mask by setting the mask to **M_NULL**.

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

Search targets

You can search for instances of a model in an image, or you can search for instances of it in an Edge Finder result buffer. In both cases, use **MmodFind()** to find the model. This function will write the results of the search in the specified Model Finder result buffer.

Finding models in an image

To search for instances of models in an image, the target image must be a 1-band, 8-bit unsigned image. The minimum and maximum target image sizes are 16x16 and 32768x32768 pixels, respectively. It is important to note that the minimum and maximum target image sizes are MIL limits. To make sure that you have enough memory, you can call **MmodFind()**; if you don't have enough memory, you will get a MIL error.

The target image must be in the same calibrated state as the models. If the models are calibrated, the target must be; if the models are not calibrated, the target must not be.

You can search for a synthetic model in a calibrated or non-calibrated target image. If the target is calibrated, the model units should be the same as the calibrated units. In addition, you must associate the calibration object of the target image with the model using **MmodControl()** with **M_ASSOCIATED_CALIBRATION**. MIL needs the calibration object for internal purposes at preprocessing time. Note that MIL will not use the calibration object to compensate for incongruencies between the model and the target, nor will MIL map the model's coordinate system to that of the target.

Note that the edge extraction process, applied to the target image, uses the same denoising operation and detail level as the one for all image-type models in the Model Finder context. For more information, see the *Extracting edges* subsection in the *Defining and adding models to your model finder context* section in *Chapter 8: Geometric Model Finder*.

Finding models in an Edge Finder result buffer

When using an Edge Finder result buffer as the target, Model Finder searches for active edges of the models only in the included edges of the result buffer. Therefore, using the Edge Finder result buffer as a target, instead of an image, allows you to be more specific about the edges in which to search. For example, you could select out unwanted edges in the target, which will speed up the search and further calculations, or create a target that is composed of user-defined edges.

Some things have to be done before you can search for instances of a model in an Edge Finder result buffer.

- The Edge Finder result buffer must have been previously allocated using **MedgeAllocResult()**.
- The Edge Finder result buffer must be compatible with the Model Finder context. Enable this using **MedgeControl()** with **M_MODEL_FINDER_COMPATIBLE**.
- **MedgeCalculate()** must have already been called using this result buffer.

If the models of the Model Finder context are calibrated, the results in the Edge Finder result buffer must also be calibrated. If the models are not calibrated, the results must not be calibrated either. In the case of synthetic models, the same rules apply as when using a calibrated image as a target.

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 is determined. The score (**M_SCORE**) and the target score (**M_SCORE_TARGET**) are the primary factors in determining which occurrences are considered matches with the models in your Model Finder context.

Score and target score

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. Active edges in the model not found in the occurrence reduce the score. 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. The model scores are calculated as follows:

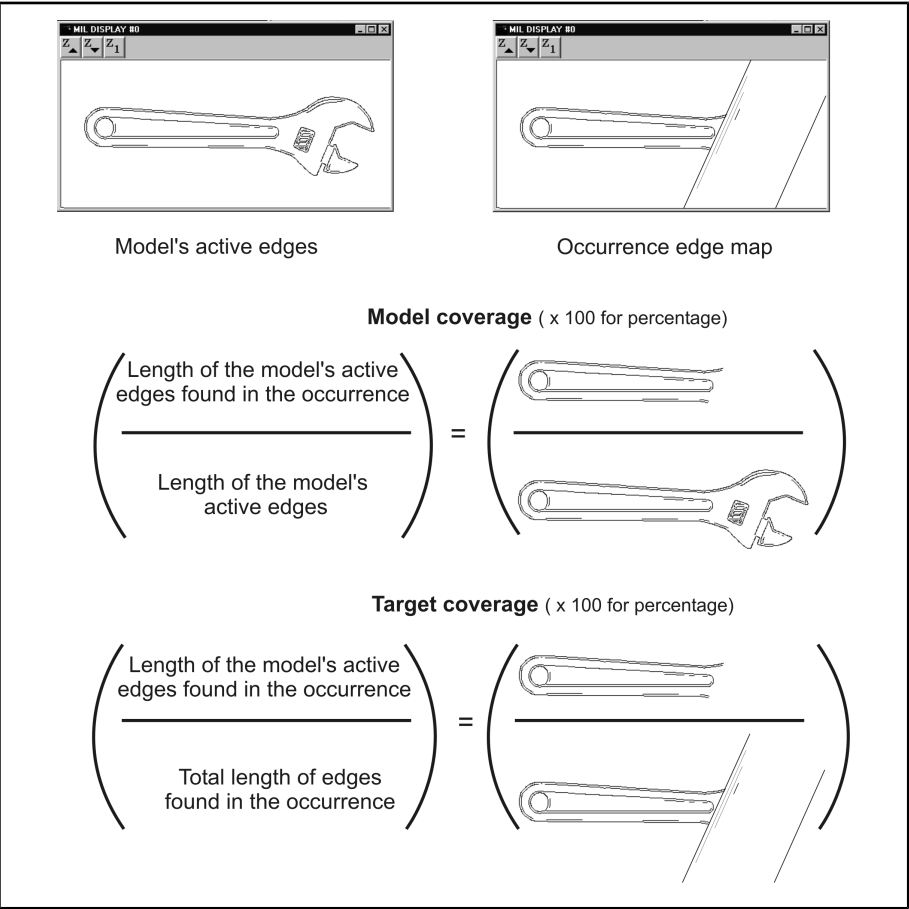
- $\text{Score} = \text{Model coverage} \times (1 - (\text{Fit error weighting factor} \times \text{Normalized Fit Error}))$.
- $\text{Target Score} = \text{Target coverage} \times (1 - (\text{Fit error weighting factor} \times \text{Normalized Fit Error}))$.
- ❖ Note: the normalized fit error is the 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 and target coverage

The model coverage and target coverage are defined as follows:

- **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 each of the model's active edges, a corresponding edge was found in the occurrence.
- **Target coverage.** The target coverage is a percentage of the total length of edges present within the occurrence's bounding box, corresponding to the model's active edges. 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.



Using a weighted mask with a model can affect the score and target score. In cases where a weighted mask is used, the model coverage would be as below:

$$\text{Model coverage} = \frac{\sum \left(\left(\text{Length of associated weighted model edges with weight } W_i \right) \times \left(\text{Weight } W_i \right) \right)}{\sum \left(\left(\text{Length of all positively weighted model edges with positive weight } W_i \right) \times \left(\text{Positive weight } W_i \right) \right)}$$

The target coverage is as follows:

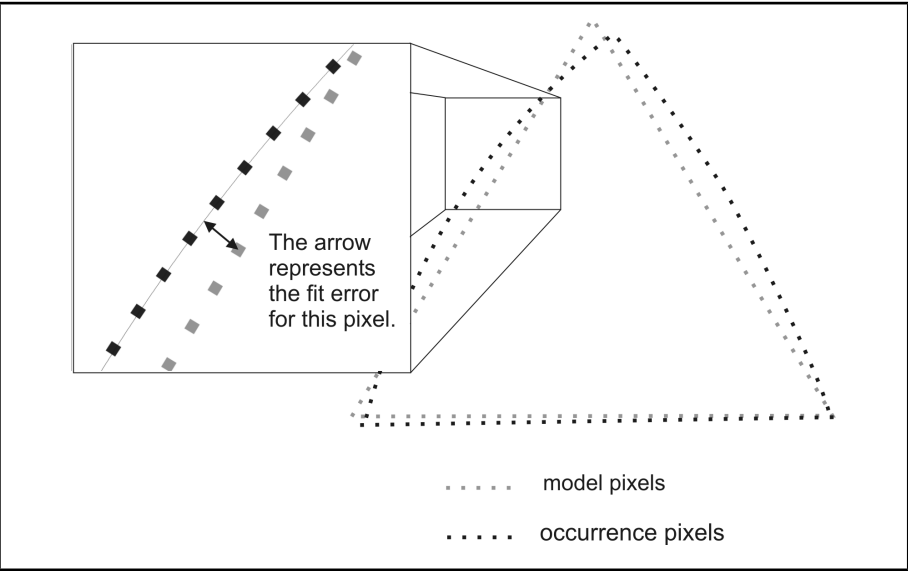
$$\text{Target coverage} = \frac{\text{Length of target edges associated with positively weighted}}{\left(\text{Length of the target's extra edges} \right) + \left(\text{Length of target edges associated with positively weighted model edges} \right)}$$

Fit error

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 edgels in the occurrence and the corresponding active edges in the model:

$$\text{Fit error} = \frac{\sum_{\text{All common pixels}} \left[\left(\text{Error in } x \right)^2 + \left(\text{Error in } y \right)^2 \right]}{\text{Number of common pixels}}$$

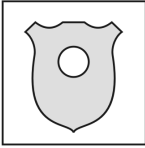
A perfect fit gives a fit error of 0.0. The fit error weighting factor (between 0.0 - 100.0) determines the importance to place on the fit error when calculating the score and target score.



Interpreting results

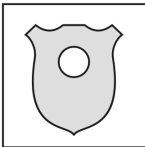
The following diagram illustrates how the model coverage, target coverage, and fit error work together to provide details about the nature of a result occurrence.

Model



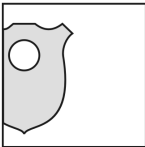
The image on the left will be used as our sample model to explain how the different scores can provide information about particular occurrences found. Note that values given are for illustrative purposes only, and as such are only qualitative.

Target images



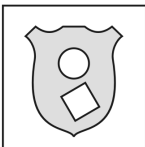
Model Coverage = 100%
Target Coverage = 100%
Fit Error = 0.0

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.



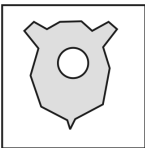
Model Coverage = 70%
Target Coverage = 100%
Fit Error = 0.0

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. The fit error is 0.0 because the model edges found in the occurrence are a perfect fit.



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. There are no extra edges in the occurrence, so the target score is still perfect. The edges found in the occurrence do not conform perfectly in shape to those in the model, resulting in a higher fit error score.

Position, angle, and scale

You can control the position, angle, and scale at which Model Finder searches for each model in a target.

You can restrict the position, angle, and scale at which model occurrences can be found, to an expected (nominal) position, angle, or scale or to a given range.

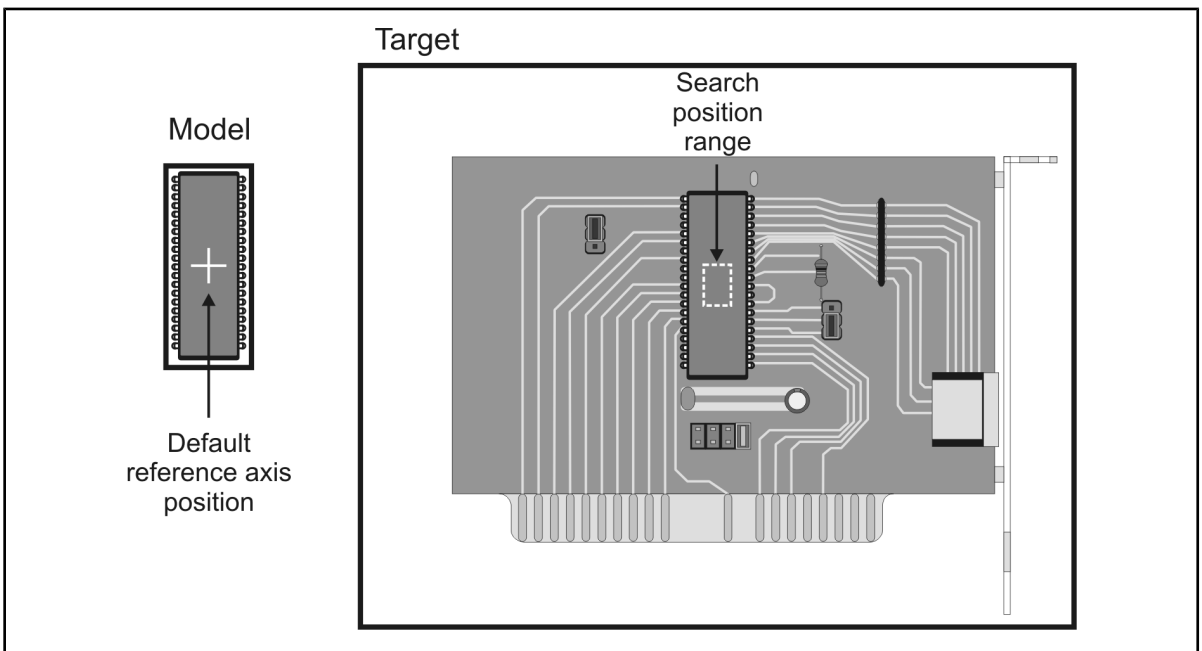
Enabling calculations specific to searching within a range

Depending on whether searching for models within a range of positions, angles, and/or scales, Model Finder uses different search strategies to evaluate the edge-based features of the target candidates. Typically, to search for models within a range, calculations specific to the corresponding search strategy should be enabled for the context (**MmodControl()** with **M_SEARCH_POSITION_RANGE**, **M_SEARCH_ANGLE_RANGE**, and/or **M_SEARCH_SCALE_RANGE**). If you expect that the occurrences sought are close to the specified nominal position, angle, or scale (for example, in a registration application), you can try disabling the calculations specific to the corresponding search strategies to see if Model Finder can still find the required occurrences. Disabling one or more of these calculations might speed up the search depending on the model and the target.

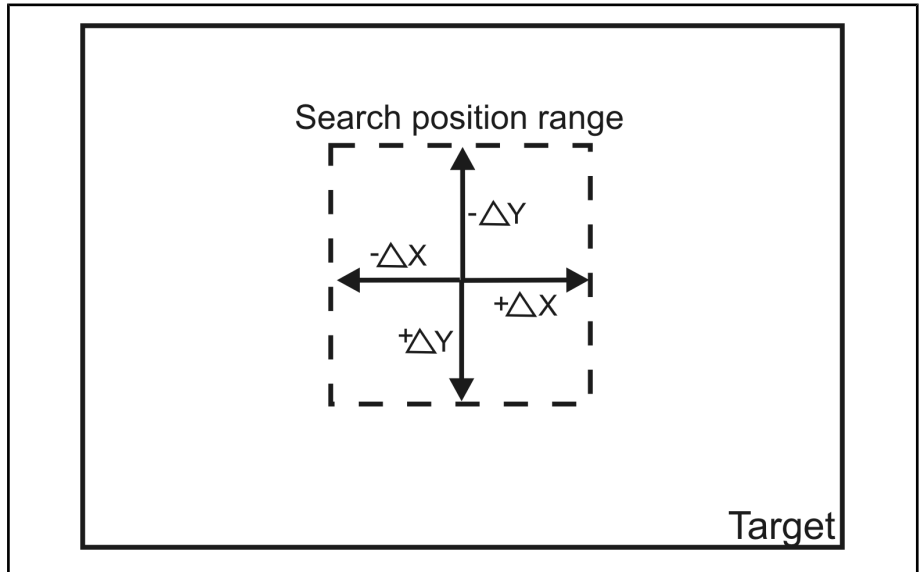
Note that candidates can only be returned as results if found within the ranges specified for their model. Therefore, if you have disabled calculations specific to a search strategy and occurrences at positions, angles, and/or scales outside of a required range are being returned, you can restrict which candidates are returned as occurrences by narrowing the appropriate ranges.

Search position and position range

You can search for each model defined in a Model Finder context at a specific position, or within a position range. The position range limits the region in which the position of a model occurrence can be found; position coordinates which fall outside this region cannot be returned as results (**MmodGetResult()** with **M_POSITION_X** and **M_POSITION_Y**). Note that the position returned for an occurrence is determined by the model's reference axis origin; by default, this position is set to the center of the model, however it can be displaced if necessary (see the *Reference axis* subsection in the *Customizing search settings* section in *Chapter 8: Geometric Model Finder*).



The region defined by the default position range is the entire image plane (**M_INFINITE**) in the positive X- and Y-direction, meaning that all X- and Y-coordinates in the positive direction (even outside the target if applicable) can be returned as results. You can use the **MmodControl()** **M_POSITION_DELTA_NEG_X**, **M_POSITION_DELTA_NEG_Y**, **M_POSITION_DELTA_POS_X**, and **M_POSITION_DELTA_POS_Y** control types to set the position range relative to the nominal position. You can, if necessary, set a different position range for each model in your context.



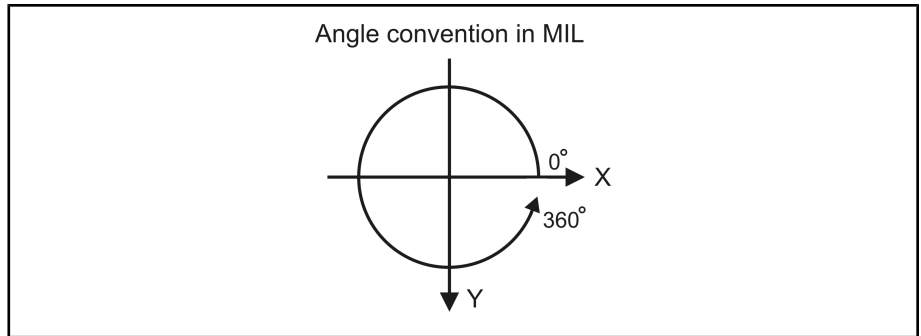
Depending on the number of details present in the target, using a small position range generally decreases the search time. Always set the position range to the minimum required when speed is a consideration; **M_POSITION_DELTA_NEG_X**, **M_POSITION_DELTA_NEG_Y**, **M_POSITION_DELTA_POS_X**, and **M_POSITION_DELTA_POS_Y** can be greater or equal to zero. If they are all set to zero, the occurrence must be at the position specified by the nominal position.

Note that you can specify a position range which defines a region partially, or totally, outside of the target; this might be necessary, depending on the reference axis origin of the model.

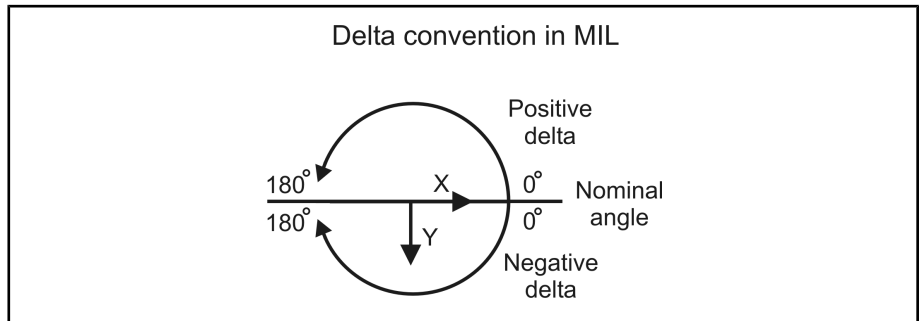
By default, calculations specific to position-range search strategies are enabled. If you are searching for a single occurrence of each model in your context and you know the nominal position of each (for example, registration applications), you can disable the calculations specific to position-range search strategies for the context, using **MmodControl()** with **M_SEARCH_POSITION_RANGE** set to **M_DISABLE**. When disabled, you must specify a good nominal position for each model (using **MmodControl()** with **M_POSITION_X** and **M_POSITION_Y**). You can restrict which candidates are returned as occurrences by narrowing the position-range.

Angle and angular range

You can search for each model defined in a Model Finder context at a specific angle, or within 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 also search within the full angular range of 360° from the nominal angle specified with **M_ANGLE**. Use the **MmodControl()** **M_ANGLE_DELTA_POS** and **M_ANGLE_DELTA_NEG** control types to specify the angular range in the counter-clockwise and clockwise direction from the nominal angle, respectively; the default for both is 180° . The 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. If you need to search for a model at discrete angles only (for example, at intervals of 90°), it is typically more efficient to define several models with different expected angles, than to search through the full angular range.



By default, calculations specific to angular-range search strategies are enabled. If you expect that the occurrences sought are close to the specified nominal angle, you can disable these calculations using **MmodControl()** with **M_SEARCH_ANGLE_RANGE** set to **M_DISABLE**. When disabled, you must specify a good nominal angle for each model, which is within the model's angular range. You can restrict which candidates are returned as occurrences by narrowing the angular-range.

Note that **M_SEARCH_ANGLE_RANGE** must be enabled to search for a rotation-invariant non-synthetic model (for example, an image-type model of a circle).

Scale and scale range

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

When the scale of occurrences can vary around the specified nominal scale, you can enable calculations specific to scale-range search strategies for the context using **MmodControl()** with **M_SEARCH_SCALE_RANGE** set to **M_ENABLE**. To specify the range of scales, use **MmodControl()** with **M_SCALE_MAX_FACTOR** (1.0 to 2.0) and **M_SCALE_MIN_FACTOR** (0.5 to 1.0). The minimum factor and the maximum factor together determine the scale range from the nominal scale (**M_SCALE**).

The maximum and minimum factors are applied to the **M_SCALE** setting as follows:

Maximum scale = (**M_SCALE**) x (**M_SCALE_MAX_FACTOR**).

Minimum scale = (**M_SCALE**) x (**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. 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.

When calculations specific to scale-range search strategies are enabled, the scale range should be used to cover an expected variance in the scale; you should not use the scale range to cover different expected scales at different positions. In this case, it is typically more efficient to define several models with different expected scales. This is because a large scale range could potentially slow down your operation; as well, you could find unwanted occurrences.

By default, calculations specific to scale-range search strategies are disabled. When disabled, you must specify a good nominal scale for each model, which is within the model's scale range. Note that occurrences can still be found within the scale range specified for their model. You can restrict which candidates are returned as occurrences by narrowing the scale-range.

Customizing search settings

With successive calls to **MmodControl()**, you can customize the global search settings of a Model Finder context, the search settings of each individual model in the context, and the general settings of a Model Finder result buffer; specify which settings to modify using the **Index** parameter (**M_CONTEXT**, the model's index, or **M_GENERAL**, respectively). You can also customize a search setting for all the models in a context by setting the **Index** parameter to **M_ALL**. However, when using **M_ALL**, ensure that the specified setting is appropriate for all models. You can inquire about any Model Finder context setting, individual model setting, or Model Finder result buffer setting using **MmodInquire()**.

This section discusses the fundamental model search settings that are usually necessary to adjust (and not previously discussed) and their inter-related global context settings. Other settings are discussed later.

Fundamental model search settings include the acceptance and certainty levels which determine what is considered a match, the expected number of occurrences to find, the reference axis which determines the position returned for a match. Most 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 as required.

Acceptance levels

For each model, 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 for the required number of occurrences, returning the occurrences with the best scores greater than or equal to 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 a 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 with a perfect fit. That is, for every active edge in the model, an equivalent edge must be found in the occurrence with a perfect fit. However, perfect matches are generally unobtainable in real images because of noise and distortion introduced when grabbing images. 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, and common edges will have a perfect fit. A setting of 0% (default) 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.

Note that the contribution of the fit can be controlled using **MmodControl()** with **M_FIT_ERROR_WEIGHTING_FACTOR**. For more information, see the *Fit error weighting factor* subsection in the *Advanced search settings* section in *Chapter 8: Geometric Model Finder*.

Certainty levels

The certainty levels are used to speed up the search when very good matches are expected often. Any occurrence that is greater than or equal to the certainty levels is considered a certain match.

The certainty levels determine the score and target score above (or equal to) which the algorithm can assume that it has found a certain match. Both of these scores must be greater than or equal to their respective certainty levels for an occurrence to be considered a certain match. If the required number of occurrences has been found above (or equal to) these certainty levels, MIL stops searching the target for better ones, avoiding exhaustive checking of all possible candidates. If not, MIL will continue to search for the required number of matches greater than or equal to the acceptance level and with the best score.

You can set the certainty level for the score of a specified model using **MmodControl()** with **M_CERTAINTY** (the default setting is 90%). You can set the certainty level for the target score of a specified model using **M_CERTAINTY_TARGET** (the default value is 0%).

If necessary, when searching for a fixed number of occurrences in a target, you can ensure that MIL always returns those occurrences with the highest match score among those found, by setting the certainty levels of each individual model to 100%. MIL will return occurrences with the best score(s) greater than or equal to the acceptance or those with scores of 100%. Note that doing so will probably increase the search time.

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.

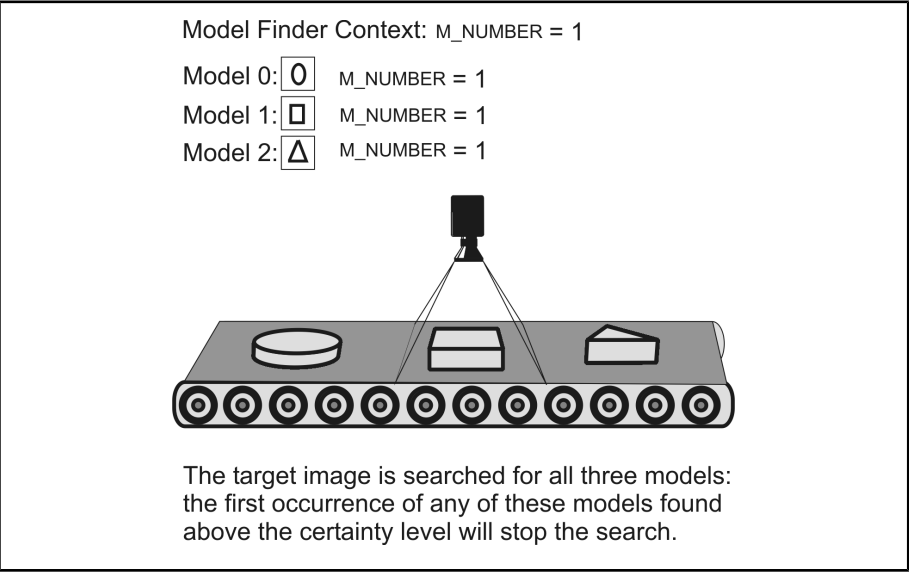
For each individual model in the context, you can set the expected number of occurrences of the model in the target. To do so, use **MmodControl()** with **M_NUMBER**, specifying the index of the particular model. The default value is 1. To find all occurrences of a model, set **M_NUMBER** to **M_ALL**.

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.

Note that setting the number of occurrences for the context to **M_ALL** and the number of occurrences for one of the models to **M_ALL** can significantly slow the **MmodFind()** operation, depending on the complexity of your model and the target (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.

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 greater than or equal to 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 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 for all three models; the first occurrence of any of these models, found above (or equal to) 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 tables below show 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

The following table shows that if certain M_NUMBER settings are specified, different combinations of maximum number of results are possible.

Model Finder Context Element	Setting for M_NUMBER	Ex. 1	Ex. 2	Ex. 3	Ex. 4	Ex. 5	Ex. 6
Context	6	6	6	6	6	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 of a model is found, the model's reference axis determines the coordinates and angle returned as the actual occurrence's found position and angle. Position results return the coordinates of the model's reference axis origin transformed at the model occurrence. Angle results are also returned relative to the reference axis, rather than the model source image axis.

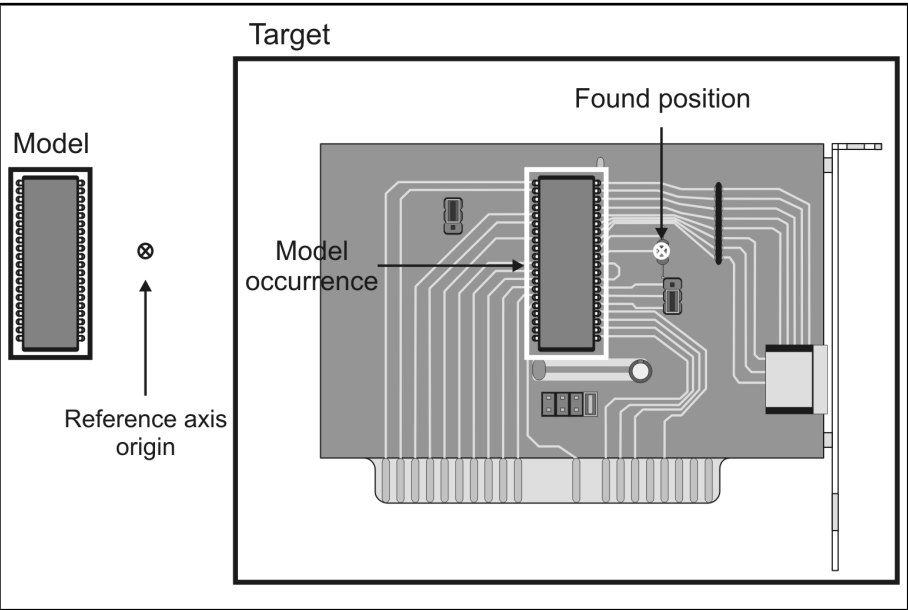
Reference axis origin

By default, the reference axis origin, for an image-type or result-type model, is at the center of the model box. For a synthetic model, the reference axis origin is, by default, at the model origin.

If the default reference axis is not at a practical position for your application, you can change the position of the reference axis origin using **MmodControl()** with **M_REFERENCE_X** and **M_REFERENCE_Y**. Specify coordinates relative to the model origin; position results are returned relative to the origin of the target.

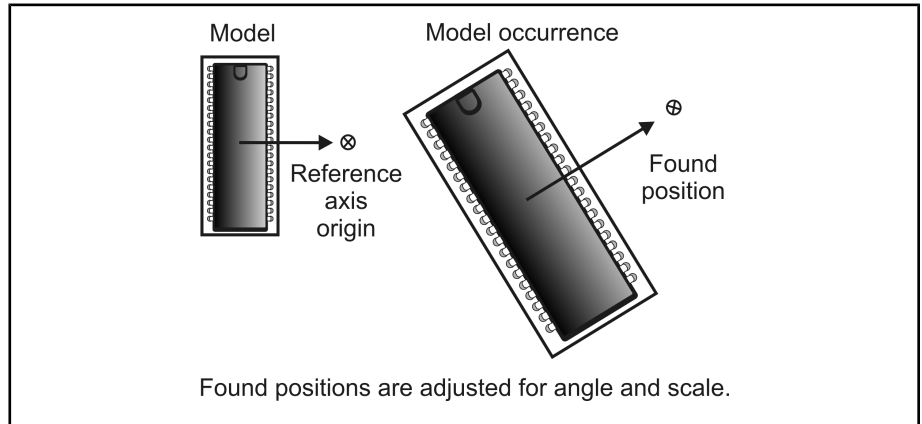
Modifying the reference axis origin can be useful if you cannot define a unique model for an object of interest. In this case, define a model that is unique and at a fixed location relative to your object of interest and move your reference axis.

In the diagram below, the easily found chip is used as the model, and the model's reference axis origin is shifted to the location of the diode.

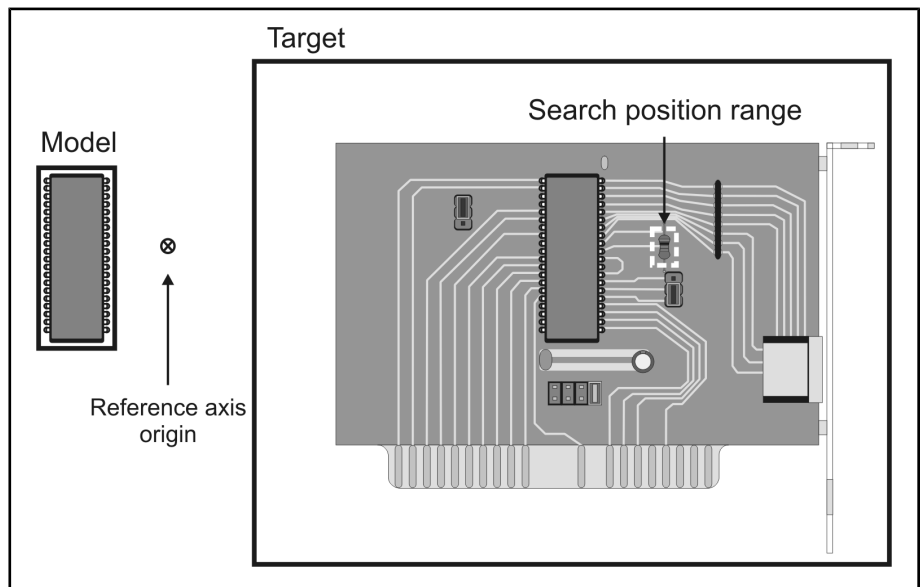


The reference axis origin does not have to reside within the model, meaning that you can place the reference axis origin outside of the model boundaries.

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.

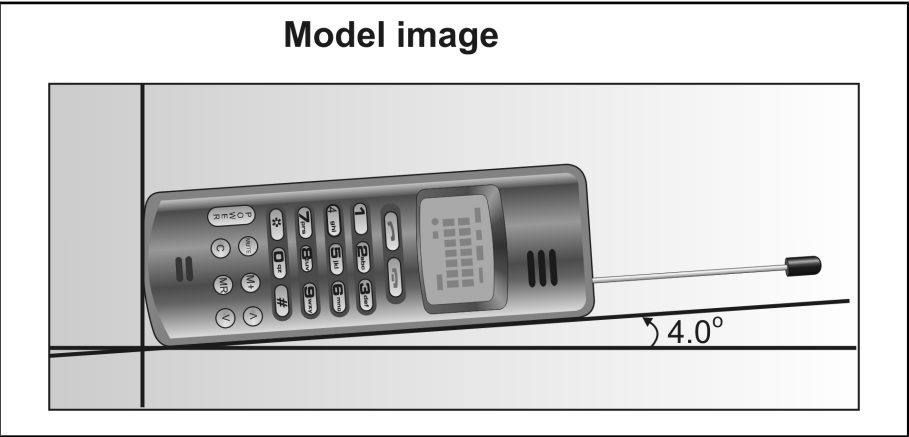


When changing the reference axis origin, make sure to update the position range if necessary, particularly when searching through a range of angles and/or scales. The position range limits the region in which the position of a model occurrence can be found; position coordinates which fall outside this region cannot be returned as results (see the *Search position and position range* subsection in the *Position angle scale* section in *Chapter 8: Geometric Model Finder*). In the diode example presented above, the position range must include, and can be limited to, a small area including the diode for the chip to be found.



Reference axis angle

Often, when allocating models, the angle of the object in your model source image is not aligned with the horizontal image axis.

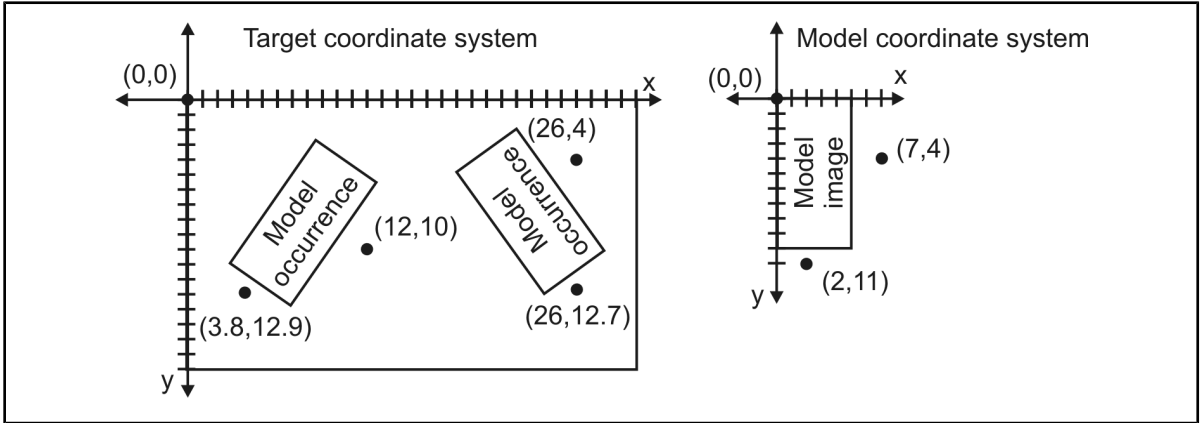


When occurrences in your target are successfully located, the angle of the occurrence returned is with respect to the model's reference axis. The actual angle of your object of interest in the model is not taken into consideration. You can align the reference axis with the object of interest using **MmodControl()** with **M_REFERENCE_ANGLE**. Angle results will then reflect the actual angle of your object of interest.

In the example above, setting the reference axis angle to 4.0 degrees will ensure that the angle returned for the occurrence will represent the actual angle of the object.

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 origin. These coefficients allow you to convert coordinates in the model coordinate system to the corresponding coordinates in the target coordinate system for that occurrence (or vice versa). These coefficients handle variations in scale, translation, and angle, allowing for easier mapping of critical points between model and occurrence.



Use the following equations:

$$xd = axs + bjs + c.$$

$$yd = -bxs + ays + d.$$

where:

- a, b, c, d : Transformation coefficients (forward or reverse).
- x_s and y_s : Source coordinates (with respect to the origin of the model coordinate system for a forward transformation or target coordinate system for a reverse transformation).
- x_d and y_d : Destination coordinates (with respect to the origin of the target coordinate system for a forward transformation or model coordinate system for a reverse transformation).

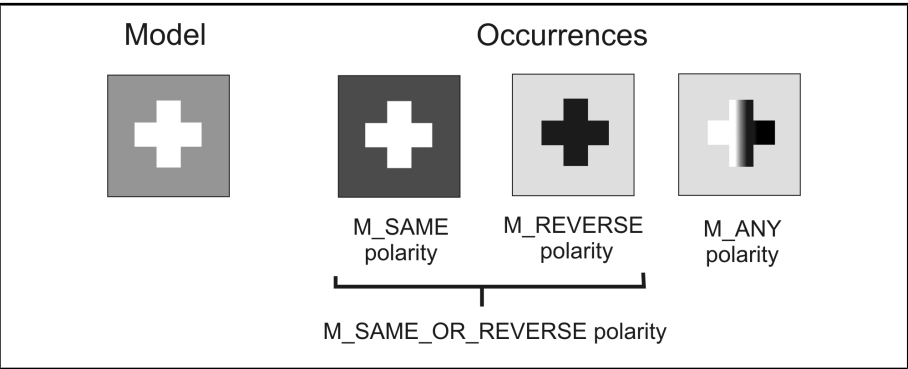
If the model is calibrated, these coefficients are given for the real world. For more information, see the *Using transformation coefficients with calibrated images* subsection in the *Calibration* section in *Chapter 8: Geometric Model Finder*.

Advanced search settings

In addition to the fundamental search settings, the Model Finder module also provides advanced settings that allow you to specify whether the polarity of the edges in a model differs from those in occurrences, how much or little separation between occurrences is permitted, as well as other information.

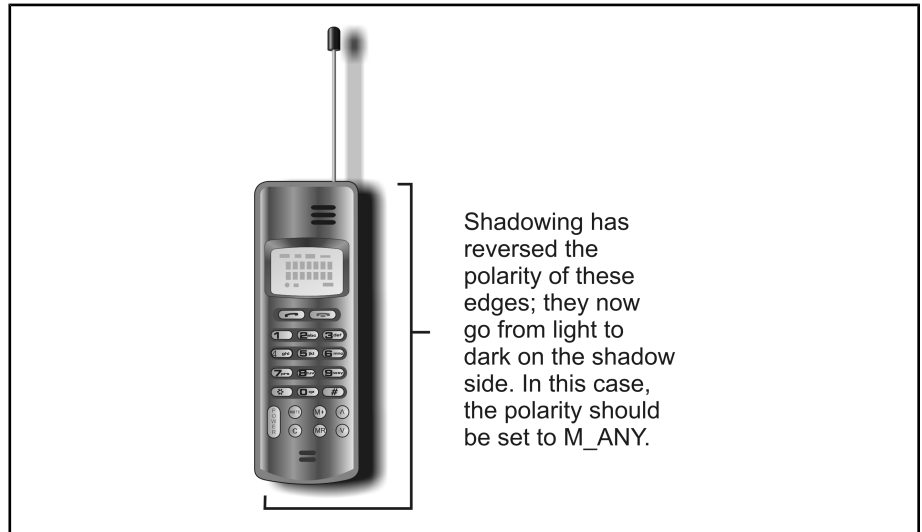
Polarity

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 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 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) for an occurrence to be considered distinct (a match). In essence, this determines what amount of overlap by different model occurrences 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.

DISTINCT OCCURRENCE = (Separation in X) OR (Separation in Y) OR (Separation in Angle) OR (Separation in Scale).

The minimum positional separation (**MmodControl()** with **M_MIN_SEPARATION_X** and **M_MIN_SEPARATION_Y**) determines the minimum distance between the found positions of two occurrences of the same model. This separation is specified as a percentage of the model size at the nominal scale (**M_SCALE**). The minimum value that you can set for **M_MIN_SEPARATION_X** and **M_MIN_SEPARATION_Y** is 0, which means that there is no minimum distance needed for occurrences to be distinct; all occurrences will be considered distinct regardless of other separation conditions. You can set very large values for **M_MIN_SEPARATION_X** and **M_MIN_SEPARATION_Y**; if they are large enough, for example, if the values are larger than the size of the image, it is equivalent to setting this separation condition to **M_DISABLE** since this condition will never be met. In this case, whether an occurrence is distinct or not will depend on the other separation conditions.

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 value that you can set for **M_MIN_SEPARATION_ANGLE** is 0, which means that there is no minimum difference in angle needed for occurrences to be distinct; all occurrences will be considered distinct regardless of other separation conditions. The maximum absolute angle difference is 180 degrees. For a diagram of the delta convention used in MIL, see the *Angle and angular range* subsection in the *Position angle scale* section in *Chapter 8: Geometric Model Finder*. If the angle is larger than 180, an error will be returned. At 180 degrees, it is equivalent to setting this separation condition to **M_DISABLE**. In this case, whether an occurrence is distinct or not will depend on the other separation conditions.

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 minimum value that you can set for **M_MIN_SEPARATION_SCALE** is 1, which means that there is no minimum difference in scale needed for occurrences to be distinct; all occurrences will be considered distinct regardless of other separation conditions. The maximum value that you can set for **M_MIN_SEPARATION_SCALE** is 4, which is equivalent to setting this separation condition to **M_DISABLE**. In this case, whether an occurrence is distinct or not will depend on the other separation conditions.

The four criteria are summarized below in equation form.

$$| \text{PositionX}_1 - \text{PositionX}_2 | \geq \frac{\text{M_MIN_SEPARATION_X}}{100} \times \text{Model width at M_SCALE}$$

$$| \text{PositionY}_1 - \text{PositionY}_2 | \geq \frac{\text{M_MIN_SEPARATION_Y}}{100} \times \text{Model height at M_SCALE}$$

$$| \text{Angle of Occurrence1} - \text{Angle of Occurrence2} | \geq \text{M_MIN_SEPARATION_ANGLE}$$

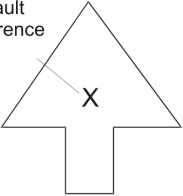
$$\text{Where } 0 \leq \text{Angle} \leq 180$$

$$\text{Max}(S_1, S_2) / \text{Min}(S_1, S_2) \geq \text{M_MIN_SEPARATION_SCALE}$$

$$\begin{aligned} &\text{Where } S_1 = \text{Scale of Occurrence 1} \\ &\quad \text{and} \\ &\quad S_2 = \text{Scale of Occurrence2} \end{aligned}$$

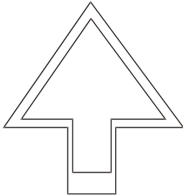
The following example illustrates the four separation criteria; an arrow represents two occurrences of the same model and the various types of separation possible.

Default
reference
axis

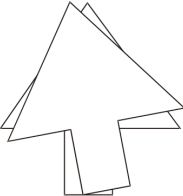


The model to the left has
the following default settings:

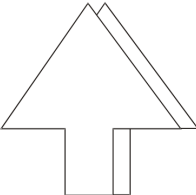
M_MIN_SEPARATION_X = 10%
M_MIN_SEPARATION_Y = 10%
M_MIN_SEPARATION_ANGLE = 10°
M_MIN_SEPARATION_SCALE = 1.1



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.



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.

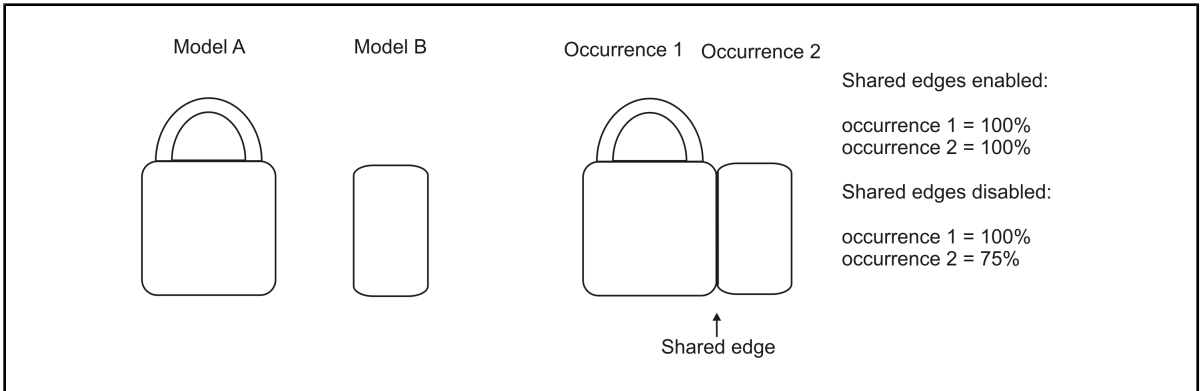


This example illustrates a case
with occurrences that have the
same scale and angle, but are
separated in the X direction by
the minimum distance and
therefore are considered distinct.

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 perfect scores.



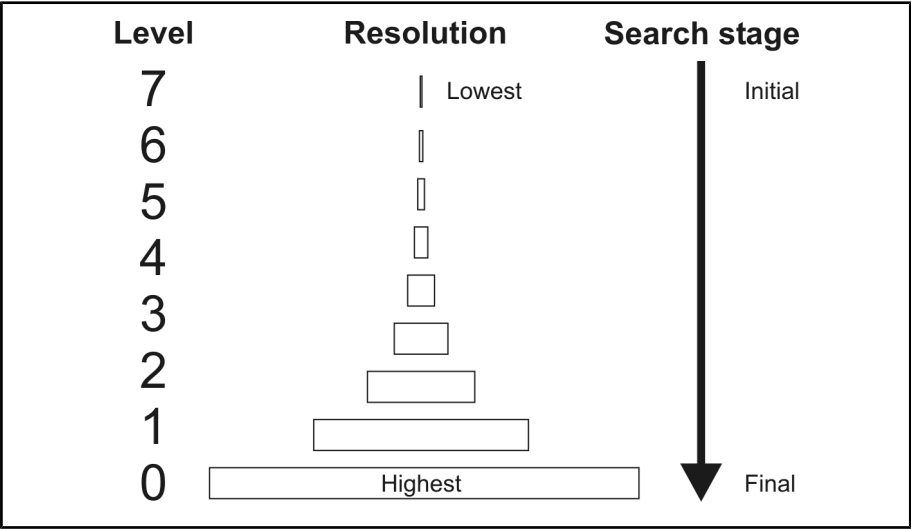
However, with shared edges disabled (default), the shared edge would be considered part of occurrence 1, since it has the greater score; the score of occurrence 2 would be subsequently reduced by the loss of the shared edge in the score calculation.

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 *Determining what is a match* section earlier in this chapter). Setting this factor to 0.0 means that the fit error is not considered in the score calculation, so the score is equal to the coverage score. Setting this factor to 100.0 means that the fit error has a maximum contribution in the score. The default value is 25.0%.

First and last levels

Model searches use a hierarchical method to reduce the time needed to complete the search. Using this method, Model Finder produces a series of smaller, lower-resolution versions of both the target and the models, 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.



The resolution level for the initial and final stages of the search are typically set automatically. These default levels usually provide the best results for search operations, and you should leave them in most circumstances. If required, you can set the resolution level for the initial and final stages of the search, using **MmodControl()** with **M_FIRST_LEVEL** or **M_LAST_LEVEL**. You can set the first and last levels to any integer value from 0 to 7. Level 0 is the original target size and each higher level is half the size (and resolution) of the previous one. The higher the level, the faster the initial search; however, the search will be less reliable at higher levels.

Very small models and search regions cannot be subsampled as many times as large ones. If you specify too high a level, the highest practical level will be used.

Global context settings

The global search settings of a Model Finder context determine the strategy used to search for all of the context's models in the target. These settings directly affect the speed and robustness of the search. The context settings can be adjusted to fit your individual application's needs, using **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.

This section discusses some of the Model Finder context settings that were not previously discussed in this chapter.

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**. When you preprocess the context, MIL analyzes the patterns of the models and determines which shortcuts are appropriate for the search speed setting. At higher search speed settings, the search can take all reasonable shortcuts; therefore, when possible, the search is performed faster than at lower speeds. You can typically use higher speed settings when high accuracy is not an issue; this is because increasing the speed might 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 ± 0.03 pixel).
- **M_HIGH** (typically ± 0.02 pixel).

Note, the actual precision achieved is dependent on the quality of the model and of the target (the values listed above are typical when using a high-quality, well-contrasted, low-noise image as the target). The default setting is **M_MEDIUM**.

Note that increasing the accuracy can slightly reduce the search speed.

Timing out your search

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

You can use **MmodGetResult()** with **M_TIMEOUT_END** to check whether the timeout limit has been reached.

Speeding up the search

There are some things you should do to help ensure that your search runs as quickly as possible:

- Adjust the search speed of the algorithm.
- Disable calculations specific to range search strategies, if possible.
- Define several models with different expected angles.
- Limit the position range.
- Limit the search scale.
- Specify the exact number of expected occurrences.
- Clean up your model.
- Change the search levels (for controlled geometric Model Finder contexts only).

Adjust the search speed of the algorithm

You can control the search speed of the algorithm used by your Model Finder context, using **MmodControl()** with **M_SPEED**. When you preprocess the context, MIL analyzes the patterns of the models and determines which shortcuts are appropriate for the search speed setting. At higher search speed settings, the search

can take all reasonable shortcuts; therefore, when possible, the search is performed faster than at lower speeds. You can typically use higher speed settings when high accuracy is not an issue; this is because increasing the speed might affect the robustness and accuracy.

Disable calculations specific to range search strategies

If you expect that the occurrences sought are close to the specified nominal position, angle, or scale (for example, in a registration application), you can try disabling the calculations specific to the search strategies for the corresponding range (**MmodControl()** with **M_SEARCH_POSITION_RANGE**, **M_SEARCH_ANGLE_RANGE**, and/or **M_SEARCH_SCALE_RANGE**) to see if Model Finder can still find the required occurrences. Disabling one or more of these calculations might speed up the search depending on the model and the target.

Define several models with different expected angles

The actual angle of the occurrence does not affect search speed. If you need to search for a model at discrete angles only (for example, at intervals of 90 degrees), it is typically more efficient to define several models with different expected angles, than to search through the full angular range.

Limit the position range

Limiting the position range of the model to the minimum required generally decreases the search time. Set the position range of each model to the minimum needed, using **MmodControl()** with the **M_POSITION_DELTA_NEG_X**, **M_POSITION_DELTA_NEG_Y**, **M_POSITION_DELTA_POS_X**, and **M_POSITION_DELTA_POS_Y** control types. If the occurrence must be at a specific position, set these control types to zero.

Limit the search scale

It is best to set a fixed scale at which to search for occurrences of a model, instead of searching in a range. Sometimes, you must search for model occurrences that can vary in size; in these cases, enabling **MmodControl()** with **M_SEARCH_SCALE_RANGE** might be necessary. In these cases, keep the scale range as small as possible to avoid slowing down your search. You should not use the scale range to cover different expected scales at different positions. In this case, it is typically more efficient to define several models with different expected scales. This is because a large scale range could potentially slow down your operation; as well, you could find unwanted occurrences.

Specify the exact number of expected occurrences

If you know the exact number of occurrences for which to search, specify it using **MmodControl()** with **M_NUMBER**. This will typically reduce search time. If you search for more occurrences than required, the search might take longer than necessary.

Clean up your model and your target

Typically, reducing the number of unwanted edges in your models and your target will accelerate your search. There are several different ways to clean up your model and your target:

- **Reduce the noise.** Noise in your models can cause inaccurate results. In addition, noise in your models and/or your target can also increase the length of time for a search. The most efficient way to remove noise is to use the **MmodControl()** **M_SMOOTHNESS** and **M_DETAIL_LEVEL** control types. These affect all image-type models in your context, as well as the target when it is an image.
- **Use masking.** You can use **MmodMask()** to mask regions of the model. Using **MmodMask()**, you can effectively remove areas of the model from consideration. This might help speed up your search.
- **Use the Edge Finder module.** You can use the Edge Finder module to extract only the required edges from the image that you want to use as your model source or target, and then use the Edge Finder results as the model source or target instead.

Changing the search levels

It is possible to speed up the search by adjusting the resolution levels for the search. This can be done using **MmodControl()** with **M_FIRST_LEVEL** and **M_LAST_LEVEL**. This should be done cautiously; their default setting (**M_AUTO**) determines the resolution levels automatically and usually provides the best results for the search operation.

Retrieving and analyzing results

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

Possible results

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 X and position Y.
- Scale.
- Polarity.
- Angle.
- Whether timeout limit has been reached.

Results are returned in descending order of match score, such that the result with 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 can be 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 Reference.

Results are indexed as positive integers starting from 0.

Drawing results

The **MmodDraw()** function provides several operations for drawing results in any specified image buffer. You can also choose to draw in the display's overlay buffer. By drawing into the display's overlay buffer, you can annotate an image non-destructively (see the *Annotating the displayed image nondestructively* section in *Chapter 20: Displaying an image*). You can also draw zoomed results. You can draw a model box, or a bounding box around the occurrence, the active edges of the model, the edges of the result occurrence, or draw a cross-like symbol at the model's reference axis origin or occurrence position.

Typically, all drawing operations in **MmodDraw()** can be combined; you can therefore draw multiple results simultaneously. For example, to draw the edges of the result occurrence and its position, you would specify **M_DRAW_EDGES + M_DRAW_POSITION**.

The following code snippet shows how to combine drawing operations using MIL constant combination:

```
MmodDraw(M_DEFAULT, MilResult, MilOverlayImage, M_DRAW_EDGES+M_DRAW_POSITION, M_DEFAULT, M_DEFAULT);
```

When drawing results, you can draw the active edges of the model transformed at the occurrence position, or the edges of the target in the region of the occurrence.

You can also draw a zoomed version of results that were obtained from a specified region in the target. To do so, specify the appropriate values for the **MmodControl()** **M_DRAW_RELATIVE_ORIGIN_X**, **M_DRAW_RELATIVE_ORIGIN_Y**, **M_DRAW_SCALE_X**, and **M_DRAW_SCALE_Y** control types. The relative origin values must be specified in pixels, and are relative to the coordinates of the top-left corner of the region in the model source, while the scale values specify the X- and Y-scaling factors used to fill the destination buffer. For more information on zooming, see the *Defining and adding models to your Model Finder context* section earlier in this chapter.

You can use a previously allocated graphics context (see *Chapter 21: Generating graphics*) to control the drawing color, or use the default graphics context (**M_DEFAULT**).

Calibration

The Model Finder module supports calibration. If the target is calibrated, matching is performed and results are calculated in the world coordinate system, compensating for any image distortion. Calibration is performed transparently, without the need for any setting adjustments.

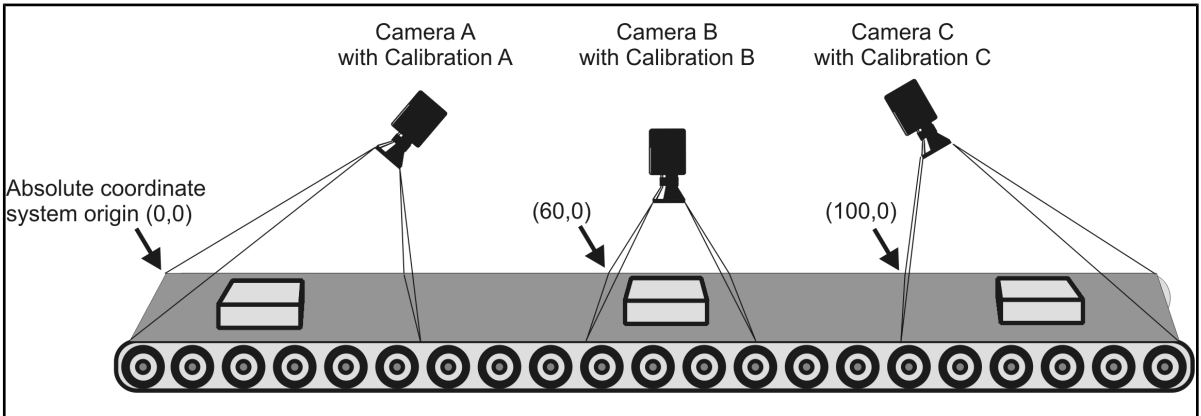
Requirements

To search in a calibrated target with a Model Finder context, all models of the context must also be calibrated. To calibrate image-type, result-type, and merge-type models, use calibrated model sources or associate a calibration object with the models, using **MmodControl()** with **M_ASSOCIATED_CALIBRATION**. Note that defining models from calibrated model sources and adjusting individual model settings is done in pixels. Synthetic models can be used with a calibrated target if defined in real-world units and associated with the calibration object of the target.

The models in a Model Finder context can be associated with different calibration objects, however, you should ensure that all calibration objects use the same absolute coordinate system (otherwise, results will be skewed).

Targets must also be calibrated using the same absolute coordinate system as the models in the Model Finder context. Targets can be either physically corrected or not, without affecting the robustness of the search.

Targets 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.



If the target is calibrated, results can be retrieved in real-world or pixel units. Note, however, in the presence of distortion some results are meaningless when converted from real-world to pixel units (for example, angle, scale, and transformation coefficient results). For example, if a model appears warped in the target, but the calibration object of the target compensates for this during the model search, the resulting angle is meaningful in the real-world coordinate system, and meaningless in the image coordinate system.

When using **MmodDraw()** to draw features from a calibrated model or results from a calibrated target, 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.

Models and their calibration object

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, 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.

Setting the aspect ratio control

When dealing with aspect ratio distortion, you can use **MmodControl()** with **M_ASPECT_RATIO** to compensate for the distortion, if the target is not calibrated. This will allow you to avoid using a calibration object during the search. This will not compensate for aspect ratio distortion in the model, so this method is mostly useful when searching for synthetic models in a target that has aspect ratio distortion.

When performing the search, results will be returned for the corrected target coordinate system. The results will be converted to the original target coordinate system if you draw them using **MmodDraw()**.

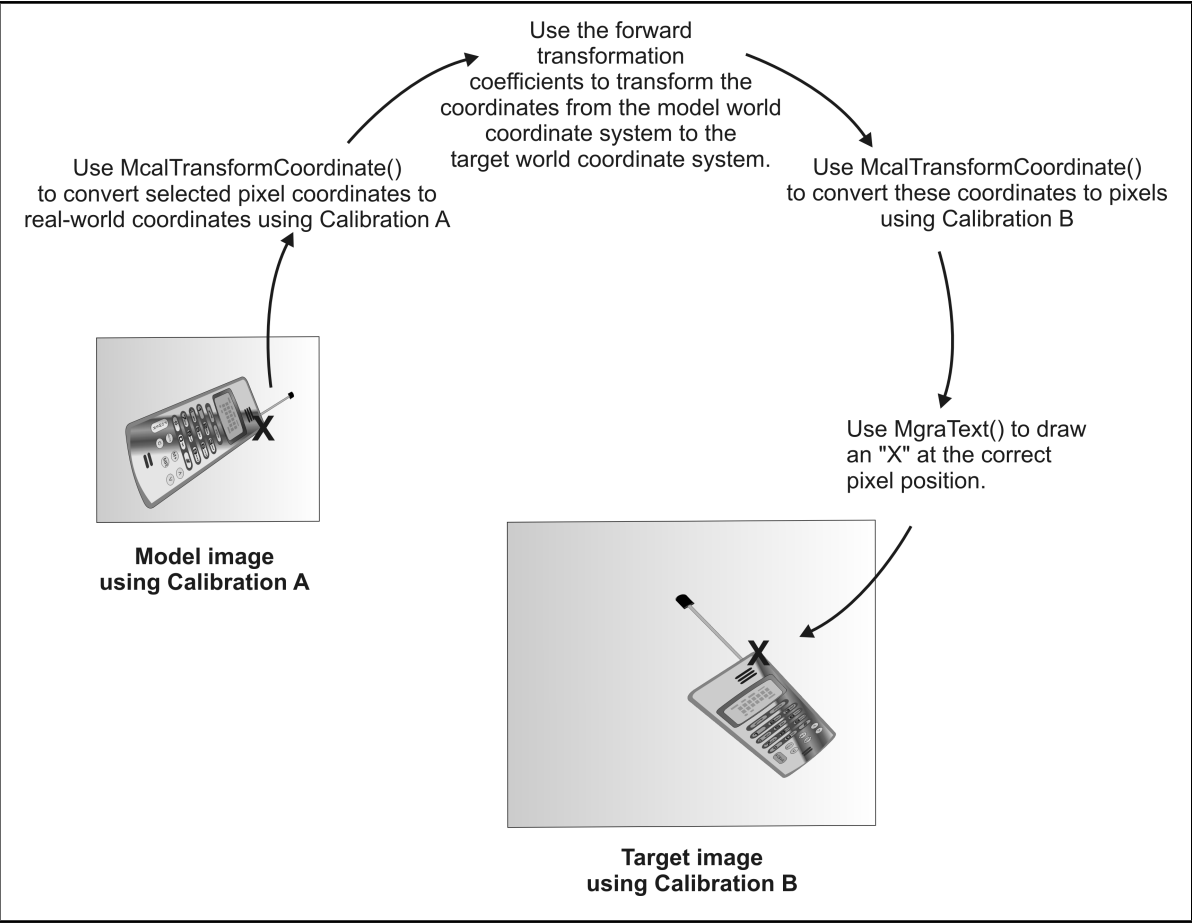
Using transformation coefficients with calibrated images

You can use forward and reverse transformation coefficients if you want to map points of interest other than the reference axis origin (see the *Forward and reverse transformation coefficients* subsection in the *Customizing search settings* section in *Chapter 8: Geometric Model Finder*). When using a calibrated model and a calibrated target, forward and reverse transformation coefficients (see **MmodGetResult()**) are given for the real-world. This means that they can be used to convert any coordinates in the model world coordinate system to the corresponding coordinates in the target world coordinate system for an occurrence (or vice versa).

Forward and reverse transformation coefficients can be used to determine and draw equivalent positions in a calibrated model image and a calibrated target image, for an occurrence. Note, however, that the **Mgra...** functions only take pixel coordinates. Therefore, to draw a pixel of interest from a calibrated model to a calibrated target, you must first convert the required pixel coordinates in the model image to real-world coordinates, using **McalTransformCoordinate()**. Following that, you have to get the forward transformation coefficients using

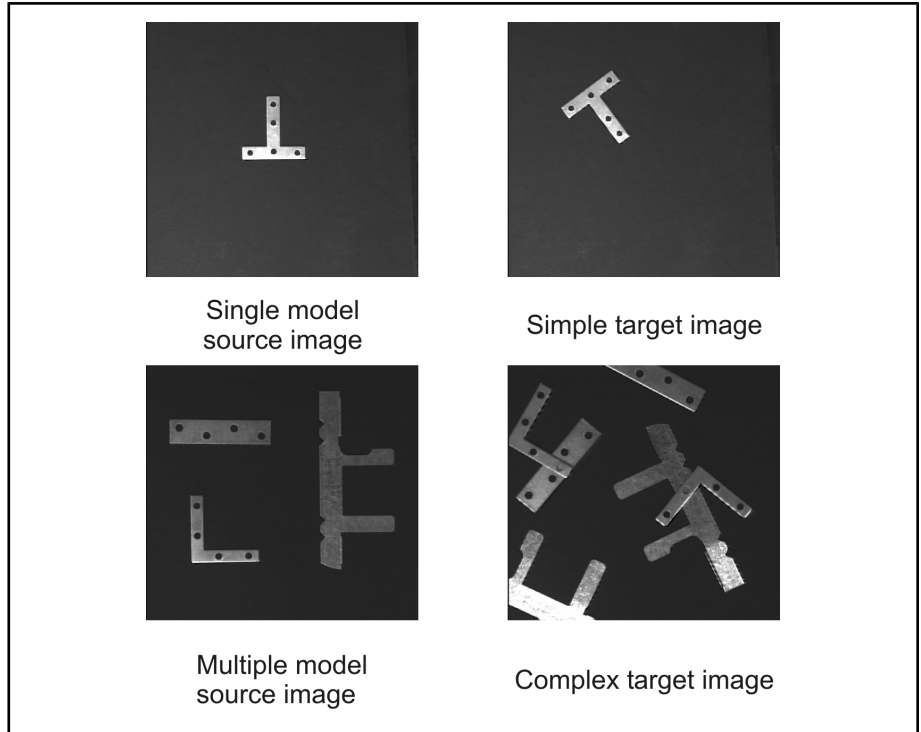
MmodGetResult(). Apply the transformation coefficients to the coordinates; then convert the real-world coordinates back to pixel coordinates, using **McalTransformCoordinate()** again. These coordinates can then be used to draw in the target image. A reverse transformation is done similarly, except you start from the calibrated target.

See the image below for an example of a forward transformation.



Geometric Model Finder example

The Model Finder example, *MMod.cpp* illustrates how the model can be used with the following source images:



Specifically, *MMod.cpp* 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: MMod.cpp
*
* Synopsis: This program uses the Geometric Model Finder module to define geometric
*           models and searches for these models 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 (multiple models,
*           multiple occurrences in a complex scene with bad search conditions).
*/
#include <mil.h>

```

```

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

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

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

    /* Run single model example. */
    SingleModelExample(MilSystem, MilDisplay);

    /* Run multiple model example. */
    MultipleModelsExample(MilSystem, MilDisplay);

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

    return 0;
}

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

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

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

/* Search speed: M_VERY_HIGH for faster search, M_MEDIUM for 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

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.      */
MIL_DOUBLE  ModelDrawColor = M_COLOR_RED;  /* Model draw color.       */
MIL_INT     Model[MODEL_MAX_OCCURRENCES],  /* Model index.            */
            NumResults = 0L;               /* Number of results found.*/
MIL_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 annotation. */
MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

/* Allocate a Geometric Model Finder context. */
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_SIZEX, 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 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);

/* Pause to show the model. */
MosPrintf(MIL_TEXT("\nGEOMETRIC MODEL FINDER:\n"));
MosPrintf(MIL_TEXT("-----\n\n"));
MosPrintf(MIL_TEXT("A model context was defined with "));
MosPrintf(MIL_TEXT("the model in the displayed image.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

```

```

/* Clear the overlay image. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

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

/* Dummy first call for bench measure purpose only (bench stabilization,
   cache effect, etc...). This first call is NOT required by the application. */
MmodFind(MilSearchContext, MilImage, MilResult);

/* Reset the timer. */
MapTimer(M_TIMER_RESET+M_SYNCHRONOUS, M_NULL);

/* Find the model. */
MmodFind(MilSearchContext, MilImage, MilResult);

/* Read the find time. */
MapTimer(M_TIMER_READ+M_SYNCHRONOUS, &Time);

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

/* If a model was found above the acceptance threshold. */
if ( (NumResults >= 1) && (NumResults <= MODEL_MAX_OCCURRENCES) )
{
    /* Get the results of the single model. */
    MmodGetResult(MilResult, M_DEFAULT, M_INDEX+M_TYPE_MIL_INT, 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. */
    MosPrintf(MIL_TEXT("The model was found in the target image:\n\n"));
    MosPrintf(MIL_TEXT("Result    Model    X Position    Y Position    ")
              MIL_TEXT("Angle    Scale    Score\n\n"));
    for (i=0; i<NumResults; i++)
    {
        MosPrintf(MIL_TEXT("%-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]);
    }
    MosPrintf(MIL_TEXT("\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);
    }
}

```

```

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

/* Wait for a key to be pressed. */
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

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

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

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

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

/* Search speed: M_VERY_HIGH for faster search, M_MEDIUM for 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_COLOR_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_COLOR_GREEN

/* 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_COLOR_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. */
MIL_INT      Models[MODELS_MAX_OCCURRENCES], /* Model indices. */
ModelsOffsetX[MODELS_ARRAY_SIZE] = MODELS_OFFSETX, /* Model X offsets array. */
ModelsOffsetY[MODELS_ARRAY_SIZE] = MODELS_OFFSETY, /* Model Y offsets array. */
ModelsSizeX[MODELS_ARRAY_SIZE] = MODELS_SIZEX, /* Model X sizes array. */
ModelsSizeY[MODELS_ARRAY_SIZE] = MODELS_SIZEY; /* Model Y sizes array. */
MIL_DOUBLE  ModelsDrawColor[MODELS_ARRAY_SIZE]=MODELS_DRAW_COLOR; /* Model drawing colors.*/
MIL_INT      NumResults = 0L;        /* Number of results found. */
MIL_DOUBLE  Score[MODELS_MAX_OCCURRENCES], /* Model correlation scores.*/
XPosition[MODELS_MAX_OCCURRENCES], /* Model X positions. */
YPosition[MODELS_MAX_OCCURRENCES], /* Model Y positions. */
Angle[MODELS_MAX_OCCURRENCES], /* Model occurrence angles. */
Scale[MODELS_MAX_OCCURRENCES], /* Model occurrence scales. */
Time = 0.0; /* Time variable. */
int          i; /* Loop variable */

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

/* Prepare for overlay annotation. */
MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);
MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

/* 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,
               (MIL_DOUBLE)ModelsOffsetX[i], (MIL_DOUBLE)ModelsOffsetY[i],
               (MIL_DOUBLE)ModelsSizeX[i], (MIL_DOUBLE)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);

#if (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);

#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 identify 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. */
MosPrintf(MIL_TEXT("A model context was defined with the ")
          MIL_TEXT("models in the displayed image.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Clear the overlay image. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

```

```

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

/* Dummy first call for bench measure purpose only (bench stabilization,
cache effect, etc...). This first call is NOT required by the application. */
MmodFind(MilSearchContext, MilImage, MilResult);

/* Reset the timer. */
MapTimer(M_TIMER_RESET+M_SYNCHRONOUS, M_NULL);

/* Find the models. */
MmodFind(MilSearchContext, MilImage, MilResult);

/* Read the find time. */
MapTimer(M_TIMER_READ+M_SYNCHRONOUS, &Time);

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

/* If the models were found above the acceptance threshold. */
if( (NumResults >= 1) && (NumResults <= MODELS_MAX_OCCURRENCES) )
{
    /* Get the results for each model. */
    MmodGetResult(MilResult, M_DEFAULT, M_INDEX+M_TYPE_MIL_INT, 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);

    /* Print information about the target image. */
    MosPrintf(MIL_TEXT("The models were found in the target "));
    MosPrintf(MIL_TEXT("image although there is:\n  "));
    MosPrintf(MIL_TEXT("Full rotation\n  Small scale change\n  "));
    MosPrintf(MIL_TEXT("Contrast variation\n  Specular reflection\n  "));
    MosPrintf(MIL_TEXT("Occlusion\n  Multiple models\n"));
    MosPrintf(MIL_TEXT("  Multiple occurrences\n\n"));

    /* Print the results for the found models. */
    MosPrintf(MIL_TEXT("Result  Model  X Position  Y Position  ")
              MIL_TEXT("Angle  Scale  Score\n\n"));
    for (i=0; i<NumResults; i++)
    {
        MosPrintf(MIL_TEXT("%-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]);
    }
    MosPrintf(MIL_TEXT("\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
{
    MosPrintf(MIL_TEXT("The models were not found or the number of ")
            MIL_TEXT("models found is greater than\n"));
    MosPrintf(MIL_TEXT("the defined value of maximum occurrences !\n\n"));
}

/* Wait for a key to be pressed. */
MosPrintf(MIL_TEXT("Press <Enter> to end.\n\n"));
MosGetch();

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


Chapter

9

Edge Finder

This chapter explains how to use the MIL Edge Finder module to extract and analyze edges in your image.

MIL Edge Finder module

The MIL Edge Finder module is a set of powerful functions that allow you to extract and analyze edges in a source image.

Many applications can be implemented using the Edge Finder module, such as complex measurement operations, defect detection, shape recognition, and shape analysis. You can also perform advanced edge manipulations, such as closing broken edges.

Edge Finder allows you to calculate a large number of edge features that can provide you with useful information, such as the edge's minimum and maximum Feret diameter. You can either calculate all required features for all edges, or perform a post-calculation on a refined subset of selected edges, which can speed up your processing time considerably.

The Edge Finder module can also be used in conjunction with the *Geometric Model Finder module* section in *Chapter 8: Geometric Model Finder*; that is, you can define models from the result of an edge extraction, or you can find model occurrences in the result of an edge extraction.

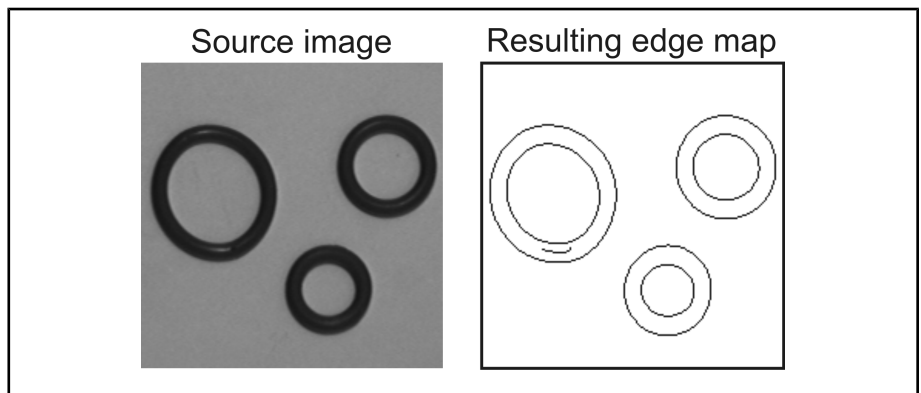
If your source image is calibrated, results are calculated in the world coordinate system, otherwise they are calculated in the image coordinate system. Also, if your source image is calibrated, you can retrieve results in either real-world or pixel units. For more information about calibration, see *Calibration*.

Edges and Edge Finder

Edges are curves that delineate a boundary. These can be established from intensity transitions in an image. Well-defined edges come from sharp transitions in value, typically found in highly-contrasted images. Conversely, weak edges come from gradual transitions in value, typically found in smooth images.

Depending on your settings, Edge Finder will extract one of two edge types: object contours or line crests. An object contour is a type of edge that defines the outline of objects in an image. A line crest is a type of edge that defines the skeleton of objects in an image. Note that line crests should be used to extract edges from objects consisting of thin curvilinear shapes, such as fingerprints.

In either case, edges are extracted from the source image and used to form the image's edge map, which represents how the image is defined as a set of edges.



All Edge Finder operations, such as annotations and feature calculations are performed using the image's edge map.

You can only extract line crests from grayscale images. However, you can extract object contours from both grayscale and color images. When operating on color images, Edge Finder takes into consideration all three bands (RGB) to establish the presence of an edge.

Steps to extract and analyze edges

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

1. Allocate an Edge Finder context, using **MedgeAlloc()**.
2. If necessary, adjust general processing controls to fit your application, using successive calls to **MedgeControl()**.
3. If necessary, select the required features to calculate, using successive calls to **MedgeControl()**.
4. Allocate an Edge Finder result buffer to hold the results of the edge extraction, using **MedgeAllocResult()**.
5. Calculate edges and edge features, using **MedgeCalculate()**.
6. If necessary, exclude or delete edges that do not meet a required criteria, using **MedgeSelect()**. Excluded edges will be ignored in future calculations, while deleted edges will be removed from the Edge Finder result buffer.
7. If necessary, select new features and perform a post-calculation, using **MedgeCalculate()**.
8. Get the number of edges currently included and retrieve the required results from the Edge Finder result buffer, using **MedgeGetResult()**. Results for the excluded or deleted edges will not be returned.
9. If necessary, get the coordinates of edgels from an Edge Finder result buffer that correspond to the closest neighbors from a list of user-specified source point coordinates, using **MedgeGetNeighbors()**.
10. If necessary, copy data from user-supplied arrays to a specified Edge Finder result buffer, using **MedgePut()**.
11. If necessary, draw edges and edge features, using **MedgeDraw()**.

12. If necessary, save your Edge Finder context, using **MedgeSave()**.
13. Free all your allocated objects, using **MedgeFree()**.

You can repeat steps 6, 7, and 8 until all the required results are obtained. Note that successively excluding or deleting unwanted edges and then calculating more features is the recommended procedure to achieve the required results, especially if you have a large number of unwanted edges.

Basic concepts

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

- **Edge.** A curve that delineates a boundary, which can be established from intensity transitions in an image. In Edge Finder, an edge is considered to be an edge chain and its features.
- **Edge angle.** The angle between the horizontal axis and the edge's perpendicular direction at each edgel location.
- **Edge chain.** The set of connected edgels that construct an edge.
- **Edge feature.** Information describing an edge, such as its length.
- **Edge magnitude.** The strength of the edge at each edgel location. For object contours, the magnitude is the norm of the gradient vector at the edgel position. For line crests, the magnitude is equal to the maximum eigenvalue of the Hessian matrix at the edgel position.
- **Edge map.** The set of edges extracted from the source image.
- **Edgel.** An elementary point (or edge element) within an edge.
- **Line crest edge.** A type of edge that defines the skeleton of objects in an image. Typically, objects should be thin curvilinear shapes.
- **Object contour edge.** A type of edge that defines the outline of objects in an image. Typically, objects should not be thin curvilinear shapes.

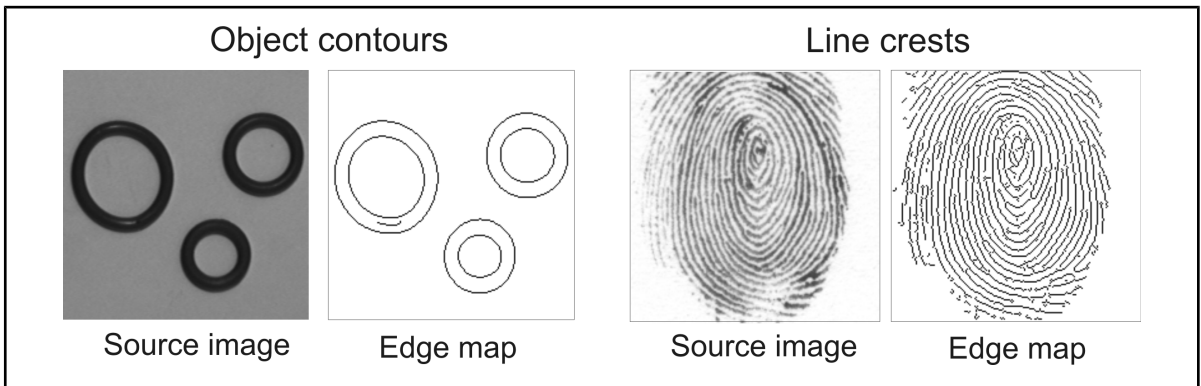
- **Post-calculation.** Any calculation made to refine results after an initial call to `MedgeCalculate()`.
 - **Source image.** The image from which the edges are extracted.
- ❖ After edge chains have been extracted, the terms edge chain and edge are used interchangeably.

Extracting the edges

Before you write even the simplest application, it is important to have a basic understanding of how Edge Finder extracts edges, and what the differences are between the various types of edges (object contours and line crests). Note that if you are writing slightly more advanced applications, it is also worth having a more in-depth understanding of how edges are calculated.

The basics

Edge Finder uses operations that are based on differential analysis, where edges (either contours or line crests) are extracted by analyzing intensity transitions in images. Transitions in intensity can come from many physical phenomena, such as shadows and illumination variations. However, strong transitions are typically caused by the presence of an object contour or a line crest in the image, as illustrated below.



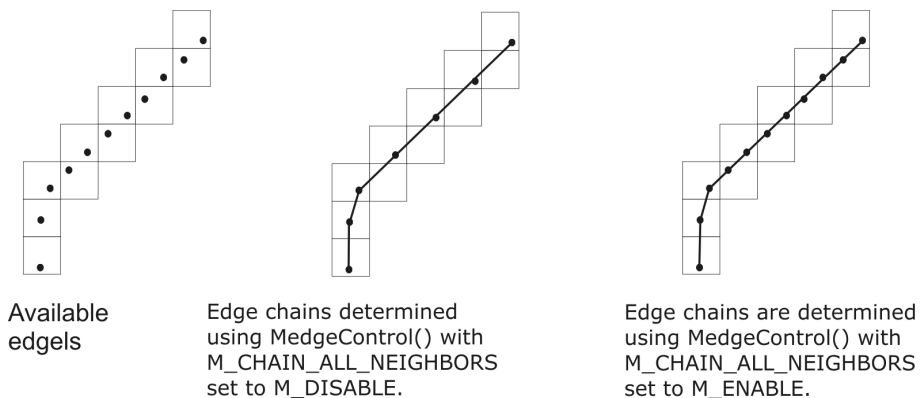
- ❖ Note that the main intensity transition in the image is observed in the perpendicular direction relative to the object border (for an object contour) or line shape (for a line crest).

Edges are extracted in three general steps. First, a filtering process provides an enhanced image of the edges based on the computation of the image's derivatives. Second, detection and thresholding operations determine, from the image enhancement, all pertinent edge elements, or edgels, and accurately calculates their positions. Third, neighboring edgels are connected to build the edge chains (which results in the image's edge map), and features are calculated for each edge. To fit the requirements of your application, Edge Finder allows you to customize each step's corresponding processing controls.

Edgels are connected into edge chains respecting the following constraints, whereby pixels are considered connected based on an 8-connected lattice:

- Consecutive edgels must occupy separate connected pixels.
- No branches are allowed (they start or end a separate chain).
- There must not be an edge in the edge map that is more than 1 pixel wide.

You can control how edgels are connected using **MedgeControl()** with **M_CHAIN_ALL_NEIGHBORS**. If the control type is enabled, the chain is created using as much edgel information as possible. If disabled, the chain is created using the least amount of edgel information possible.

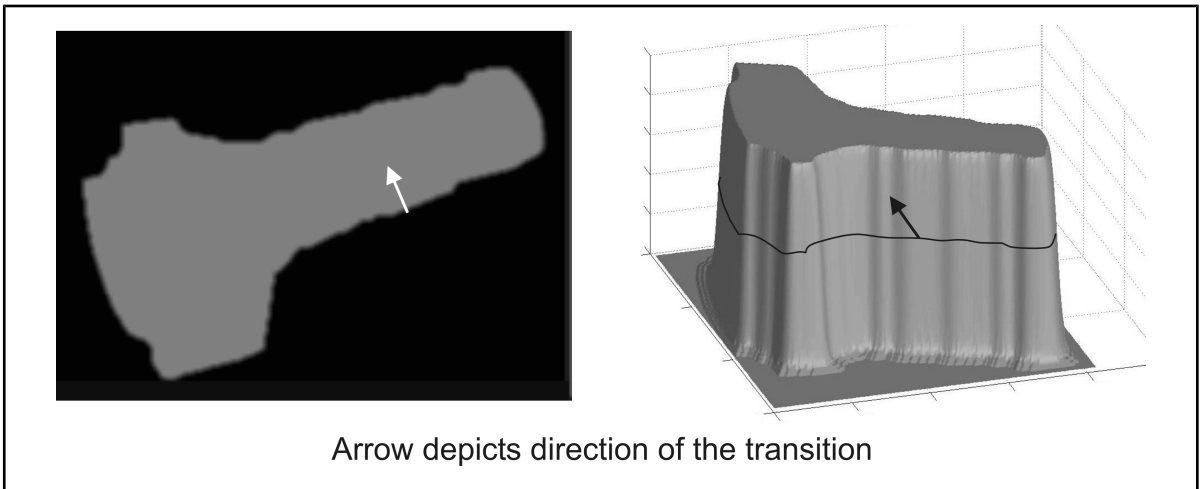


Object contours versus line crests

To properly analyze the structures present in an image, it is often useful to consider the image as a topographic surface, where the intensity of each pixel is seen as the elevation value of the surface. By doing so, contours and crests can be more easily understood.

Object contours

An object contour is the inter-pixel boundary that splits into thin layers (delaminates) the separation between the object and the image background, as illustrated below. The edge of the contour is located at the sharpest point within the intensity transition. Moreover, the intensity transition is strongest in the perpendicular direction to the object contour.

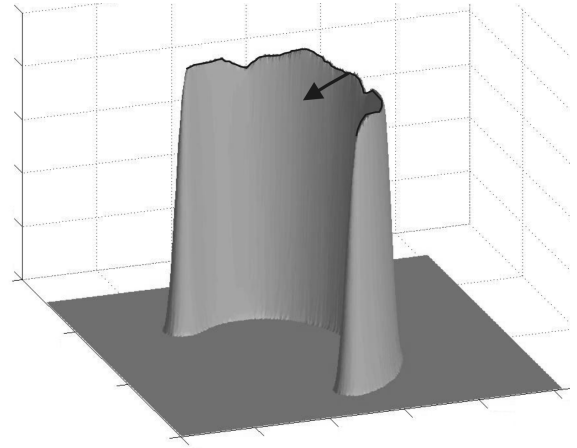
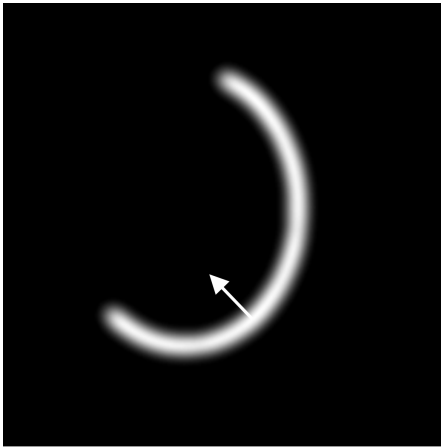


Well-defined contours have sharp transitions in value. The smoother the image, the more gradual the change, and the weaker the contour.

To allocate an Edge Finder context for object contours, use **MedgeAlloc()** with **M_CONTOUR**.

Line crests

A line crest is the skeleton of a thin curvilinear shape in the image, as illustrated below. The crest is located at the sharpest point of the Gaussian-like profile of the line shape. Moreover, the intensity transition is strongest in the perpendicular direction to the line crest.



Arrow depicts direction of the transition

Well-defined line crests are thin and have sharp transitions in value. The smoother the image or the larger the lines, the more gradual the change, and the weaker the line crests.

A line crest can either be darker (a valley-like Gaussian profile), lighter (a ridge-like Gaussian profile), or both darker and lighter than the image's background. To specify the color of the line crest to extract, set **MedgeControl()** with **M_FOREGROUND_VALUE** to **M_FOREGROUND_BLACK**, **M_FOREGROUND_WHITE**, or **M_ANY**. The default setting is **M_FOREGROUND_BLACK**.

To allocate an Edge Finder context for line crests, use **MedgeAlloc()** with **M_CREST**.

- ❖ Typically, you should use **M_CREST** Edge Finder contexts for edges that are established from thin curvilinear shapes; otherwise, you might get unexpected results.

How edges are calculated

For basic applications, you need not know how Edge Finder calculates edges. However, an understanding of this process can help you solve more advanced problems, and adjust some advanced settings.

Object contours

The enhanced image of the object contours is achieved by calculating the gradient magnitude of each pixel in the image. The stronger the intensity transition, the greater the magnitude will be.

The gradient magnitude is calculated at each pixel position from the image's first derivatives. It is defined as:

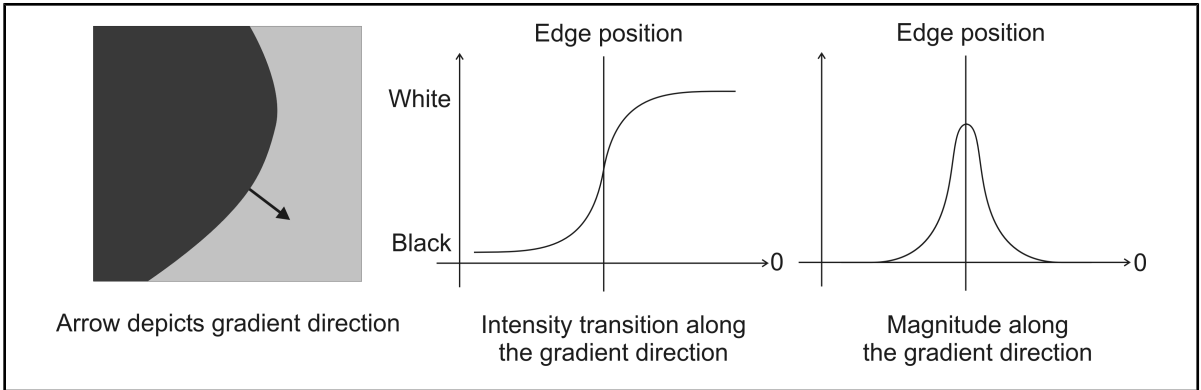
$$\text{Gradient Magnitude} = \sqrt{I_x^2 + I_y^2}$$

where I_x and I_y are, respectively, the X and Y derivative values. They define the components of the gradient vector as:

$$\text{Gradient Vector} = \begin{bmatrix} I_x \\ I_y \end{bmatrix}$$

An edgel is located at the maximum value of the gradient magnitude over adjacent pixels, in the direction defined by the gradient vector. The gradient direction is the direction of the steepest ascent at an edgel in the image, while the gradient magnitude is the steepness of that ascent. Note that the gradient direction is also

the perpendicular direction to the object contour. Well-defined contours are extracted from strong and sharp intensity transitions. Note that a strong contrast between the objects and their background improves the edge detector's robustness and location accuracy.



Note that Edge Finder also supports 3-band images, when extracting object contours. The extraction process is similar to single band images, except the enhanced image of the object contours is achieved by calculating a generalized gradient magnitude of each pixel in the image, for each color band (RGB).

Line crests

The enhanced image of the line crests is achieved by calculating the Hessian magnitude of each pixel in the image. The stronger the crest, the greater the magnitude will be.

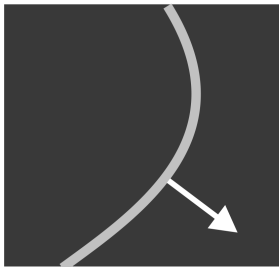
The magnitude is defined as the maximum eigenvalue of the Hessian matrix, calculated at each pixel position from the image's second derivatives. The Hessian matrix is defined as:

$$\text{Hessian} = \begin{bmatrix} I_{xx} & I_{xy} \\ I_{yx} & I_{yy} \end{bmatrix}$$

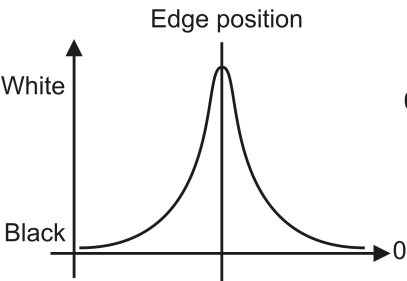
where I_{xx} , I_{yy} and I_{xy} are, respectively, the X, Y and cross second derivative values of the image.

An edgel is located at the maximum value of the image magnitude over adjacent pixels, in the direction defined by the associated eigenvector. Well-defined line crests are extracted from strong and sharp intensity transitions. Note that a strong contrast between the lines and the image's background improves the edge detector's robustness and the location accuracy.

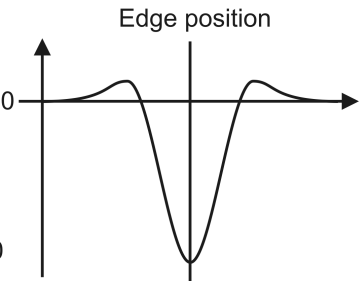
Light line crest (ridge)



Arrow depicts Hessian principal direction

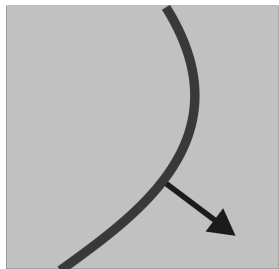


Intensity transition along the Hessian principal direction

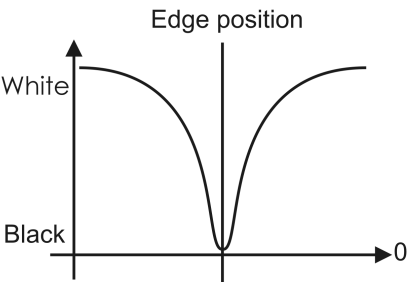


Magnitude along the Hessian principal direction

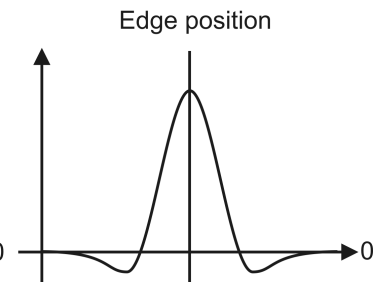
Dark line crest (valley)



Arrow depicts Hessian principal direction



Intensity transition along the Hessian principal direction



Magnitude along the Hessian principal direction

Note that the sign of the magnitude at the crest's edgel locations changes with the color of the line crest. That is, a dark crest has a strong positive magnitude, while a light crest has a strong negative magnitude. The geometric interpretation of the magnitude and its associated orientation is the direction where the crest is the sharpest. Similar to object contours, this direction is also the direction perpendicular to the line crest.

Customizing the edge extraction settings

Once you have allocated an Edge Finder context with the right context type (using **MedgeAlloc()**), you can customize the image processing algorithm using **MedgeControl()**. For example, to fit the requirements of your application, you can select the appropriate:

- Edge extraction filter (**M_FILTER_TYPE**).
- Filter mode (**M_FILTER_MODE**).
- Noise reduction factor (**M_FILTER_SMOOTHNESS**).
- Sensitivity of the edgel detection (**M_THRESHOLD_MODE**).
- Accuracy of the edgel positions (**M_ACCURACY**).

You can also choose to fill in the edge gaps, which is the process of linking broken edges together based on a set of criteria (**M_FILL_GAP_DISTANCE** and **M_FILL_GAP_ANGLE**).

Generally, the default control settings described in this section are sufficient for the majority of images. You should therefore adjust these settings only when dealing with very noisy images, extremely low or multi-contrast images, or images with very thin, refined features. In all cases, it is recommended that you try the default settings first and, if necessary, experiment with different settings to achieve the required edge map and accuracy.

For more advanced information on extracting edges, see the *Advanced edge extraction* section later in this chapter.

Note that the edge extraction process involves a denoising operation to even out rough edges and remove noise. The filter type, filter mode, kernel depth, smoothness factor, and threshold mode all work in concert to control this operation, and ultimately determine which edges to extract. Changing any one of these factors will, to varying degrees, alter the edges that are extracted.

Filter type

The edge extraction filter is used to compute the source image's derivatives, which in turn are used to calculate the edge's magnitude (Gradient or Hessian) and angle. You can set the type of filter used when performing the edge extraction using **MedgeControl()** with **M_FILTER_TYPE**. You can choose either an Infinite Impulse Response (IIR) edge extraction filter, or a Finite Impulse Response (FIR) edge extraction filter. The default setting is the IIR filter, **M_SHEN**.

IIR filters allow you to extract both crests and contours, while FIR filters only allow you to extract contours. IIR filters also allow you to take advantage of Edge Finder's smoothing capabilities, while FIR filters ignore your smoothness setting. For more information on smoothing, see the *Smoothing* subsection in this section.

Infinite Impulse Response (IIR) filters

The **M_FILTER_TYPE** setting in **MedgeControl()** establishes the contribution of each neighboring pixel. It provides you with the following IIR filters to extract edges:

- Shen-Castan (**M_SHEN**), which is an exponential weighting function, of the general form.

$$Ke^{-\beta|n|}$$

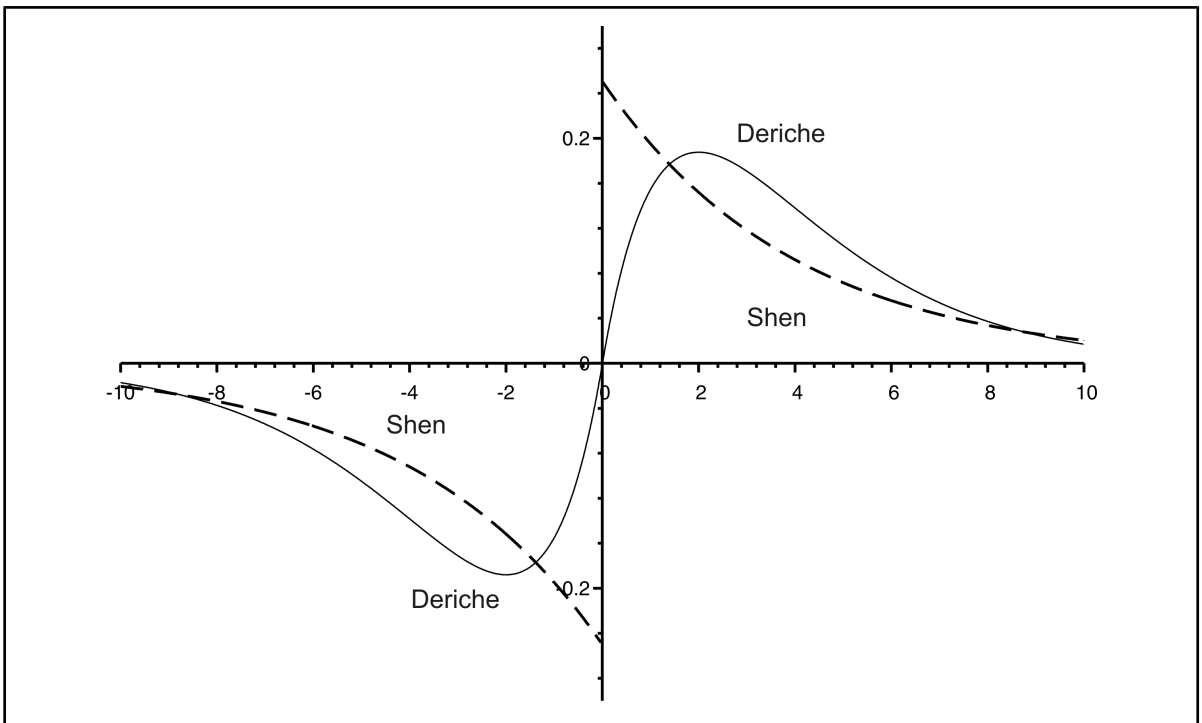
- Canny-Deriche (**M_DERICHE**), which is an exponential weighting function, of the general form.

$$k(a|n| + 1)e^{-a|n|}$$

Note that to perform the edge extraction, IIR filters can either be implemented in recursive or kernel mode. For more information on filter modes, see the *Filter mode* subsection in the *Customizing the edge extraction settings* section in *Chapter 9: Edge Finder*.

The default filter, Shen-Castan, typically performs an effective edge extraction; it achieves a very good localization of edges, which makes it ideal for the majority of images.

However, Shen-Castan can sometimes be unexpectedly sensitive to noise, or can yield inappropriate results if you are extracting unusually thick crests. If this is the case, you should try Canny-Deriché, which is better suited to handle these situations, since it places more emphasis on an edge's neighbors than Shen-Castan. The following image illustrates how the first derivative distribution of values can affect results for both Shen-Castan and Canny-Deriché:



Note that, as mentioned, IIR filters allow you to extract both crests and contours.

Finite Impulse Response (FIR) filters

The **M_FILTER_TYPE** setting in **MedgeControl()** provides you with the following FIR filters to extract edges:

- Frei Chen (**M_FREI_CHEN**), which can be represented with the following convolution kernels:

$$\begin{bmatrix} -1 & -\sqrt{2} & -1 \\ 0 & 0 & 0 \\ 1 & \sqrt{2} & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} -1 & 0 & 1 \\ -\sqrt{2} & 0 & \sqrt{2} \\ -1 & 0 & 1 \end{bmatrix}$$

- Prewitt (**M_PREWITT**), which can be represented with the following convolution kernels:

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

- Sobel (**M_SOBEL**), which can be represented with the following convolution kernels:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Note that, as mentioned, FIR filters only allow you to extract contours.

Filter mode

Filtering is a neighborhood calculation that is implemented during the edge extraction process. Depending on the filter, this calculation can either be implemented recursively, or using a convolution kernel. Since FIR filters use a specific 3x3 kernel defined by Edge Finder, you cannot calculate them recursively, nor can you alter their kernel size.

On the other hand, IIR filters use an exponential weighting function to perform the filtering. This function can either be implemented recursively, or with a kernel of different sizes. To specify the mode in which to perform the filtering, use **MedgeControl()** with **M_FILTER_MODE** set to either **M_RECURSIVE** or **M_KERNEL**.

Typically, the default filter mode is **M_RECURSIVE**. However, if you have dedicated hardware performing the operation, the default is **M_KERNEL**. Unless you have dedicated hardware, an **M_KERNEL** convolution should not be set since it can take a long time to process.

- ❖ If you do not have dedicated hardware, **M_RECURSIVE** is faster than **M_KERNEL**, unless you are using very small kernels (for example, 3x3). If you have dedicated hardware, **M_KERNEL** is typically faster.

If you specify **M_RECURSIVE**, the filtering is done using a recursive implementation of the IIR filter. If this mode actually used a kernel, the kernel size would be theoretically infinite.

If you specify **M_KERNEL**, the filtering is done using a kernel approximation of the IIR filter. In this case, you must set the size of the convolution kernel using **MedgeControl()** with **M_KERNEL_WIDTH**. The size can either be determined automatically or directly. Note that the size of the kernel can be constrained by the available hardware resources.

When using **M_KERNEL**, you must also specify the bit depth with which kernel values are calculated. To do so, set **M_KERNEL_DEPTH** to either 8 or 16. The default is 8-bit, which is faster than 16-bit, although it is less accurate.

Typically, **M_KERNEL_DEPTH** only affects Infinite Impulse Response (IIR) type filters, which you can set using **MedgeControl()** with **M_FILTER_TYPE**.

Automatically determined kernel size

To specify the size of the kernel automatically, set **MedgeControl()** with **M_KERNEL_WIDTH** to **M_AUTO**. This is the default setting. In this case, Edge Finder establishes the maximum possible kernel size, for a given smoothness factor. You can set the smoothness factor using **MedgeControl()** with **M_FILTER_SMOOTHNESS**.

The size of the kernel, for a given smoothness factor, is calculated as the size at which the quantified filter values are zero. If the hardware resources are available, the smoothness factor will always be adhered to. However, if the smoothness factor calls for a kernel size that is beyond the limits of your hardware, the smoothness factor will be sacrificed to achieve the largest possible kernel size.

Note that changing the smoothness factor can also change the convolution kernel size, which can affect processing time. That is, the larger the kernel, the longer the processing will take.

Directly determined kernel size

To specify the size of the kernel directly, set **MedgeControl()** with **M_KERNEL_WIDTH** to an odd integer that is greater than or equal to 3. Note that larger kernels result in longer processing times.

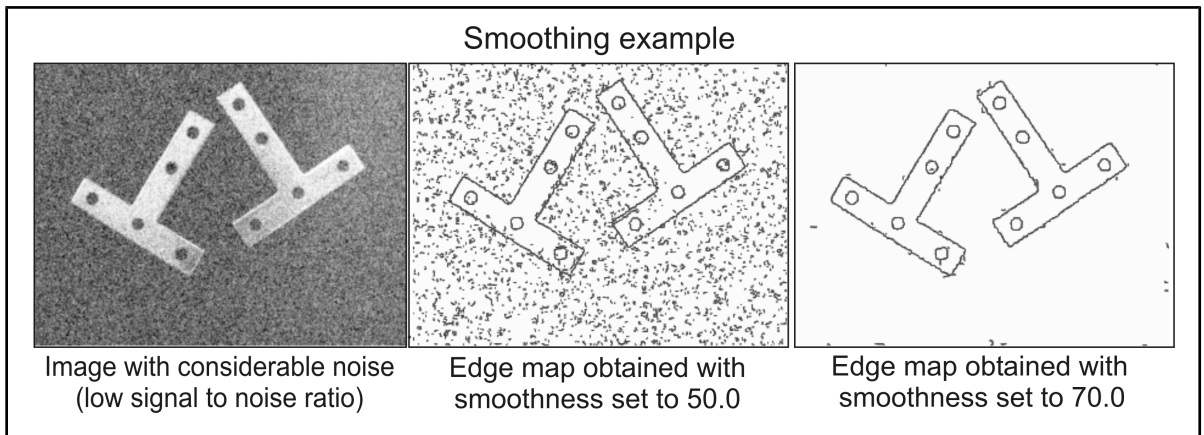
If your hardware cannot handle the kernel size, the effective smoothness factor will be saturated to the maximum possible value for the specified kernel size. That is, a filter is always made to fit in the kernel without a preimposed truncation; therefore, the best possible approximation is calculated, given the limited kernel size.

Smoothing

The **M_FILTER_SMOOTHNESS** setting in **MedgeControl()** allows you to control the degree of smoothness applied in the edge extraction. The smoothing operation evens out rough edges and removes noise; in effect, the smoothness factor affects which edges are extracted.

The range of this control varies from 0 (no smooth) to 100 (a very strong smooth). The default setting is 50.0. Note that if you are using an FIR filter, the smoothness value is ignored.

Increasing the smoothness control value does not affect the processing time. However, a very high smoothing level can result in a loss of important detail and a decrease in precision. Note that, the range of smoothing is not linear; that is, the higher the smoothing factor, the greater the difference between settings. For example, changing the smoothing from 30.0 to 50.0 is less significant than changing the smoothing from 50.0 to 70.0.



Note that **M_FILTER_SMOOTHNESS** is only relevant if **MedgeControl()** with **M_FILTER_TYPE** is set to **M_DERICHE** or **M_SHEN**.

Thresholding

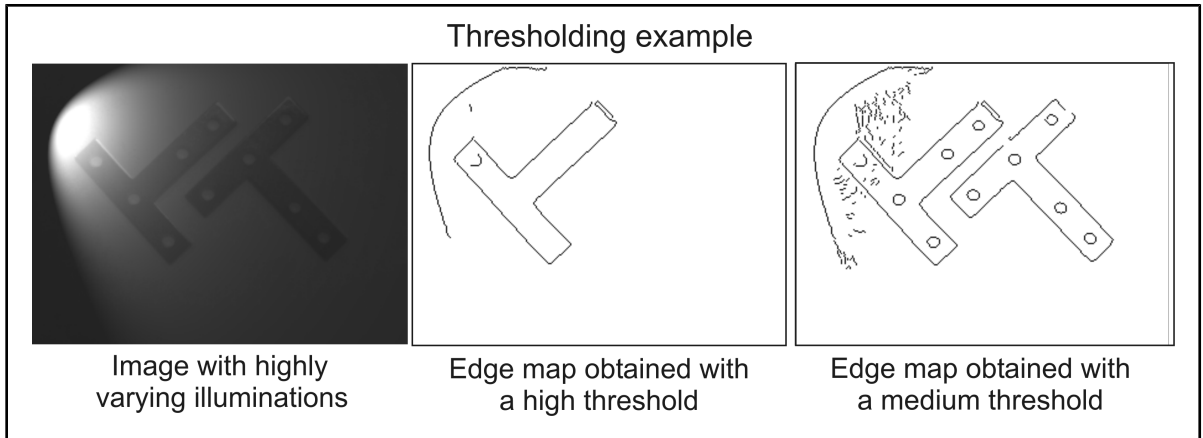
Edge Finder determines which edges are extracted from the source image based on a thresholding of the magnitude of each edgel. More specifically, an hysteresis thresholding is used to perform the edge extraction. With this type of thresholding, the extracted edge chains are built such that the magnitude values of all connected edgels are stronger than the lower bound threshold value (**M_THRESHOLD_LOW**) and at least one edgel in each edge chain has a magnitude that is stronger than the upper bound threshold value (**M_THRESHOLD_HIGH**). For more information on edgel magnitude, see the *Magnitude type* subsection in this section.

Thresholding can also be applied during post-calculation. In this case, threshold settings are applied to the Edge Finder result buffer, rather than the source image. Except for filtering operations, the edge extraction process is completely recalculated. As a result, all internal processing buffers (**M_SAVE_...**) must have been initially saved, using **MedgeControl()**. For more information on post-calculation, see the *Post-calculation* subsection in the *Calculating and retrieving results* section in *Chapter 9: Edge Finder*.

Edge Finder provides several predefined threshold modes that automatically select appropriate values for the lower and the upper bound thresholds, based on an internal image evaluation and noise estimation. To automatically determine the threshold settings with which to extract edges, set **MedgeControl()** with **M_THRESHOLD_MODE** to one of the following: **M_VERY_HIGH**, **M_HIGH**, **M_MEDIUM**, **M_LOW**, or **M_DISABLE**. The default setting is **M_HIGH**. Note that lower threshold modes result in a more sensitive edgel detection; that is, more edgels are detected. For example, **M_VERY_HIGH** always results in a lower (or equal) number of edgels than **M_HIGH**, **M_HIGH** always results in a lower (or equal) number of edgels than **M_MEDIUM**, and so on.

The default setting (**M_HIGH**) is typically sufficient since it offers a robust detection of pertinent edgels, even from images presenting some contrast variations, noise, and non-uniform illumination. However, for multi-contrast images, or for images with a lot of noise or non-uniform illumination, some edges can be missed. In these cases, the setting **M_MEDIUM** should be used. If all relevant edges are still not being extracted, then use **M_LOW**, which will get all edges over a minimum noise-based estimated threshold. Finally, to extract all edgels in the image, use **M_DISABLE**, which will set the lower and upper bound threshold values to 0. Note that **M_LOW** and **M_DISABLE** should be used carefully since a large number of unnecessary edgels might be extracted. For images that have

strong contrast, little noise, and consistent illumination, the **M_VERY_HIGH** setting can be used, since it will extract only the strongest edges. Note that **M_VERY_HIGH** should also be used carefully since pertinent edgels might not be extracted.



In the majority of cases, Edge Finder's predefined threshold modes should provide you with sufficient thresholding control. However, for advanced applications, you might want to explicitly set the threshold bounds. To do so, set **M_THRESHOLD_MODE** to **M_USER_DEFINED**, and use the **M_THRESHOLD_LOW** and **M_THRESHOLD_HIGH** control types to specify the lower and upper threshold bounds, respectively. The default settings are 0.

- ❖ If **M_THRESHOLD_MODE** is not set to **M_USER_DEFINED**, your **M_THRESHOLD_LOW** and **M_THRESHOLD_HIGH** values have no effect.

Note that the threshold bounds must be provided as positive values. However, for line crests, Edge Finder will consider these values to be negative and/or positive, depending on whether the edges were extracted as darker (negative edgel values), lighter (positive edgel values), or both darker and lighter (negative and positive edgel values) than the image's background. For more information on extracting line crests, see the *Object contours versus line crests* subsection in the *Extracting the edges* section in *Chapter 9: Edge Finder*.

Advanced users might want to customize the behavior of the hysteresis thresholding; to do so, see the *Advanced thresholding* subsection in the *Advanced edge extraction* section in *Chapter 9: Edge Finder*.

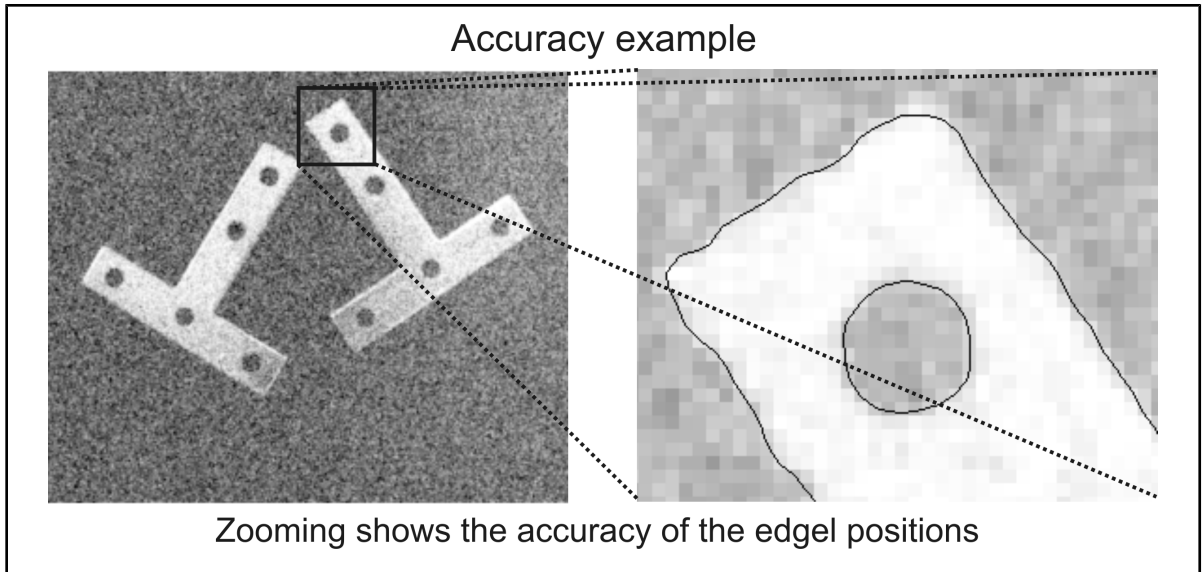
Note that when interfacing with the MIL Geometric Model Finder module, you should use **MedgeControl()** with **M_DETAIL_LEVEL** to set the threshold mode. Note that an **M_DETAIL_LEVEL** setting overrides an **M_THRESHOLD_MODE** setting. For more information, see the *Interfacing with Geometric Model Finder* section later in this chapter.

Edgel accuracy

The **M_ACCURACY** setting in **MedgeControl()** determines the accuracy used to estimate edgel positions. Edgel accuracy can be set to one of the following: **M_HIGH**, **M_VERY_HIGH**, or **M_DISABLE**. The default setting is **M_HIGH**. When you set the accuracy to high, edgels are calculated with subpixel accuracy, which is typically sufficient. When you set the accuracy to very high, a mathematical model is used to compensate for the deformation introduced by the square shape of each pixel, resulting in very precise subpixel edge accuracy. If you want to disable subpixel accuracy and calculate edgels with pixel precision, set **M_ACCURACY** to **M_DISABLE**.

For the most part, edgel accuracy varies with the image's dynamic range, sharpness, and noise. The best accuracy is achieved in well-contrasted, noise-free images. In perfect situations, Edge Finder can provide a subpixel edgel location accuracy of up to 1/128th of a pixel. However, in real situations, it is reasonable to expect an accuracy from 1/10th of a pixel to 1/40th of a pixel, depending on physical limitations due to the transition's dynamic range and the image's noise.

You can notice the accuracy of the edgels if you zoom a region of the source image that was used to calculate results by specifying the appropriate **M_DRAW_RELATIVE_ORIGIN_...** and **M_DRAW_SCALE_...** values with **MedgeControl()**.



Magnitude type

You can specify the type of magnitude used to determine edgel positions using **MedgeControl()** with **M_MAGNITUDE_TYPE**. The magnitude type can be set to either **M_NORM**, where the gradient magnitude is used, or **M_SQR_NORM**, where the square of the gradient magnitude is used. The default setting is **M_SQR_NORM**.

Since **M_SQR_NORM** uses the square of the gradient magnitude, it is faster, though less precise than **M_NORM**. Typically, **M_SQR_NORM** is used since it preserves a very good edgel location accuracy. Note that changing **M_MAGNITUDE_TYPE** can slightly modify the resulting edge map.

Filling the edge gaps

Edge Finder allows you to fill gaps between edges; that is, to automatically link the extremities of non-closed edges together, depending on their relative positions and orientations. Unexpected broken edges can occur when dealing with noisy images, and connecting them can significantly improve the quality of your results. To help you choose which edges can be linked, Edge Finder provides a series of constraints, described below. Note that edges will only be linked if they adhere to each constraint.

- ❖ Filling edge gaps can also be done in post-calculation; however, there are some restrictions that must be taken into account. For more information, see the *Post-calculation* subsection in the *Calculating and retrieving results* section in *Chapter 9: Edge Finder*.

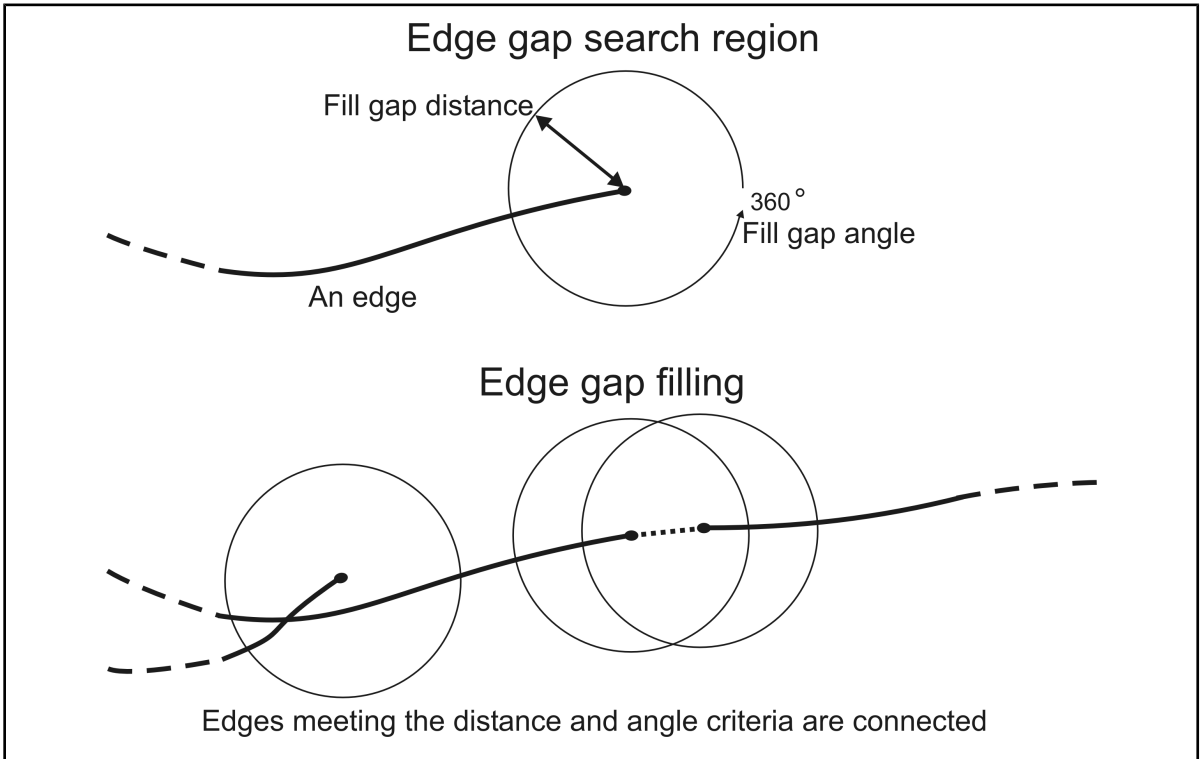
Candidate constraint

You can choose whether to join an edge extremity with the other extremity of the same edge, or with an extremity of any edge. To connect an edge's extremity with the extremity of any edge, use **MedgeControl()** with **M_FILL_GAP_CANDIDATE** set to **M_ANY**. This is the default value.

To connect the edge extremities of the same edge, use **MedgeControl()** with **M_FILL_GAP_CANDIDATE** set to **M_SAME**. This is the same as closing an open edge.

Region constraint

The **M_FILL_GAP_DISTANCE** and **M_FILL_GAP_ANGLE** settings in **MedgeControl()** define the region where two edge extremities can be linked. That is, when searching for edge extremity candidates to fill edge gaps, **M_FILL_GAP_DISTANCE** sets the maximum distance radius, while **M_FILL_GAP_ANGLE** sets the aperture angle. Note that two edge extremities can only be linked if both meet the **M_FILL_GAP_DISTANCE** and **M_FILL_GAP_ANGLE** criteria, and if both are included in the search region of the other.



The gap distance (**M_FILL_GAP_DISTANCE**) must be specified in pixel units. The gap distance should be set carefully and large values should generally be avoided, otherwise the wrong edges can be connected. Typically, distance values should not exceed 5 pixels; however, if necessary, you can also specify no distance restraint (**M_INFINITE**). The default **M_FILL_GAP_DISTANCE** value is 0 (no edge linking) and the default **M_FILL_GAP_ANGLE** value is 360 degrees.

Polarity constraint

When searching for edge extremity candidates, you can also use the **M_FILL_GAP_POLARITY** control type to specify that only edges with a certain polarity can be linked. The edge's polarity indicates whether edges were established as a transition from light to dark, or vice versa. By default (**M_ANY**), all edges that meet the distance and angle restraints are considered candidates, regardless of the

edge's polarity. If **M_FILL_GAP_POLARITY** is set to **M_SAME**, only edges with the same polarity are considered potential candidates. If **M_FILL_GAP_POLARITY** is set to **M_REVERSE**, only edges with reverse polarity are considered potential candidates.

Note that the **M_SAME** and **M_REVERSE** polarity settings are not available for **M_CREST** Edge Finder contexts.

Continuity constraint

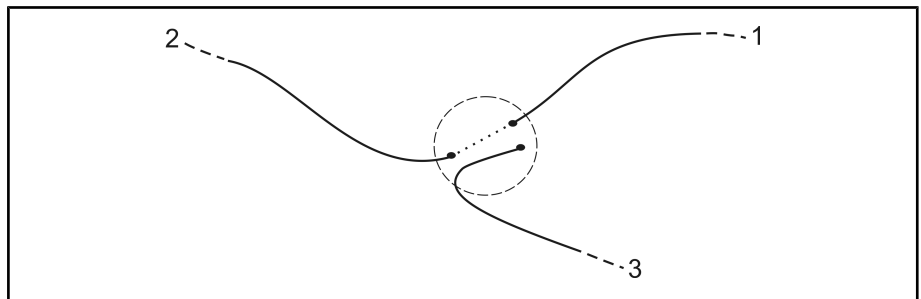
It is possible that multiple edges meet the distance (**M_FILL_GAP_DISTANCE**), angle (**M_FILL_GAP_ANGLE**), and polarity (**M_FILL_GAP_POLARITY**) settings. When this occurs, you must decide which edges should be linked. To help choose, Edge Finder provides a continuity constraint.

The continuity constraint allows you to decide whether the candidate chosen to fill the edge gap is selected according to its proximity or its maximum continuity.

To set the continuity constraint, use **MedgeControl()** with

M_FILL_GAP_CONTINUITY. The range of this control varies from 0 to 100.

When set to 0, the closest edge extremity is chosen to link edges together. When set to 100, the edge that gives the most continuous result (minimum curvature) is chosen. The default value is 50. In the following example, a candidate must be chosen to fill the gap with edge 1. If the continuity constraint is set to 100, edge 2 will be selected instead of edge 3; even though edge 3 is closer, edge 2 is more continuous.



Edge features

Edge Finder allows you to calculate many edge features that can provide useful information, such as the edge's length and center of gravity. To enable (**M_ENABLE**) or disable (**M_DISABLE**) the calculation of a specific edge feature, use **MedgeControl()**; you can also select a group of features for calculation in a single call. To know the calculation state of a feature, use **MedgeInquire()**. To retrieve the results of the feature calculation, use **MedgeGetResult()**. For more information on retrieving results, see the *Retrieving the results* subsection in the *Calculating and retrieving results* section in *Chapter 9: Edge Finder*.

Note that the label feature (**M_LABEL_VALUE**) is the only edge feature that is enabled by default. The label value is a positive integer greater or equal to one; it is assigned to each edge in an image for identification. Each edge in an image has a unique label value. The label allows you to track or choose particular edges for drawing, edge manipulation, or result retrieval purposes. Typically, you should not disable the edge's label value.

Edge features generally fall into one of the following groups:

- Dimension features (typically very fast to compute).
- Location features (typically fast to compute).
- Advanced features (can take some time to compute, due to their complexity).

To compute features as efficiently as possible, you should calculate a few features first (preferably, the fastest), and eliminate as many unnecessary edges as possible. Then, post-calculate expensive features on the remaining edges. For more information on how to select and post-calculate edges, see the *Calculating and retrieving results* section later in this chapter.

Note that occasionally, enabling a feature for calculation will result in another feature being calculated and available for result retrieval. For example, enabling **M_MOMENT_ELONGATION** for calculation in **MedgeControl()** will allow you to retrieve the result of **M_CENTER_OF_GRAVITY** in **MedgeGetResult()**, even if you have disabled the calculation of the center of gravity.

The following subsections list all the Edge Finder features that you can enable or disable for calculation using **MedgeControl()**, according to dimension, location, and advanced application. There is also a subsection on selecting a group of features for calculation in a single call. Note that only the more complicated features are explained in some detail. For a description of each feature, see **MedgeControl()** in the MIL Reference.

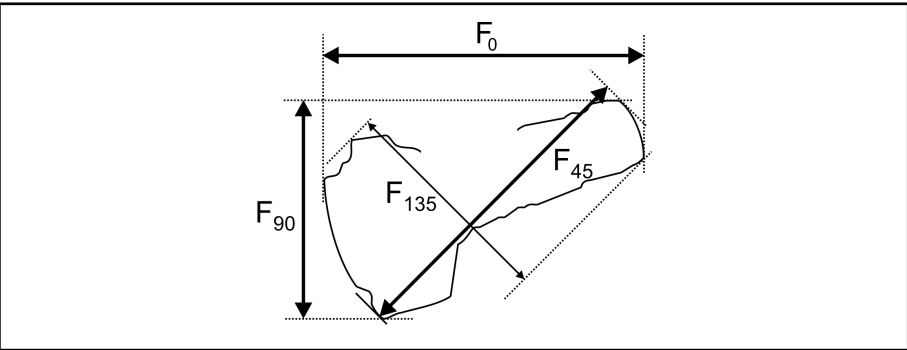
Dimension features

The Edge Finder dimension features are:

M_CLOSURE	M_ELLIPSE_FIT_MAJOR_AXIS	M_FERET_X
M_CIRCLE_FIT_CENTER_X	M_ELLIPSE_FIT_MINOR_AXIS	M_FERET_Y
M_CIRCLE_FIT_CENTER_Y	M_FAST_LENGTH	M_GENERAL_FERET
M_CIRCLE_FIT_COVERAGE	M_FERET_BOX	M_GENERAL_FERET_ANGLE
M_CIRCLE_FIT_ERROR	M_FERET_ELONGATION	M_LENGTH
M_CIRCLE_FIT_RADIUS	M_FERET_MAX_ANGLE	M_LINE_FIT_A
M_ELLIPSE_FIT_ANGLE	M_FERET_MAX_DIAMETER	M_LINE_FIT_B
M_ELLIPSE_FIT_CENTER_X	M_FERET_MEAN_DIAMETER	M_LINE_FIT_C
M_ELLIPSE_FIT_CENTER_Y	M_FERET_MIN_ANGLE	M_LINE_FIT_ERROR
M_ELLIPSE_FIT_COVERAGE	M_FERET_MIN_DIAMETER	M_SIZE
M_ELLIPSE_FIT_ERROR		

Feret

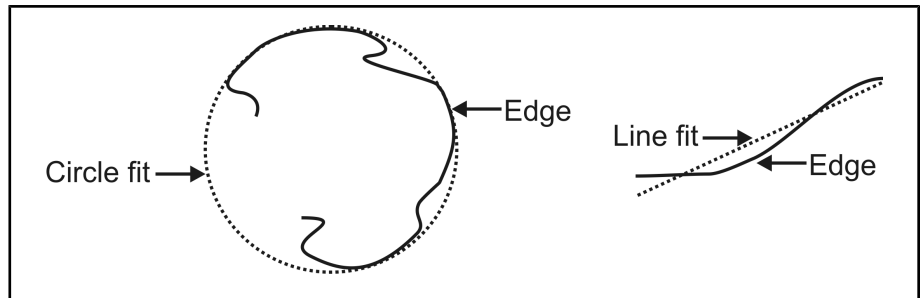
The Feret is the diameter of the edge at various angles. Several Feret diameters are illustrated in the diagram below. Note that the angle at which the Feret diameter is taken (relative to the horizontal axis) is specified as a subscript to F.



Some features, such as the edge's maximum Feret diameter, are determined by testing the Feret diameter of the edge at several angles. You can specify the angular region used for calculating the Feret features by using **MedgeControl()** with **M_FERET_ANGLE_SEARCH_MAX** and **M_FERET_ANGLE_SEARCH_MIN**. The search will be done in the counter-clockwise direction. To set the number of angles, use **MedgeControl()** with **M_NUMBER_OF_FERETS**. The default number of angles tested is 8, which is typically sufficient. Note that increasing the number of Ferets increases the accuracy of the results; however, it also increases the processing time.

Circle fit and line fit

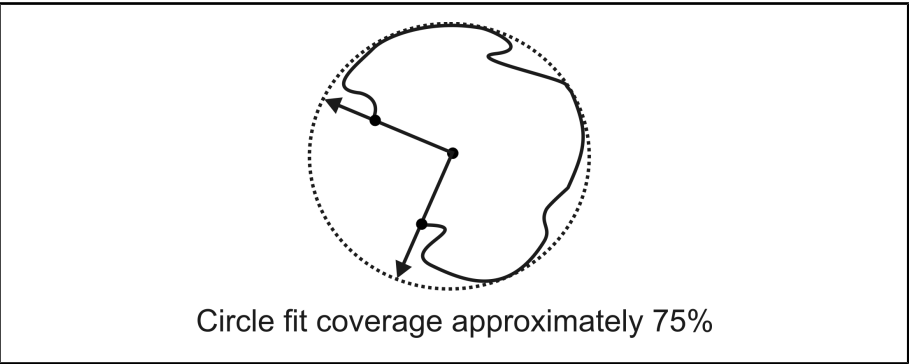
The circle fit is calculated as the circle that best fits the edge, while the line fit is calculated as the line that best fits the edge.



To calculate the coordinates of the center of the circle that is the best fit for each edge, use **M_CIRCLE_FIT_CENTER_X** and **M_CIRCLE_FIT_CENTER_Y**. To calculate the radius of the circle that is the best fit for each edge, use **M_CIRCLE_FIT_RADIUS**.

To calculate the A , B , and C variables that define the line that is the best fit for each edge, use **M_LINE_FIT_A**, **M_LINE_FIT_B**, and **M_LINE_FIT_C**. Note that these variables are based on the linear equation, $Ax + By + C = 0$.

To calculate the coverage of the circle fit, use **M_CIRCLE_FIT_COVERAGE**. The coverage indicates what angular fraction of the fitted circle is subtended by the radii going to the endpoints of the edge. The value returned is between 0.0 and 1.0, inclusive. For example, if the edge is closed, the coverage is 1.0 (100% coverage), for a quarter circle the coverage is 0.25 (25% coverage).

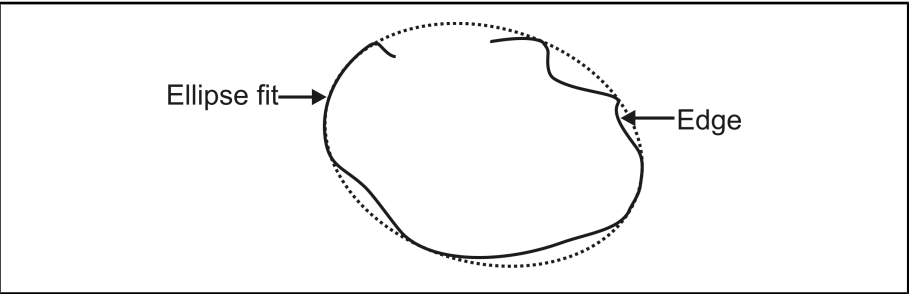


Note that since a line is theoretically infinite, it is impractical to calculate its coverage.

You can also calculate the error of either the circle fit or line fit using **M_CIRCLE_FIT_ERROR** or **M_LINE_FIT_ERROR**, respectively. These values are calculated as the average quadratic error of the fit.

Ellipse fit

The ellipse fit is calculated as the ellipse that best fits the edge.



To calculate the coordinates of the center of the ellipse that is the best fit for each edge, use **M_ELLIPSE_FIT_CENTER_X** and **M_ELLIPSE_FIT_CENTER_Y**. To calculate the major and minor axes of the ellipse that is the best fit for each edge, use **M_ELLIPSE_FIT_MAJOR_AXIS** and **M_ELLIPSE_FIT_MINOR_AXIS**, respectively.

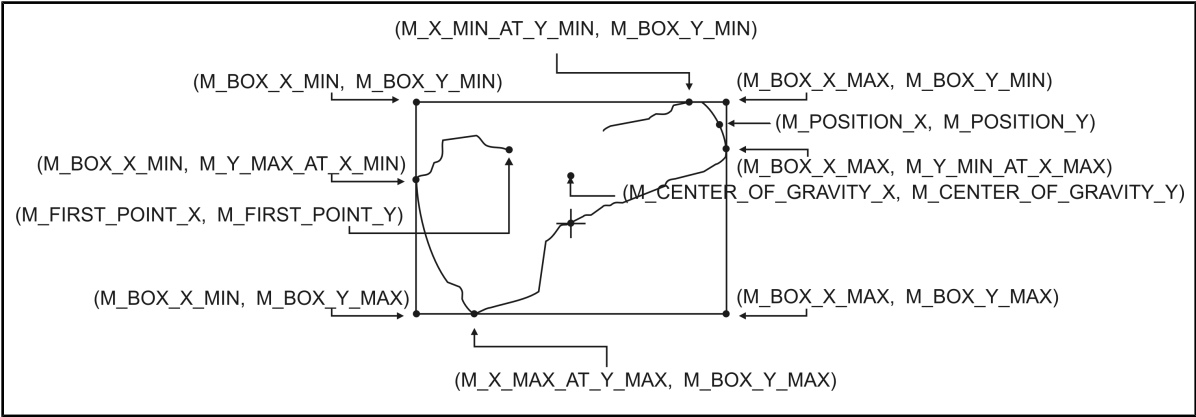
To calculate the angle of the ellipse fit, use **M_ELLIPSE_FIT_ANGLE**. To calculate the error of the ellipse fit, use **M_ELLIPSE_FIT_ERROR**. This value is calculated as the average quadratic error of the fit.

Location features

The Edge Finder location features are:

M_BOX	M_CENTER_OF_GRAVITY_X	M_POSITION_X
M_BOX_X_MAX	M_CENTER_OF_GRAVITY_Y	M_POSITION_Y
M_BOX_X_MIN	M_CONTACT_POINTS	M_X_MAX_AT_Y_MAX
M_BOX_Y_MAX	M_FIRST_POINT	M_Y_MAX_AT_X_MIN
M_BOX_Y_MIN	M_FIRST_POINT_X	M_X_MIN_AT_Y_MIN
M_CENTER_OF_GRAVITY	M_FIRST_POINT_Y	M_Y_MIN_AT_X_MAX

The following edge map illustrates where each edge location feature is located:



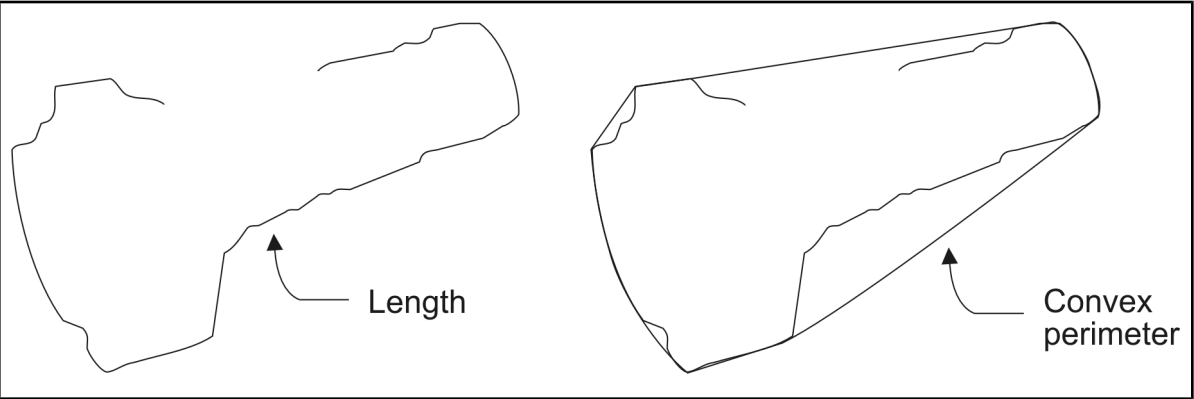
Advanced features

The Edge Finder advanced features are:

M_AVERAGE_STRENGTH	M_MOMENT_ELONGATION	M_STRENGTH
M_CONVEX_PERIMETER	M_MOMENT_ELONGATION_ANGLE	M_TORTUOSITY

Convex perimeter

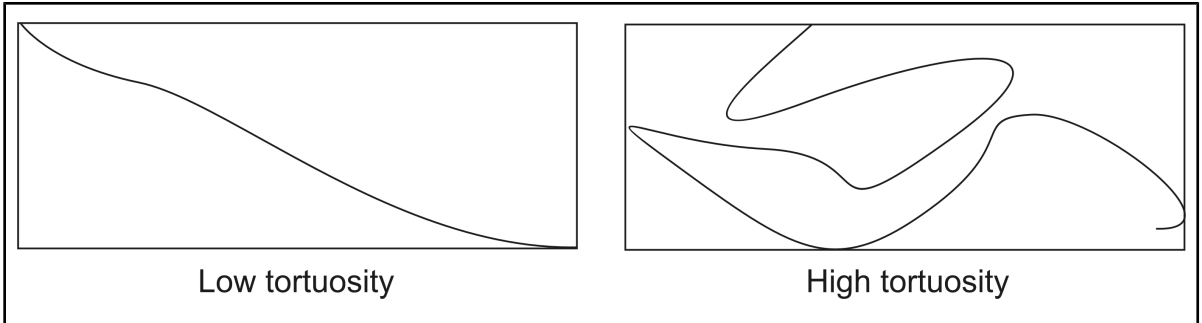
If you enable the convex perimeter feature (**M_CONVEX_PERIMETER**), an approximation of the perimeter of each edge's convex hull will be calculated. Abstractly, an edge's convex perimeter is very much like taking a rubber band, and placing it tautly around the edge.



The convex perimeter is derived by taking the diameter of the edge at different angles. The greater the number of Ferets used to calculate the diameter, the more accurate the approximation. Use **MedgeControl()** with **M_NUMBER_OF_FERETS** to adjust the number of Ferets. Note that increasing the number of Ferets increases the accuracy of the results; however, it also increases the processing time.

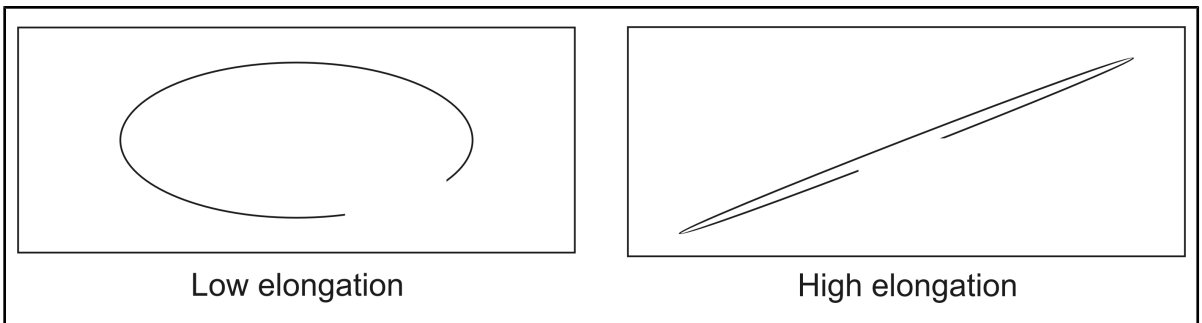
Tortuosity

An edge's tortuosity (**M_TORTUOSITY**) is the diagonal length of the edge's bounding box (**M_BOX**), divided by the length of the edge (**M_LENGTH**). Therefore, a non-tortuous edge (a straight line) will have a tortuosity of 1.0, while a tortuous edge will have its tortuosity decreasing towards zero.



Moment elongation

The moment elongation (**M_MOMENT_ELONGATION**) angle is a geometric elongation measure of the edge. It is defined as the ratio of the principal values of the edge's inertial matrix, which corresponds to the principal directions of the edge shape. This can be approximately defined as the ratio between the edge's minimum and maximum moment. The elongation value varies from 0.0 to 1.0. Values closer to 1.0 are considered to have a low elongation, while values closer to 0.0 are considered to have a high elongation.



Note that you can also calculate the angle of the principal axis along each edge's moment elongation, using `M_MOMENT_ELONGATION_ANGLE`.

Grouped features

The Edge Finder features that allow you to select a group of features for calculation in a single call are:

<code>M_ALL_FEATURES</code>	<code>M_ELLIPSE_FIT</code>
<code>M_BOX</code>	<code>M_FERET_BOX</code>
<code>M_CENTER_OF_GRAVITY</code>	<code>M_FIRST_POINT</code>
<code>M_CIRCLE_FIT</code>	<code>M_LINE_FIT</code>
<code>M_CONTACT_POINTS</code>	<code>M_POSITION</code>

Each of these features has multiple features associated with it. Therefore, if you enable one of them for calculation, then all the features associated with it will also be enabled. For example, enabling `M_CIRCLE_FIT` will result in all the circle fit values of each edge to be calculated. This is equivalent to individually enabling each of the following: `M_CIRCLE_FIT_CENTER_X`, `M_CIRCLE_FIT_CENTER_Y`, `M_CIRCLE_FIT_RADIUS`, `M_CIRCLE_FIT_ERROR`, and `M_CIRCLE_FIT_COVERAGE`.

The features that belong to each group is typically self evident. For a description of each, see `MedgeControl()`.

Calculating and retrieving results

Once you have allocated an Edge Finder context and result buffer (**MedgeAlloc()** and **MedgeAllocResult()**), and have calculated the edge features (**MedgeCalculate()**), you can retrieve the results from your Edge Finder result buffer, using **MedgeGetResult()**.

After the initial calculation, you can select edges that meet a specified criterion with **MedgeSelect()**, and post-calculate new features for them. By doing so, you can avoid unnecessarily calculating time-consuming edge features for all edges in a source image.

For optimum performance, the source image buffer on which you make calculations should be a 1-band 8-bit unsigned buffer. Other buffer depths and types are generally accepted, but can slightly decrease performance. 3-band source image buffers are also supported, but only when extracting edge contours. When the source image buffer is a 32-bit floating-point buffer, Edge Finder uses floating-point precision calculations. In all cases, you can force the edge processing operations to use floating-point precision by enabling **M_FLOAT_MODE** in **MedgeControl()**. Note that changing this processing mode can slightly affect the resulting edge map; also, floating-point precision calculations will typically take more time.

Occasionally, an operation can take an unexpectedly long time to calculate. To prevent this, you can use **MedgeControl()** with **M_TIMEOUT** to set a maximum edge extraction and calculation time. By default, there is no time limit.

Sorting keys

The results obtained from **MedgeGetResult()** 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, add **M_SORTn_UP** or **M_SORTn_DOWN** to the feature when selecting it for calculation using **MedgeControl()**. Assign the numbers 1, 2, or 3 to indicate the sorting precedence of the feature(s). Note that features will only be sorted after an **MedgeCalculate()**.

You can also post-sort edges by assigning a sorting key to a previously calculated edge feature. Note that, although the post-calculation changes the order in which the edges are sorted, it does not change their label values, which were assigned at the edges' initial calculation. For more information on post-calculation, see the *Post-calculation* subsection in this section.

Retrieving the results

Edge Finder offers several types of results that provide considerable information on the nature of the extracted edges, such as magnitude and Feret values. This information can be retrieved for all edges or for a specified edge, from your result buffer.

Typically, before retrieving any edge feature result, you should first retrieve the number of edges found in the image to determine the size of the result array needed to hold the results. To do so, use **MedgeGetResult()** with **M_NUMBER_OF_CHAINS**. Similarly, if you need to retrieve edgel results, such as the edgels' coordinates, magnitude, or angle values, you should also determine the size of the result array needed to hold the results. To do so, you must retrieve the number of extracted edgels for each edge using **MedgeGetResult()** with **M_NUMBER_OF_CHAINED_EDGELS**.

Edge results are indexed as positive integers starting at 0 and, if applicable, are ordered with respect to the sorting keys. You can also label edges by enabling **M_LABEL_VALUE** in **MedgeControl()** (enabled by default). The edge's label value is a positive integer greater than or equal to 1. Each edge in an image is given a unique label value; that is, unlike an edge's index value, an edge's label value will not change, regardless of future operations.

Edge Finder allows you to retrieve results for a particular edge or for all edges, using **MedgeGetResult()**. To retrieve results for a particular edge, you must specify either the edge's index or label value. However, since an edge's index value can change, it is recommended that you specify the label of the edge, especially if you want to perform selection and post-calculation operations. To retrieve results for all edges, you must specify **M_ALL** as the edge's index or label value.

Note that before an edge feature can be returned with **MedgeGetResult()**, you must first enable the feature for calculation, using **MedgeControl()**, and then call **MedgeCalculate()**. Every edge feature, except for the label value (**M_LABEL_VALUE**), is disabled by default. For more information on edge features, see the *Edge features* section earlier in this chapter.

To verify the availability of an edge feature (if it has been calculated), use **MedgeGetResult()**, and combine **M_AVAILABLE** to the specified result type. For a complete description of all possible results, see **MedgeGetResult()** in the MIL Reference.

In general, results are returned in pixels, and coordinates are relative to the center of the top-left pixel in the source image. If you calculate edges using a calibrated source image, results are automatically returned in the output units specified by the associated calibration object of the source image. However, you can change the output units to pixel units or world units by setting **McalControl()** with **M_OUTPUT_UNITS** to either **M_PIXEL** or **M_WORLD**, respectively. For more information, see Calibration.

Note that if your image is calibrated, results are calculated in the real world; therefore, retrieving results in pixel units instead of world units will typically be slower. Also note that, in the presence of distortion, some results are meaningless when converted from real-world to pixel units (for example, the Feret angles). For example, if an edge appears warped in the source image, but the calibration object of the source image compensates for this during the extraction, the resulting Feret angles are meaningful in the real-world coordinate system, and meaningless in the image coordinate system.

Selecting the results

If you do not have many unwanted edges, it is usually faster to simply calculate all required features for all edges. However, calculating many features on a large number of unwanted edges can be unnecessarily time-consuming. To speed up this process, you can use **MedgeSelect()** to select a subset of edges for calculation and result retrieval.

Edges that meet the **MedgeSelect()** selection criteria can be included, excluded, or deleted from the result buffer. Further calculations are subsequently applied to included edges only. Excluded edges can be re-included at any time, while deleted edges are permanently removed from the result buffer.

Typically, if you have many unwanted edges, you calculate (for all edges) only those features that allow you to distinguish between wanted and unwanted edges. Then, you exclude the unwanted edges from further calculations, and calculate all the required features for the remaining included edges. To arrive at the required set of results, you can make as many calls as necessary to **MedgeSelect()** and **MedgeCalculate()**.

Any calculation made to refine results after an initial call to **MedgeCalculate()** is considered a post-calculation. Post-calculating edges can be an effective way of speeding up processing time, especially if advanced features are only calculated on a small subset of selected edges. For more information on post-calculation, see the *Post-calculation* subsection in this section.

Note that, if at any time you calculate edges in a new source image, all current results will be discarded and replaced by the newly calculated edges. In addition, all selected features will be recalculated for all edges in the new source image. This means that you will have to restart the selection procedure.

The **MedgeSelect()** function can be used to select edges based on one of the following:

- A calculated edge feature, where the edge selection depends on whether the specified edge feature meets the specified condition.
- The inter-relationship of edges, where the edge selection depends on whether edges meet the specified box or chain condition of the specified edge or group of edges.

- The current status of edges, where the edge selection depends on a specific edge, all edges, all included edges, or all excluded edges. These will be included, included only, excluded, excluded only, or deleted. For example, you can include only (**M_INCLUDE_ONLY**) the excluded edges (**M_EXCLUDED_EDGES**), which will essentially swap the previously included and excluded edges.
- The proximity of an edge or edges to a specified point, where the edge selection depends on the specified radius, and the nearest neighbor condition.

Note that you can also use **MedgeSelect()** to crop and select a portion of a specified edge. For more information, see the *Advanced edge extraction* section later in this chapter.

If the edges have been calculated using a calibrated source image, you must specify relevant values in the real world. If the edges have not been calculated using a calibrated source image, you must specify relevant values in pixels. Note that you can change the output units using **McalControl()** with **M_OUTPUT_UNITS** set to **M_PIXEL** or **M_WORLD**.

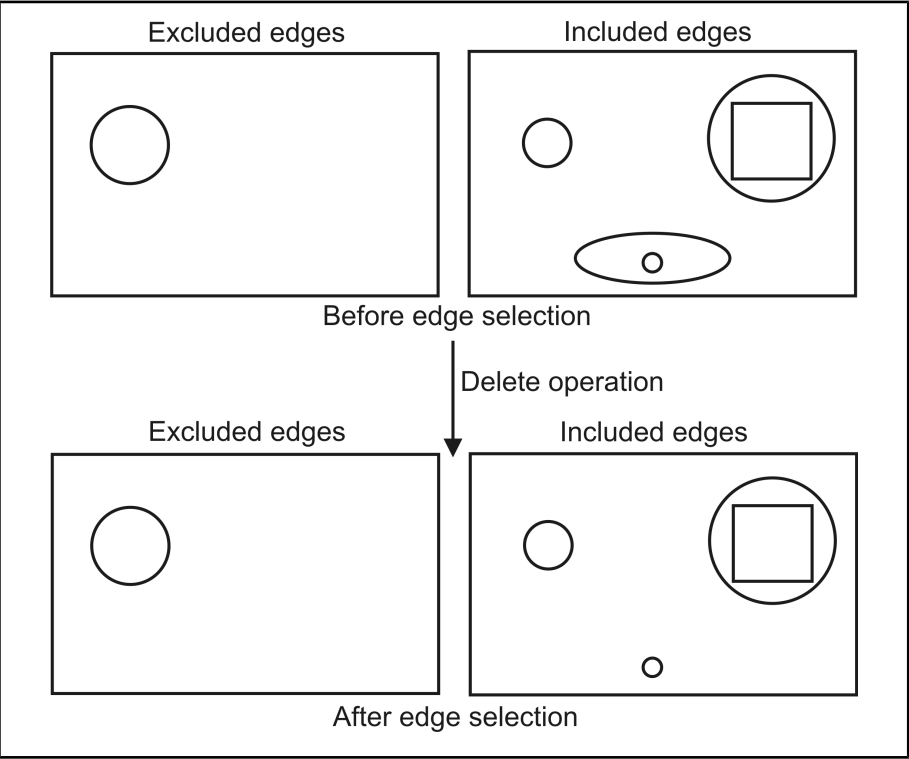
Edge selection based on edge features

You can include, exclude, or delete edges based on whether the specified edge feature meets the specified condition. The edge feature used to select the edges must have been previously calculated for the included edges. After several selection and post-calculation operations, an edge feature might not be available or might only be available for a few edges of the included subset. To verify the availability of an edge feature, use **MedgeGetResult()**, and combine **M_AVAILABLE** to the specified feature result type.

The following code shows you how to apply a selection based on edge features:

```
MedgeSelect(MilResult, M_DELETE, M_MOMENT_ELONGATION, M_LESS, 0.8, M_NULL);
```

In this example, all edges with a moment elongation that is less than 0.8 are deleted.



Edge selection based on the inter-relationship of edges

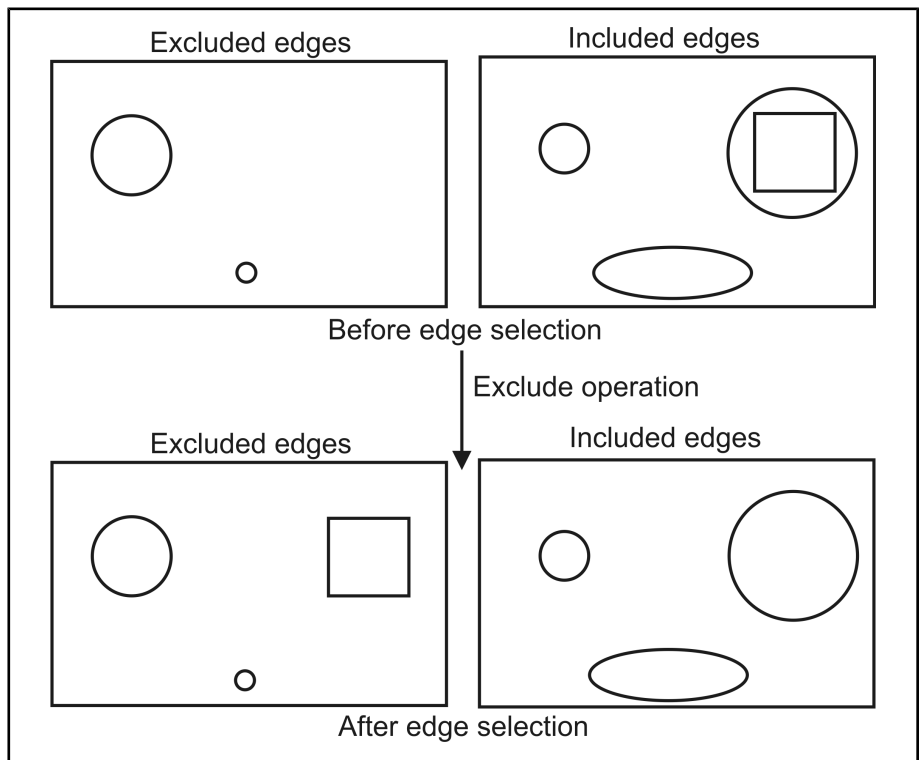
You can include, exclude, or delete edges based on whether edges meet the specified box or chain condition. That is, you can include, exclude, or delete all edges that are inside or outside the bounding box of a specified edge or group of edges; similarly, you can include, exclude, or delete all edges that are inside or outside a specified edge or group of edges.

- ❖ Inside and outside chain conditions only take effect when the edge is closed.

The following code shows you how to apply a selection based on the inter-relationship of edges:

```
MedgeSelect(MilResult, M_EXCLUDE, M_INCLUDED_EDGES, M_INSIDE_CHAIN,
            M_NULL, M_NULL);
```

In this example, all edges inside included edges are excluded.



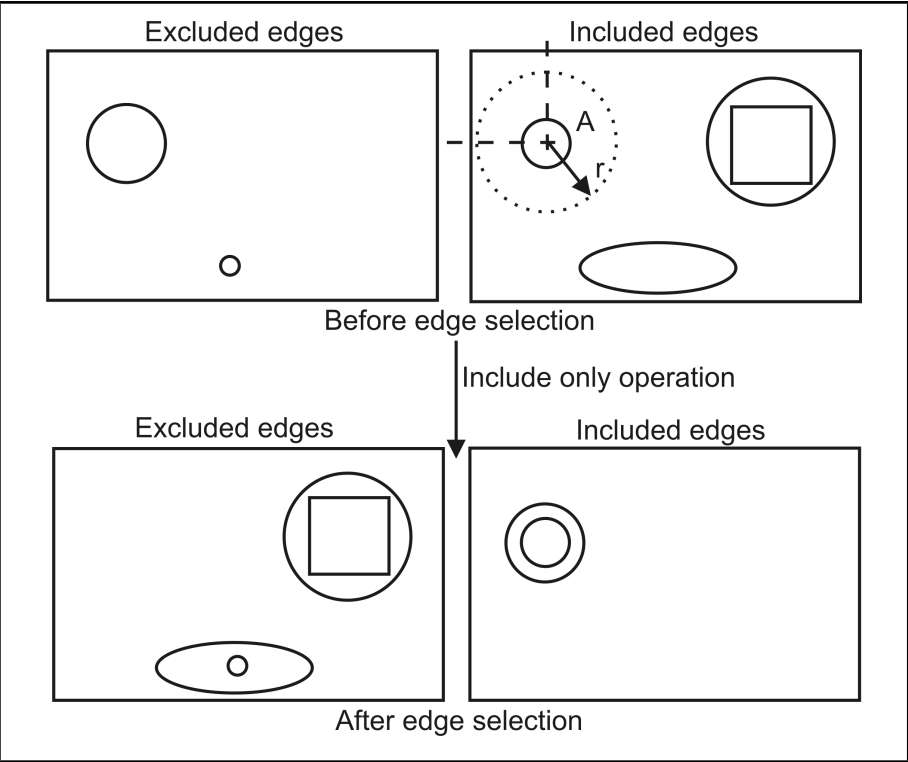
Edge selection based on the proximity of the edges to a point

You can include, exclude, or delete edges based on their proximity to a specified point. You can select either the closest edge to a point, or all the edges in the specified vicinity of a point. Use **MedgeControl()** with **M_NEAREST_NEIGHBOR_RADIUS** to specify the maximum radius distance from the point.

The following code shows you how to apply a selection based on the proximity of the edges to a point:

```
MedgeSelect(MilResult, M_INCLUDE_ONLY, M_NULL, M_ALL_NEAREST_NEIGHBORS, Ax, Ay);
```

In this example, only the edges closest to the point A (Ax, Ay), and within the radius r, are included.



Internal processing buffers

When performing edge extraction, Edge Finder uses internal processing buffers. These can be saved in the Edge Finder result buffer. The internal processing buffers are used when extracting edges. They include image derivative, angle, magnitude, source image, and mask buffers.

Some results can only be post-calculated if the appropriate internal buffer(s) have been previously saved in the result buffer. For example, you can only post-calculate the strength of an edge feature if the internal magnitude buffer was initially saved. For more information, see the *Post-calculation* subsection in this section.

Using **MedgeControl()**, you can set whether the internal processing buffers are saved. By default, the internal processing buffers are not saved (**M_DISABLE**). You must enable saving them in the result buffer, using **MedgeControl()**, before the first call to **MedgeCalculate()**. Saving these buffers does not affect processing time.

Even if an internal processing buffer has been saved in the result buffer, you cannot access it directly. You can, however, allocate a new buffer with the same properties as the internal buffer and use **MedgeDraw()** to copy the contents of the internal buffer into the newly allocated buffer. To copy an internal result buffer you must:

1. Use **MedgeGetResult()** to get the required information from the internal buffer. You will need the following information about the internal buffer to be cloned:
 - The buffer range and bit depth of the buffer (add **M_SIGN** or **M_SIZE_BIT** to the internal buffer, respectively). Alternatively you can add **M_TYPE**, which returns both.
 - The width and height of the buffer (add **M_SIZE_X** and **M_SIZE_Y** to the internal buffer).
2. Allocate your own buffer, which should have the same properties as the internal buffer you want to copy.
3. Use **MedgeDraw()** to copy the contents of the internal buffer into your own buffer.

Derivatives

You can save the internal derivative buffers used when extracting edges, in the Edge Finder result buffer. To save the buffers, use **MedgeControl()** with **M_SAVE_DERIVATIVES**.

The following internal derivative buffers are stored within the Edge Finder result buffer:

- **M_CROSS_DERIVATIVE_ID**. This buffer contains the cross derivatives calculated from the source buffer.
- **M_FIRST_DERIVATIVE_X_ID**. This buffer contains the first derivatives in X-direction calculated from the source buffer.
- **M_FIRST_DERIVATIVE_Y_ID**. This buffer contains the first derivatives in the Y-direction calculated from the source buffer.
- **M_SECOND_DERIVATIVE_X_ID**. This buffer contains the second derivatives in the X-direction calculated from the source buffer.
- **M_SECOND_DERIVATIVE_Y_ID**. This buffer contains the second derivatives in the Y-direction calculated from the source buffer.

Note that you can retrieve the first derivatives in the X- and Y-direction in one call using **MedgeGetResult()** with **M_FIRST_DERIVATIVES_ID**. You can also retrieve the second derivatives in the X- and Y-direction in one call using **MedgeGetResult()** with **M_SECOND_DERIVATIVES_ID**.

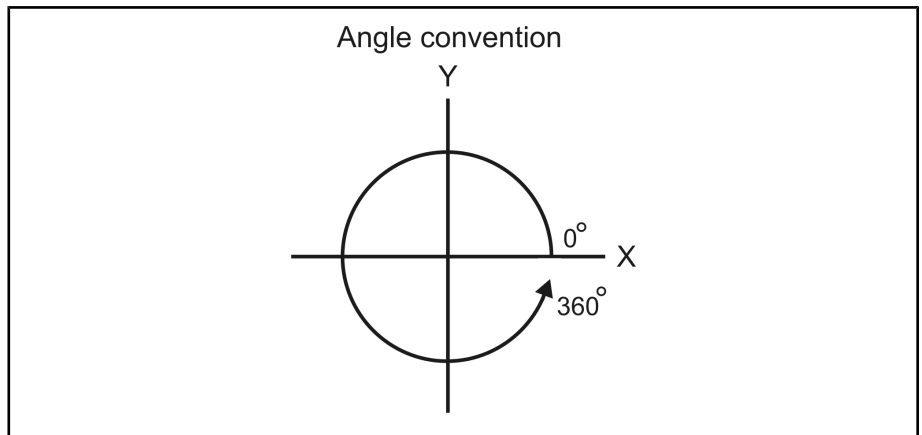
Typically, you will let Edge Finder manage its own derivative images to perform the edge extraction. However, you can specify your own derivative image buffers, so that the edge extraction is done with them instead. For more information on derivative images, and how they are used to extract edges, see the *Providing the image's derivatives* subsection in the *Advanced edge extraction* section in *Chapter 9: Edge Finder*.

Angles

You can save the internal angle buffer used when extracting edges, in the Edge Finder result buffer, using **MedgeControl()** with **M_SAVE_ANGLE**. To retrieve information about the internal angle buffer, use **MedgeGetResult()** with **M_ANGLE_ID**. You can also save the angle value of the edge at each edgel position, in the Edge Finder result buffer, using **MedgeControl()** with **M_SAVE_CHAIN_ANGLE**. Angle values refer to the angle between the horizontal axis and the edge's perpendicular direction at each edgel location.

If you save the chain angle, the angle values of the extracted edges are saved. If you save the angle buffer, the angle values of all the edges in the source image are saved; in this case, the only relevant angle values are those whose edgels have a magnitude that is above the lower-bound threshold value.

Angles are returned counter-clockwise and mapped in the range of 0 to 255. That is, 0° corresponds to 0, and 360° corresponds to 256.



Note that for edge contours, the ascending gradient angle is returned. For line crests, the strongest gradient angle between 0 and 180 (either ascending or descending) is returned.

When extracting edge contours from a color image, the angle of the polarity cannot be obtained. For example, it is impossible to determine if red is darker or lighter than green; therefore, it is impossible to ascertain the direction of the transition.

Magnitude

You can save the internal magnitude buffer used when extracting edges, in the Edge Finder result buffer, using **MedgeControl()** with **M_SAVE_MAGNITUDE**. To retrieve information about the internal magnitude buffer, use **MedgeGetResult()** with **M_MAGNITUDE_ID**. You can also save the magnitude value of the edge at each edgel position, in the Edge Finder result buffer, using **MedgeControl()** with **M_SAVE_CHAIN_MAGNITUDE**.

For object contours, the magnitude is the norm of the gradient vector at the edgel position. For line crests, the magnitude is equal to the maximum eigenvalue of the Hessian matrix at the edgel position.

If you save the chain magnitude, the magnitude values of the extracted edges are saved. If you save the magnitude buffer, the magnitude values of all the edges in the image are saved.

For more information on magnitude, and how it is used to extract edges, see the *Customizing the edge extraction settings* section earlier in this chapter.

Source image and mask

You can save both the source image, and the image used to mask the source image, in the Edge Finder result buffer. To do so, use **MedgeControl()** with **M_SAVE_IMAGE** and **M_SAVE_MASK**, respectively. To retrieve information about the source image buffer, use **MedgeGetResult()** with **M_IMAGE_ID**. To retrieve information about the internal mask buffer, use **MedgeGetResult()** with **M_MASK_ID**. For more information on masking, see the *Masking the edges* subsection in the *Advanced edge extraction* section in *Chapter 9: Edge Finder*.

Post-calculation

To avoid repeatedly calculating time-consuming edge features for all edges in a source image, you can perform post-calculations on included edges of an Edge Finder result buffer to decrease processing time. The operation of selecting edges and adding new features can be repeated until the required result is calculated.

Post-calculations are done with the result buffer only; that is, you should not provide a source image to **MedgeCalculate()**. If you do provide a source image, all current results will be discarded and replaced by the newly calculated edges. In addition, all selected features will be recalculated for all edges in the new source image.

Restrictions

Typically, anything that can be calculated can also be post-calculated. However, some post-calculations are not valid unless certain values have been initially saved.

The internal magnitude buffer (**MedgeControl()** with **M_SAVE_MAGNITUDE**) must have been initially saved if you want to post-calculate the following:

- **MedgeControl()** with **M_AVERAGE_STRENGTH**.
- **MedgeControl()** with **M_STRENGTH**.
- **MedgeControl()** with **M_SAVE_CHAIN_MAGNITUDE**.

Note that, if any of the values above were initially calculated, and you intend to fill edge gaps during post-calculation, the internal magnitude buffer (**M_SAVE_MAGNITUDE**) must have been initially saved.

The internal angle buffer (**MedgeControl()** with **M_SAVE_ANGLE**) must have been initially saved if you want to post-calculate the following:

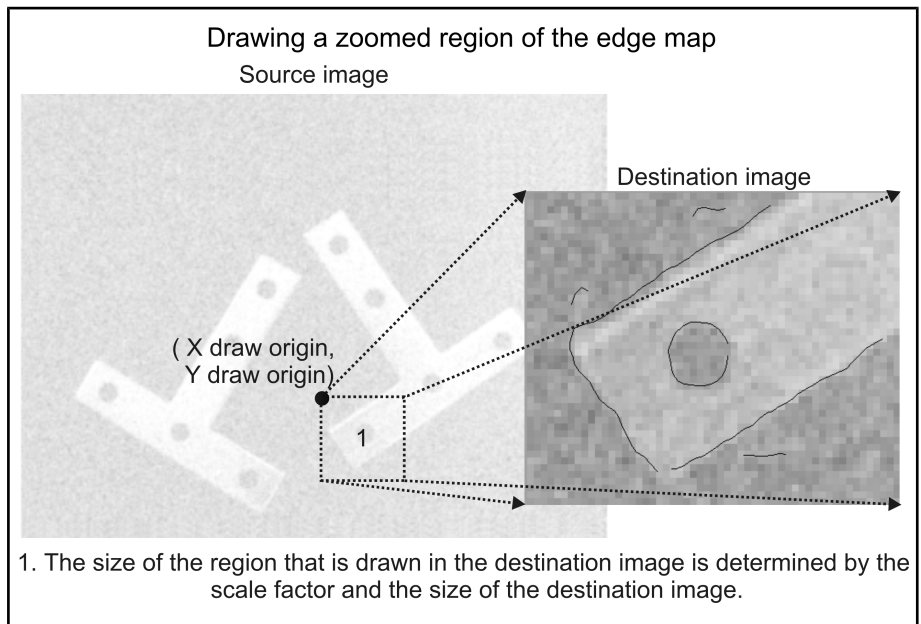
- **MedgeControl()** with **M_SAVE_CHAIN_ANGLE**.
- The **M_FILL_GAP_POLARITY** settings **M_SAME** or **M_REVERSE**.

Note that, if any of the values above were initially calculated, and you intend to fill edge gaps during post-calculation, the internal angle buffer (**M_SAVE_ANGLE**) must have been initially saved.

Thresholding can also be applied during post-calculation. In this case, all internal processing buffers (**M_SAVE_...**) must have been initially saved, in **MedgeControl()**. For more information, see the *Thresholding* subsection in the *Customizing the edge extraction settings* section in *Chapter 9: Edge Finder*.

Annotating the results

The **MedgeDraw()** function offers many drawing operations for annotating your results. You can, for example, draw a zoomed region of the edge map that was used to calculate results by specifying the appropriate values for **M_DRAW_RELATIVE_ORIGIN_X**, **M_DRAW_RELATIVE_ORIGIN_Y**, **M_DRAW_SCALE_X**, and **M_DRAW_SCALE_Y**, in **MedgeControl()**. The relative origin values must be specified in pixels, and are relative to the coordinates of the top-left corner of the region in the source image, while the scale values specify the X- and Y-scaling factors used to fill the destination buffer. For example:



You can also perform many other drawing operations with **MedgeDraw()**, such as draw a bounding box around each edge, a cross at each edge's center of gravity, or draw a cross at each edgel. In addition, you can draw the operation's numerical value by adding **M_DRAW_VALUE** to the following drawing operations: **M_DRAW_CENTER_OF_GRAVITY**, **M_DRAW_POSITION**, **M_DRAW_FERET_MIN**, **M_DRAW_FERET_MAX**, and **M_DRAW_GENERAL_FERET**. For example, if you add **M_DRAW_VALUE** to **M_DRAW_CENTER_OF_GRAVITY**, the center of gravity's coordinates are drawn within parenthesis, and centered above the drawing cross.

Typically, all drawing operations in **MedgeDraw()** can be combined; you can, therefore, draw multiple effects simultaneously. For example, to draw the edge's chains, center of gravity, bounding box, and the source image's internal cross-derivative buffer, you would specify **M_DRAW_EDGE + M_DRAW_CENTER_OF_GRAVITY + M_DRAW_BOX + M_DRAW_CROSS_DERIVATIVE**.

- ❖ Note that only one internal drawing buffer can be drawn at a time; for example, you cannot combine **M_DRAW_ANGLE** and **M_DRAW_MAGNITUDE** in the same operation.

You can either use a previously allocated graphics context to control the drawing color, or you can use the default (**M_DEFAULT**) graphics context. You can either draw directly into the selected image buffer, or non-destructively annotate in a display's overlay buffer. For more information on MIL graphics, see the *Overview* section in *Chapter 20: Displaying an image*.

Note that if the edges are calculated using a calibrated source image, Edge Finder takes the calibration into account; that is, drawings might be distorted, according to the calibration. For example, a straight line (in the world) might be drawn as a curve.

For a complete list and explanation of each drawing operation, see **MedgeDraw()** in the MIL Reference.

Advanced edge extraction

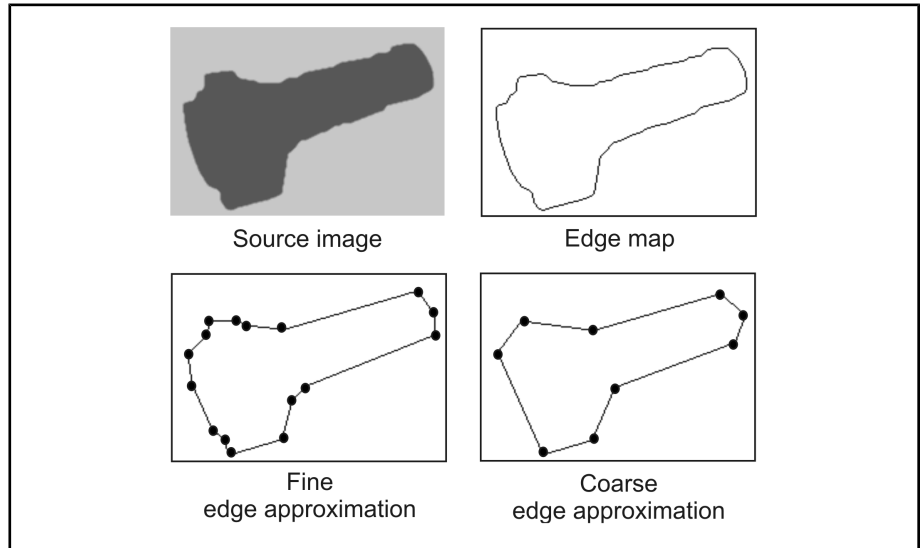
In addition to the fundamental edge extraction settings, Edge Finder provides advanced settings that allow you to write customized applications. Edges can be approximated, masked, cropped, and extracted using advanced thresholding settings. You can provide your own derivative images, and you can put data from user-supplied arrays into an Edge Finder result buffer. You can also get edgels, from an Edge Finder result buffer, which are the closest neighbors to a list of user-specified edgel coordinates.

Approximating the edges

If you are interested in a basic, geometric representation of the edge map, you can approximate your edges by specifying a polygonal segmentation. To do so, set **MedgeControl()** with **M_CHAIN_APPROXIMATION** to **M_LINE**. The default setting is **M_DISABLE**.

When set to **M_LINE**, each edge is piecewise approximated using a connected series of line segments, commonly called polylines. The polylines are defined by the set of automatically calculated vertex coordinates, which determine the location of the line segment ending points. Use **MedgeGetResult()** to retrieve the X- (**M_VERTICES_X**) and Y- (**M_VERTICES_Y**) coordinates of the vertices. This information can be retrieved for either a group of edges or a particular edge. You can also get the index of each vertices' corresponding edge or edgel, using **MedgeGetResult()** with **M_VERTICES_CHAIN_INDEX** and **M_VERTICES_INDEX**, respectively.

The resolution of the edge approximation can be adjusted using **MedgeControl()** with **M_APPROXIMATION_TOLERANCE**. The range of this control varies from 0 to 100. When set to 0, a very fine approximation of the edges is performed. When set to 100, a coarse approximation of the edges is performed. The default value is 50.



When **M_CHAIN_APPROXIMATION** is set to **M_LINE**, both the edge map and the edge approximation is calculated. When **M_CHAIN_APPROXIMATION** is disabled, only the edge map is calculated. For an **M_LINE** edge approximation, each vertex has a null bulge value.

Masking the edges

The **MedgeMask()** function allows you to apply a mask to the source image in the Edge Finder context. A mask is a binary image used to define irrelevant, inconsistent, or featureless areas in the source image. As a result, when a mask is set, subsequent calls to **MedgeCalculate()** will only extract edges in the source image's unmasked regions.

To set a mask, you must specify an image buffer (also known as a **mask buffer**) that identifies the masked pixels, using **MedgeMask()**. A masked pixel in the source image corresponds to a non-zero value in the mask buffer. Mask buffers are useful for creating non-rectangular regions of interest (ROIs). For rectangular ROIs, it is more effective to set a mask with child buffers. For more information, see the *Manipulating and controlling certain data buffer areas* section in *Chapter 18: Specifying and managing your data buffers*.

If necessary, you can remove the mask by setting the mask buffer identifier to **M_NULL**.

You can also mask edges during post-calculation. To do so, you must apply the mask after the initial call to **MedgeCalculate()**. In this case, masked edges are excluded from the result buffer and subsequent calculations. Note that partially masked edges are cropped.

Cropping the edges

Selecting an edge or a group of edges for calculation and result retrieval is typically sufficient for the majority of cases. However, in some advanced applications, you might only be interested in one part of a single edge. In this case, you can use **MedgeSelect()** with **M_CROP_CHAIN**.

Cropping an edge essentially means splitting one edge into two, thereby allowing you to choose only one of the edges to be included, excluded, or deleted from the Edge Finder result buffer. The selected portion receives a new label value, while the other portion keeps the original label value.

You can make as many cropping operations as necessary to arrive at the required set of results. Note that all calculated edge features are updated after each cropping operation.

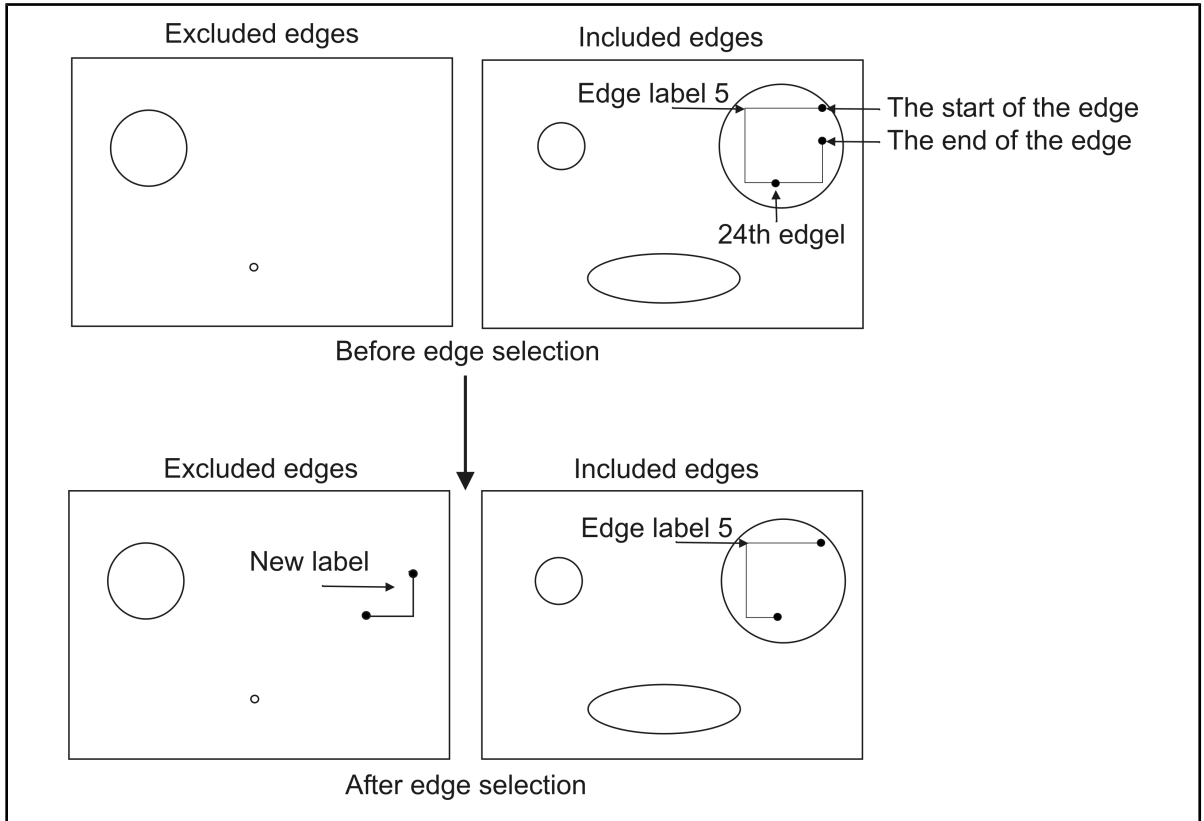
To split an edge into two edges, you must specify the following:

- The edge to split. This is identified by the edge's label value.
- The point at which to split the edge. This is identified by the edgel's index value.
- The portion of the edge to select. This is identified by all edgels in the specified edge with index values that are greater than, less than, greater than or equal to, or less than or equal to, the point at which the edge was split.

The following code shows you how to crop and exclude a portion of the edge labeled 5:

```
MedgeSelect(MilResult, M_EXCLUDE, M_CROP_CHAIN, M_GREATER, 5, 24);
```


In this example, the edge labeled 5 is split into two edges at edgel 25. One edge consists of all edgels ranging from the start of edge 5 to the 24th edgel; this edge keeps the label 5 and is not selected for exclusion. The other edge consists of all edgels ranging from the 25th edgel to the end of edge 5; this edge gets a new label value and is selected for exclusion.



Advanced thresholding

Edge Finder uses an hysteresis thresholding method to perform the edge extraction. That is, edge chains are extracted such that the magnitude values of all connected edgels are stronger than the lower bound threshold value, and at least one edgel in each edge chain has a magnitude that is stronger than the upper bound threshold value. For more information on thresholding, and on setting these bounds, see the *Thresholding* subsection in the *Customizing the edge extraction settings* section in *Chapter 9: Edge Finder*.

Edge Finder also allows you to customize the behavior of the hysteresis thresholding using the **MedgeControl()** with **M_THRESHOLD_TYPE**. That is, **M_THRESHOLD_TYPE** allows you to set how the upper and lower bound threshold values will be used (as opposed to setting the bounds themselves).

If you set **M_THRESHOLD_TYPE** to **M_HYSTERESIS**, both the lower bound threshold value and the upper bound threshold value will be used. This is the default setting.

If you set **M_THRESHOLD_TYPE** to **M_NO_HYSTERESIS**, the lower bound threshold value is ignored, and is considered to be equal to the upper bound threshold value.

If you set **M_THRESHOLD_TYPE** to **M_FULL_HYSTERESIS**, only the upper bound threshold value is used; the lower bound threshold value is ignored, and is considered to be 0. In this case, the extracted edges are sets of connected edgels such that the magnitude of at least one edgel in each edge is stronger than the upper bound threshold value. This type of thresholding can be used to extract edges in noise-free highly multi-contrast images.

Providing the image's derivatives

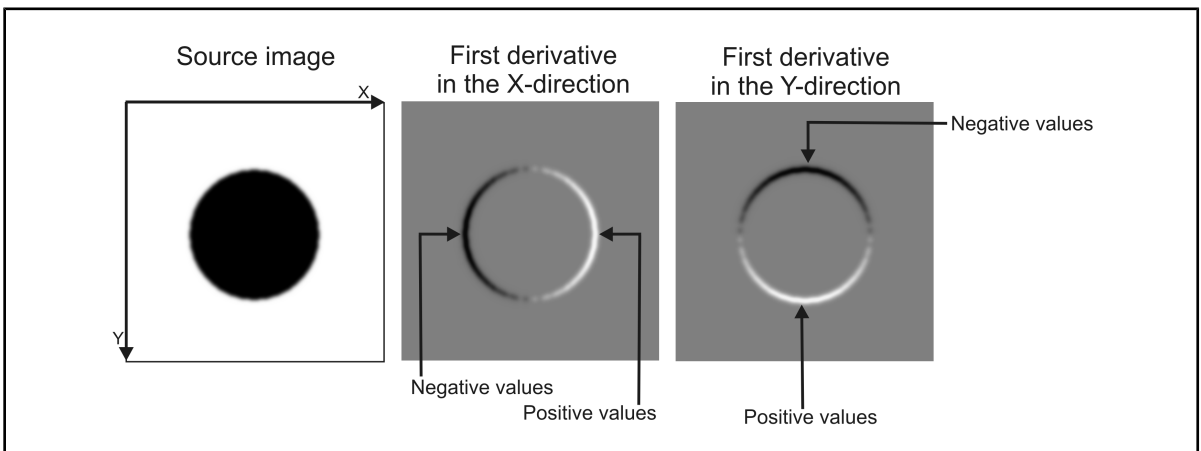
Typically, you will let Edge Finder manage its own derivative images to perform the edge extraction. However, you can specify your own derivative image buffers, so that the edge extraction is done with them instead. In this case, when you call **MedgeCalculate()**, set the source image to **M_NULL**, and specify the required derivative image buffers instead. For example:

```
MedgeCalculate(MilEdgeContext, M_NULL, DerivX, DerivY, M_NULL,
               MilResult, M_DEFAULT);
```

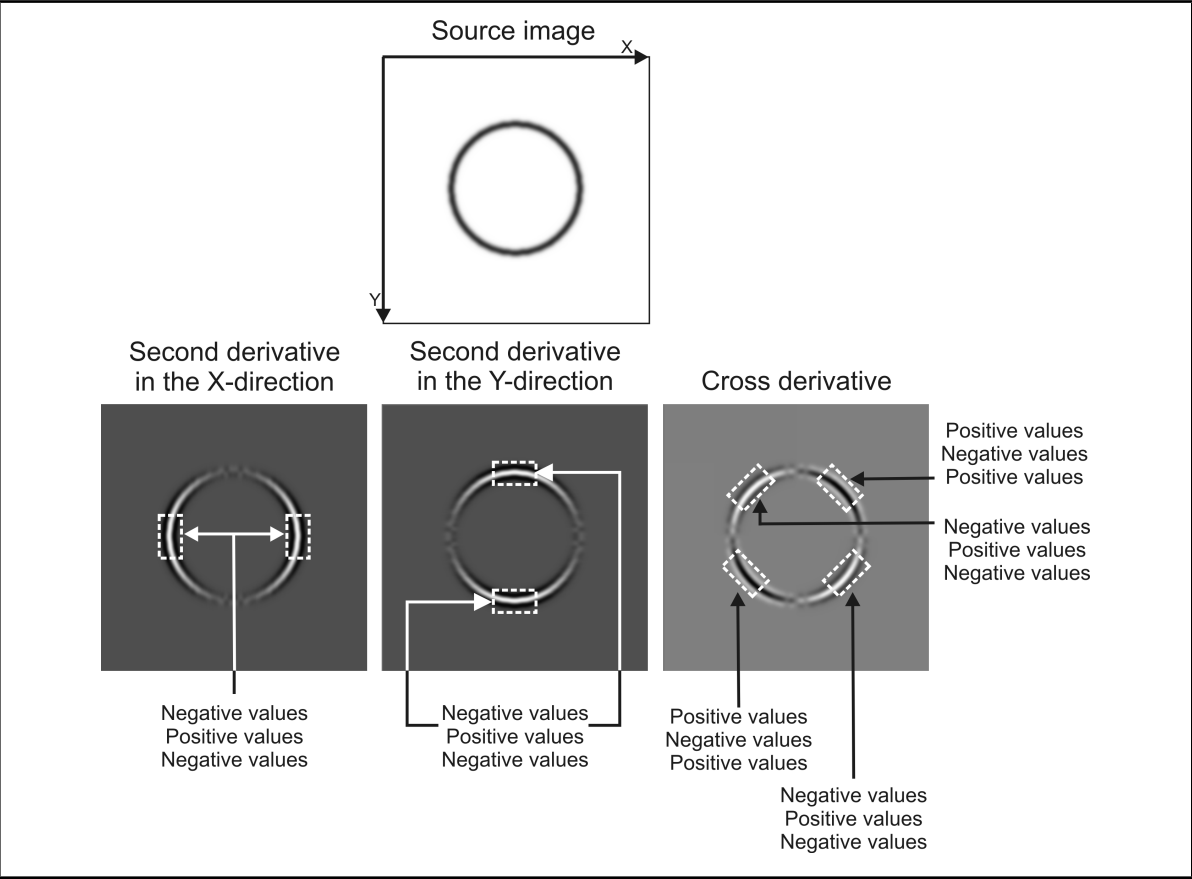
Note that, when specifying your own derivative image buffers, the majority of the processing control settings remain in effect, except for filter settings (for example, **M_FILTER_SMOOTHNESS**).

Derivative image buffers must be 1-band 16-bit signed or 1-band 32-bit floating-point buffers. In addition, all derivative image buffers must be consistent (for example, they must have the same scaling factor), and they must be of the same type and size. When the derivative image buffers are 32-bit floating-point buffers, Edge Finder uses floating-point precision calculations.

When extracting object contours, you must provide the image's first derivatives in both the X- and Y-directions. Note that the image derivatives must be consistent with the image axis; that is, if the intensity transition in the source image increases (from black to white) with the axis directions, the associated derivative values must be positive. Similarly, if the intensity transition in the source image decreases (from white to black) with the axis directions, the associated derivative values must be negative.



When extracting line crests, you must provide the image's second derivatives in both the X- and Y-direction, as well as the image's cross derivative. Note that the image derivatives must be consistent with the image axis; that is, a light line crest in the source image must result in negative second derivative values. Similarly, a dark line crest in the source image must result in positive second derivative values.



Putting data into an Edge Finder result buffer

Typically, you will use the Edge Finder result buffer to hold edge extraction information after an edge selection and calculation. However, Edge Finder also allows you to copy data from user-supplied arrays into an Edge Finder result buffer, using **MedgePut()**. This can be useful if you want to construct an Edge Finder result buffer that cannot be obtained from one image. You can therefore combine results from multiple Edge Finder result buffers to, for example, build a complete model to add to a Model Finder context. For more information, see the *Defining and adding models to your model finder context* section in *Chapter 8: Geometric Model Finder*.

Note that once edges are added to the Edge Finder result buffer, their features can be calculated with **MedgeCalculate()**.

If edgels are calculated with pixel accuracy, you must provide edgels with coordinates that are integer values; in this case, the distance between consecutive edgels should be one pixel. If edgels are calculated with subpixel accuracy, you can provide edgels with coordinates that are double values; in this case, the distance between consecutive edgels is such that, each pixel touched by the edge corresponds to one edgel. To change the accuracy edgels are calculated with, use **MedgeControl()** with **M_ACCURACY**.

To add data to an Edge Finder result buffer, you must use **MedgePut()** to specify the following user-supplied arrays:

- The X-coordinate(s) of the edgel(s).
- The Y-coordinate(s) of the edgel(s).

You must also specify the number of edgels that you want to add to the Edge Finder result buffer. If necessary, you can also provide arrays containing the index of the edge each edgel belongs to, the orientation of the edgel and the magnitude of the edgel.

The information that you provide about the edges to add to the Edge Finder result buffer must be ordered on an edge by edge basis. For example, the following table illustrates how to supply X- and Y-coordinates for three distinct edges; note that the index provided is that of the edge each edgel belongs to:

Array of X-coordinates	Array of Y-coordinates	Array of edge indices
1	2	1
2	3	1
2	4	1
3	3	1
1	9	2
2	9	2
2	8	2
4	6	3
5	6	3

Finding the closest edgels to a list of points

From an Edge Finder result buffer, you can retrieve the coordinates of edgels that correspond to the closest neighbors from a list of user-specified source point coordinates. You must use **MedgeGetNeighbors()** to specify your input data, which must include the source points and, if necessary, a corresponding angle for each point.

You can also set a series of constraints to which potential results (edgel candidates) must adhere before being returned. Only edgels that meet all constraints (if any are set) can be returned as the closest neighbors. To set the constraint(s), use **MedgeControl()**.

You can either set simple constraints, advanced constraints, or both. Simple constraints are based on the location of the edgel candidate (the search region), while advanced constraints are typically based on the inner properties of the edgel candidate (the gradient angle).

When applying constraints, you should first set the ones based on location. This will establish the search region in the Edge Finder result buffer, and generate a group of edgel candidates. If you do not apply any edgel-based constraints, then the candidates in this region that are closest to the source points will be returned. However, if you set advanced constraints, then only the edgels in this region that also adhere to these constraints can be returned.

Advanced constraints are typically based on the source edgel's gradient angle. In this case, you must provide an angle value for each source point as part of your input data for **MedgeGetNeighbors()**. Therefore, the candidate edgel will only be returned if its gradient angle adheres to the constraint applied with the source angle. Note that an edgel's gradient angle is perpendicular to the edge, and is dependent on the edge extraction process. For more information on an edgel's gradient angle, see the *How edges are calculated* subsection in the *Extracting the edges* section in *Chapter 9: Edge Finder*.

Location-based constraints

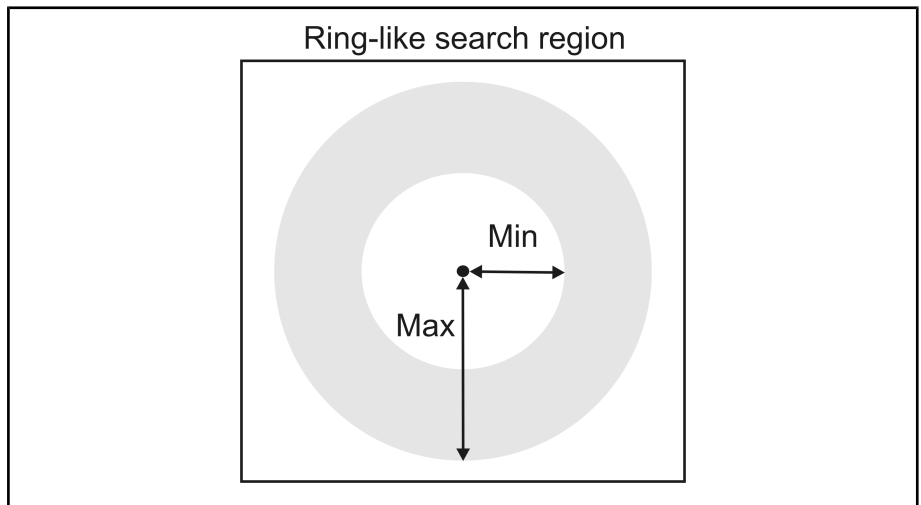
You can use location-based constraints to limit the area within the Edge Finder result buffer from which edgels matching source points can be found. All edgels that do not fall within this area will be ignored. By default, the search region encompasses all edgels within the Edge Finder result buffer.

The following **MedgeControl()** control types can be used to specify location-based constraints:

- **M_SEARCH_ANGLE.**
- **M_SEARCH_ANGLE_SIGN.**
- **M_SEARCH_ANGLE_TOLERANCE.**
- **M_SEARCH_RADIUS_MAX.**
- **M_SEARCH_RADIUS_MIN.**

The maximum (**M_SEARCH_RADIUS_MAX**) and minimum (**M_SEARCH_RADIUS_MIN**) search radius constraints set the maximum and minimum distance radii in the Edge Finder result buffer that will be searched for the closest edgels. These values must be set in pixels. You must use **MedgeGetNeighbors()** to specify the source point from which the radii are measured.

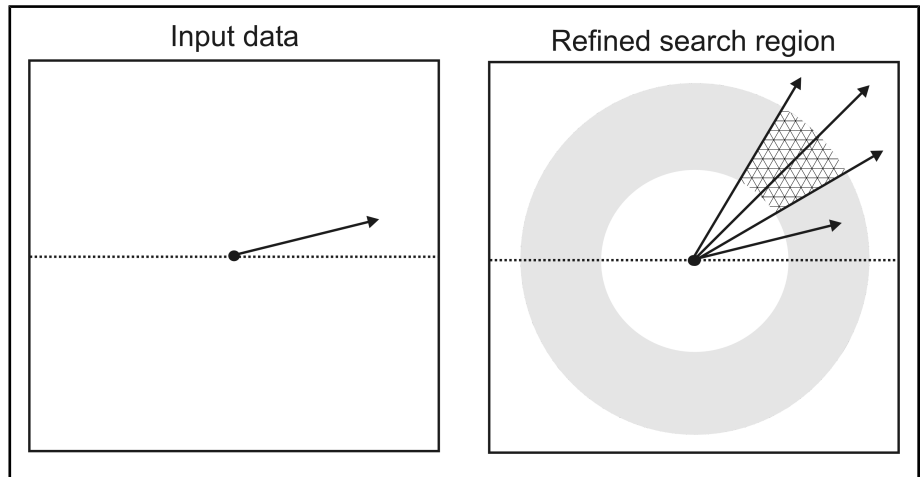
Together, **M_SEARCH_RADIUS_MAX** and **M_SEARCH_RADIUS_MIN** define a ring-like region, in the Edge Finder result buffer, that will be used to find the closest edgels. All edgels that fall outside this ring will be ignored. By default, the minimum distance radius is 0 pixels, and the maximum distance radius is an infinite number of pixels (**M_INFINITE**); that is, by default, the entire Edge Finder result buffer will be searched. Note that the **M_INFINITE** setting will retrieve all the closest edgels (one edgel per edge).



The Edge Finder result buffer region that will be searched for the closest edgels can be further constrained to an angular sector within the area defined by the radius constraints (**M_SEARCH_RADIUS_...**). You can use **M_SEARCH_ANGLE** to define the search angle, and **M_SEARCH_ANGLE_TOLERANCE** to define the tolerance. The search angle is relative to the source angle set with **MedgeGetNeighbors()**, and the tolerance is applied to the resulting angle evenly; that is, half of the tolerance angle is applied in the positive direction, and half is applied in the negative direction. All edgels that are located within this angular

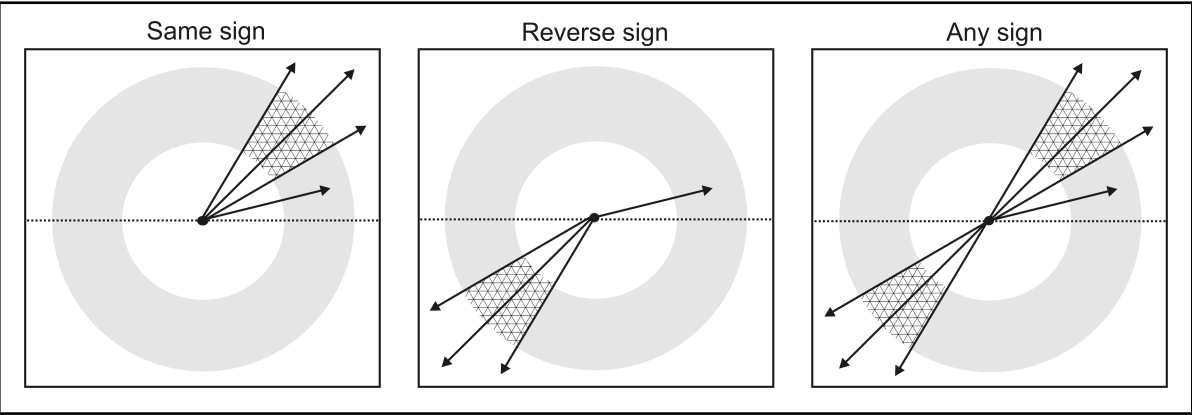
sector (tolerance) can be returned. Any value between 0.0° to 360.0° can be set for both the search angle and tolerance. By default, **M_SEARCH_ANGLE** is set to 0.0° and **M_SEARCH_ANGLE_TOLERANCE** is set to 360.0 degrees; therefore, all edgels will be considered.

For example, if you set **M_SEARCH_ANGLE** to 30.0° , **M_SEARCH_ANGLE_TOLERANCE** to 30.0° , and the source angle to 15° , only edgels that fall between 30.0° and 60.0° will be considered.

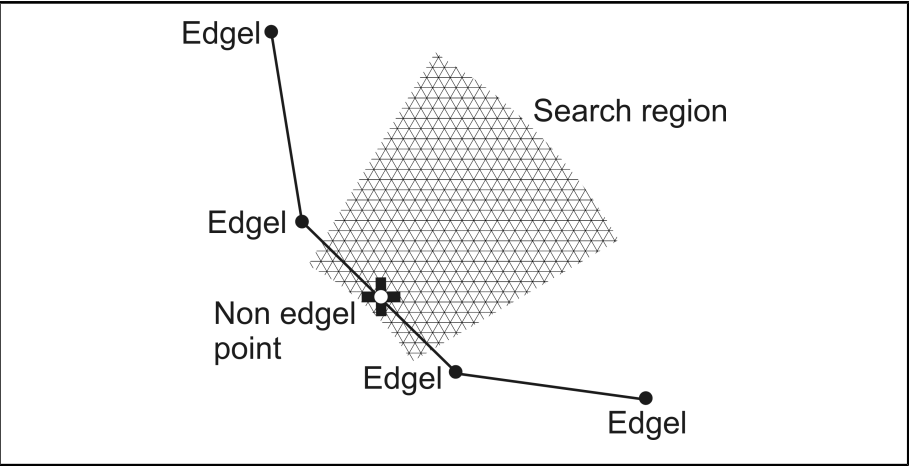


You can also set the sign of the angle that will be used to find the closest edgels with **M_SEARCH_ANGLE_SIGN**. The sign of the angle can be the same as (**M_SAME**) or the reverse of (**M_REVERSE**), the source angle set with **MedgeGetNeighbors()**. The default is **M_SAME**. The sign can also be set to **M_SAME_OR_REVERSE**, which means that it can be the same as, or the reverse of, the source angle.

For example, if you set the source angle to 15°, and the search angle and tolerance to 30.0°, then edgels that fall within an angle that has the same sign would be located between 30.0° and 60.0°, edgels that fall within an angle that has a reverse sign would be located between 210° and 240°, and edgels that fall within an angle that has any sign would be located at either between 30.0° and 60.0°, or between 210° and 240°.



Note that, if necessary, you can allow a point between two edgels to be returned, using `M_GET_SUBEDGELS` in `MedgeGetNeighbors()`. In this case, an edgel candidate is returned, as long as an edge (and not necessarily an edgel) is located within the search region.



Edgel-based constraints

As mentioned, location-based constraints limit the region in the Edge Finder result buffer that will be searched for the closest edgels. Edgel-based constraints, on the other hand, are applied to the edgels themselves. Only edgels that adhere to these constraints can be returned. Note that, if you have applied location-based constraints, only edgels within the defined search region will be considered.

The following **MedgeControl()** control types are used to specify edgel-based constraints:

- **M_NEIGHBOR_MAXIMUM_NUMBER.**
- **M_NEIGHBOR_MINIMUM_SPACING.**
- **M_NEIGHBOR_ANGLE.**
- **M_NEIGHBOR_ANGLE_TOLERANCE.**

The maximum number constraint (**M_NEIGHBOR_MAXIMUM_NUMBER**) sets the maximum number of closest edgels (that is, the N closest edgels) that can be returned, for each source point. The default maximum number is 1; therefore, the edgel closest to each source point is returned.

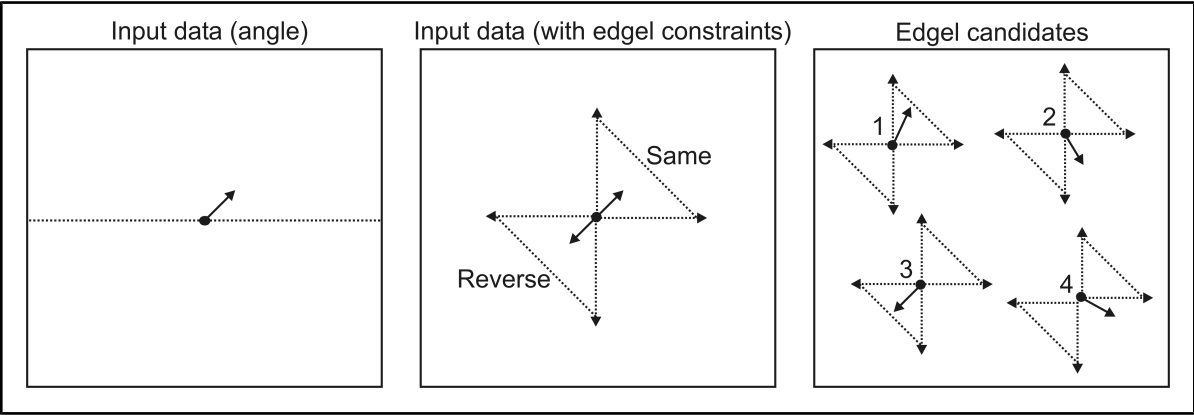
The minimum spacing constraint (**M_NEIGHBOR_MINIMUM_SPACING**) sets the minimum distance separating closest edgel candidates within the same edge. Therefore, edgels that are too close together will not be returned. The spacing is set in number of edgels and must be greater than or equal to 1. The default minimum spacing is infinite (**M_INFINITE**), which means that there is no minimum distance (number of edgels) that must separate edgel candidates.

The neighbor's angle constraint (**M_NEIGHBOR_ANGLE**) sets the gradient angle that an edgel must have, before being considered a candidate.

M_NEIGHBOR_ANGLE is based on a source angle (for each source point), set with **MedgeGetNeighbors()**. The gradient angle of the edgel candidate can be the same (**M_SAME**), the reverse (**M_REVERSE**), or the same or reverse (**M_SAME_OR_REVERSE**) as the angle of the source point. The constraint can also be set to any angle (**M_ANY**); in this case, the source angle and the source point are ignored. The default is **M_ANY**. Note that **M_NEIGHBOR_ANGLE** can only be used if the **M_SAVE_ANGLE** internal buffer was initially saved.

You can also set an angular tolerance for **M_NEIGHBOR_ANGLE** using **M_NEIGHBOR_ANGLE_TOLERANCE**. In this case, candidate edgels that have a gradient angle that falls within the specified angular range (tolerance) can be returned. The tolerance is applied to the **M_NEIGHBOR_ANGLE** value evenly; that is, half of the tolerance angle is applied in the positive direction, and half is applied in the negative direction. Any angle between 0.0° to 360.0° can be set as a tolerance. By default, **M_NEIGHBOR_ANGLE_TOLERANCE** is set to 180.0°; therefore, only candidate edgels that have a gradient angle that is the same as the source angle will be considered. Note that setting **M_NEIGHBOR_ANGLE_TOLERANCE** to 360.0° is equivalent to setting the neighbor angle constraint to **M_ANY**; in either case, edgels with any gradient angle are considered.

For example, if you set the angle of the source point to 45°, **M_NEIGHBOR_ANGLE** to **M_SAME_OR_REVERSE**, and **M_NEIGHBOR_ANGLE_TOLERANCE** to 90°, only edgels that have an angle between 0° and 90° or between 180° and 270° can be returned. In the following example, edgels 1 and 3 are considered the closest edgel candidates.



Optimizing edge extractions

There are some things that you can do to optimize the edge extraction and/or speed up the calculation process:

- Specify a region of interest (ROI).
- Perform post-calculation to calculate expensive features.
- Optimize your control type (**MedgeControl()**) settings.
- Retrieve calibrated results in world units.

Specify a region of interest (ROI)

Specify a region of interest (ROI) that contains all necessary edges. You can specify a region of interest using a child buffer. All edgels that fall outside the region will be ignored, speeding up the edge extraction. For non-rectangular ROIs, you can specify the appropriate region using a mask (**MedgeMask()**).

For more information on setting a region of interest (ROI), see the *Manipulating and controlling certain data buffer areas* section in *Chapter 18: Specifying and managing your data buffers*, or the *Masking the edges* subsection in the *Advanced edge extraction* section in *Chapter 9: Edge Finder*.

Perform post-calculation

To compute features as efficiently as possible, you should calculate a few features first (preferably, the fastest), and eliminate as many unnecessary edges as possible. Then, post-calculate expensive features on the remaining edges. The process of calculating features, and eliminating unnecessary edges can be repeated until the required result is calculated. Post-calculation greatly decreases processing time since MIL won't have to calculate the most time-consuming edge features for all edges.

For more information on post-calculation, see the *Post-calculation* subsection in the *Calculating and retrieving results* section in *Chapter 9: Edge Finder*.

Optimize your control type settings

To compute features as efficiently as possible, you should optimize your **MedgeControl()** control type settings. Some settings result in faster calculations (**MedgeCalculate()**). Note that faster calculations do not mean better results and your application might require settings that take longer to calculate.

Adjust the accuracy

Use the minimum required accuracy (**M_ACCURACY**); decreasing the accuracy increases the speed of the calculation process. Disabling **M_ACCURACY** results in pixel precision, which although not very precise, is typically sufficiently accurate when extracting the edges of large objects. By default, the accuracy is set to **M_HIGH**; this results in subpixel edgel accuracy, but typically a slower calculation process.

For more information on setting the accuracy, see the *Edgel accuracy* subsection in the *Customizing the edge extraction settings* section in *Chapter 9: Edge Finder*.

Building edge chains

If possible, reduce the detail level of edge chains by disabling **M_CHAIN_ALL_NEIGHBORS**. Disabling this control type increases the speed of the calculation but reduces the level of detail present in the edge chains. Enabling this control type results in slower calculations; however, the edge chains contain the most amount of edgel information possible. Note that by default, this control type is disabled.

For more information on edge chains, see the *The basics* subsection in the *Extracting the edges* section in *Chapter 9: Edge Finder*.

Adjust the extraction scale

If possible, reduce the scale of the image when extracting edges, by setting **M_EXTRACTION_SCALE** to a value between 0.0 and 1.0. Reducing the extraction scale can increase the speed at which edges are found and extracted. Note however, this can result in less reliable results, including the loss of important details and/or reduced accuracy of the extraction results. An extraction scale greater than 1.0 will slow down the extraction. Once the edge extraction is complete, the results are scaled to the original scale of the image. Note that the default setting of 1.0 usually provides the most accurate search results.

Specifying the IIR filter mode

When using an IIR filter, it is generally faster to implement the filtering operation recursively, unless you implement it using kernel mode with a very small convolution kernel (for example, 3x3). If you use kernel mode with a large kernel, the calculation will be slower, although more accurate. Note that if you have dedicated hardware performing the operation, it is typically faster to use a convolution kernel.

For more information on filters, see the *Customizing the edge extraction settings* section earlier in this chapter.

Specify the smoothness

If your image contains a lot of noise, increase the smoothness factor (**M_FILTER_SMOOTHNESS**). Increasing the smoothness factor reduces the noise, resulting in the edge extraction operation extracting less unwanted edges. This will typically decrease the time it takes to extract the edges. Be aware that in kernel mode, increasing the smoothness factor can increase the size of the convolution kernel, which increases the processing time. Note that a very high smoothing level can result in a loss of important detail and a decrease in precision.

For more information on specifying the smoothness factor, see the *Smoothing* subsection in the *Customizing the edge extraction settings* section in *Chapter 9: Edge Finder*.

Specify the magnitude

Use the square of the gradient magnitude, by setting **M_MAGNITUDE_TYPE** to **M_SQR_NORM**. This optimizes the edge extraction while preserving a very good edgel result. **M_SQR_NORM** is calculated faster than **M_NORM**, but is less accurate. Note that for **M_MAGNITUDE_TYPE**, the default is **M_SQR_NORM** for contour Edge Finder contexts, and the default is **M_NORM** for crest Edge Finder contexts.

For more information on the magnitude type, see the *Magnitude type* subsection in the *Customizing the edge extraction settings* section in *Chapter 9: Edge Finder*.

Setting the threshold

Use the highest possible threshold (**M_THRESHOLD_MODE**) that still detects all pertinent edgels. The lower the threshold, the more sensitive the edgel detection; that is, more edgels are detected. The default threshold mode is **M_HIGH**.

For more information on thresholding, see the *Thresholding* subsection in the *Customizing the edge extraction settings* section in *Chapter 9: Edge Finder*.

Calculating the length of each edge

Calculate the length of each edge using **M_FAST_LENGTH**. **M_FAST_LENGTH** gives a coarser but faster approximation of the edge's length than **M_LENGTH**.

For more information on calculating the length, see **M_FAST_LENGTH** and **M_LENGTH** in **MedgeControl()**.

Limiting the search radius

When retrieving the coordinates of edgels using **MedgeGetNeighbors()**, limit the search region using **M_SEARCH_RADIUS_MAX**, **M_SEARCH_RADIUS_MIN**, and **M_SEARCH_ANGLE_TOLERANCE**. All edgels that fall outside this region will be ignored, speeding up the edge extraction.

For more information on setting the search radius, see the *Location-based constraints* subsection in the *Advanced edge extraction* section in *Chapter 9: Edge Finder*.

Retrieving calibrated results

If your image is calibrated, retrieve results in world units. Retrieving results in pixel units will typically be slower. Also note that, in the presence of distortion, some results are meaningless when converted from real-world to pixel units (for example, the Feret angles).

For more information, see the *Calibration overview* section in *Chapter 5: Camera calibration*.

Interfacing with Geometric Model Finder

The MIL Model Finder module allows you to define models from an Edge Finder result buffer or find model occurrences in an Edge Finder result buffer. If you want to use Model Finder, and are also familiar with Edge Finder, you can perform specialized edge selections, which can reduce the unnecessary geometric complexity in both the model and/or the target image. For example, if you want to find model occurrences in an Edge Finder result buffer, you can use **MedgeSelect()** to select which edges to include in the buffer, based on one of several criteria. As a result, Model Finder will have much less information to analyze, thereby potentially speeding up the search for the model(s).

Another way to speed up the search for model(s) in the result of an edge extraction is to specify a lower scale for the edge extraction, using **M_EXTRACTION_SCALE** in **MedgeControl()**. **M_EXTRACTION_SCALE** must be set before calling **MedgeCalculate()**. Be aware that reducing the extraction scale can result in less reliable results, including, the loss of important details and/or can reduce the accuracy of the search result. Note that **M_EXTRACTION_SCALE** sets or returns the scale of the image at which to do the edge extraction. Once the extraction is complete, the results are scaled to the original scale of the image.

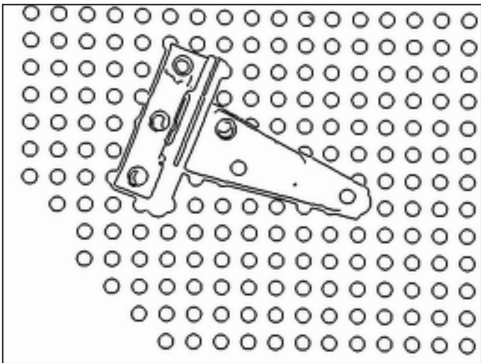
To have further control over the edges that Model Finder uses, you can allocate an Edge Finder result buffer and add the required edge chains to the result buffer, using **MedgePut()**. This allows you to combine results from multiple result buffers, or create a target that contains user-defined edge chains. Note that you do not need to call **MedgeCalculate()** after adding edge chains to a result buffer. For more information on adding edge chains, see the *Putting data into an Edge Finder result buffer* subsection in the *Advanced edge extraction* section in *Chapter 9: Edge Finder*.

- ❖ Models and target images should use the same edge types (object contours or line crests). Therefore, if the target edges are object contours, the model edges must be object contours; if the target edges are line crests, the model edges must be line crests.

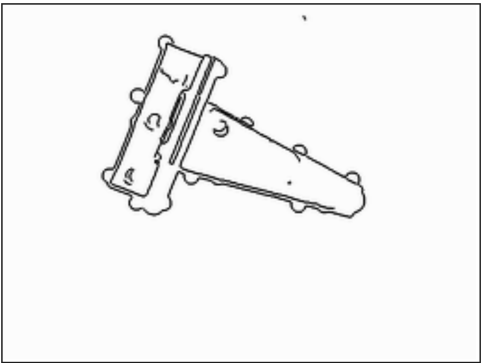
If you expect to use an Edge Finder result buffer with Model Finder, you must enable **M_MODEL_FINDER_COMPATIBLE** in **MedgeControl()** before the initial call to **MedgeCalculate()**. To add models defined from an Edge Finder result buffer to a Model Finder context, use **M_EDGE_RESULT** in **MmodDefine()**.

To set the sensitivity with which edgels are detected, Edge Finder users can use **MedgeControl()** with either **M_THRESHOLD_MODE** or **M_DETAIL_LEVEL**. **M_DETAIL_LEVEL** helps users already familiar with Model Finder and should only be used when interfacing with Model Finder. In **MedgeControl()**, the **M_DETAIL_LEVEL** setting overrides the **M_THRESHOLD_MODE** setting.

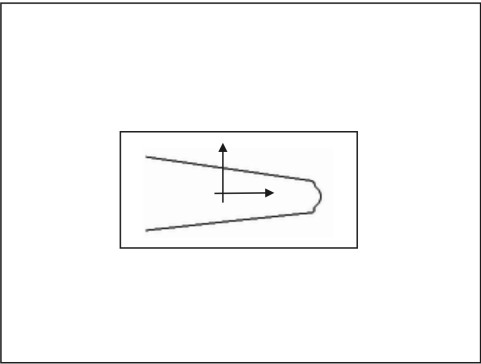
The following is an example of a typical Edge Finder/Model Finder application:



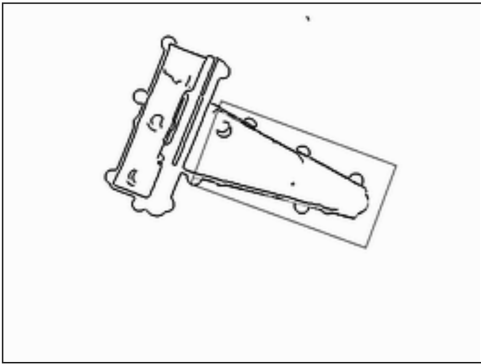
Edge map
(edge extraction results)



Edge map after selection operation
(edge extraction results)



ModelFinder model

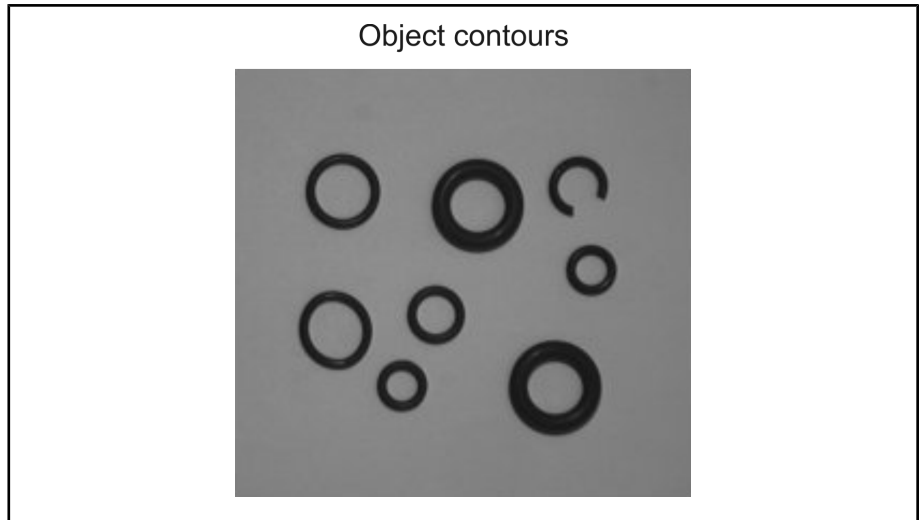


Model occurrence found

For more information on Model Finder, see the *Geometric Model Finder module* section in *Chapter 8: Geometric Model Finder*.

MIL Edge Finder example

The Edge Finder example *MEdge.cpp* illustrates how the module can be used with the following source image:



Specifically, *MEdge.cpp* shows how to define an Edge Finder context to extract and select object contours in the above source image.

```

/*****
/*
 * File name: MEdge.cpp
 *
 * Synopsis: This program uses the MIL Edge Finder module to find and measure
 *           the outer diameter of good seals in a target image.
 */
#include <mil.h>

/* Source MIL image file specifications. */
#define CONTOUR_IMAGE           M_IMAGE_PATH MIL_TEXT("Seals.mim")
#define CONTOUR_MAX_RESULTS     100
#define CONTOUR_MAXIMUM_ELONGATION 0.8
#define CONTOUR_DRAW_COLOR      M_COLOR_GREEN
#define CONTOUR_LABEL_COLOR     M_COLOR_RED

/* Main function. */
int MosMain(void)

```

```

{
MIL_ID      MilApplication,          /* Application identifier.  */
            MilSystem,               /* System Identifier.       */
            MilDisplay,              /* Display identifier.      */
            MilImage,                /* Image buffer identifier. */
            MilOverlayImage,         /* Overlay image.          */
            MilEdgeContext,          /* Edge context.           */
            MilEdgeResult;           /* Edge result identifier.  */
MIL_DOUBLE  EdgeDrawColor = CONTOUR_DRAW_COLOR, /* Edge draw color.        */
            LabelDrawColor = CONTOUR_LABEL_COLOR; /* Text draw color.        */
MIL_INT     NumEdgeFound = 0L,        /* Number of edges found.   */
            NumResults = 0L,          /* Number of results found. */
            i;                        /* Loop variable.          */
MIL_DOUBLE  MeanFeretDiameter[CONTOUR_MAX_RESULTS]; /* Edge mean Feret diameter. */

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

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

/* Prepare for overlay annotations. */
MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);
MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

/* Pause to show the original image. */
MosPrintf(MIL_TEXT("\nEDGE MODULE:\n"));
MosPrintf(MIL_TEXT("-----\n\n"));
MosPrintf(MIL_TEXT("This program determines the outer seal diameters ")
            MIL_TEXT("in the displayed image \n"));
MosPrintf(MIL_TEXT("by detecting and analyzing contours")
            MIL_TEXT("with the Edge Finder module.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Allocate a Edge Finder context. */
MedgeAlloc(MilSystem, M_CONTOUR, M_DEFAULT, &MilEdgeContext);

/* Allocate a result buffer. */
MedgeAllocResult(MilSystem, M_DEFAULT, &MilEdgeResult);

/* Enable features to compute. */
MedgeControl(MilEdgeContext, M_MOMENT_ELONGATION, M_ENABLE);
MedgeControl(MilEdgeContext, M_FERET_MEAN_DIAMETER+M_SORT1_DOWN, M_ENABLE);

/* Calculate edges and features. */
MedgeCalculate(MilEdgeContext, MilImage, M_NULL, M_NULL, M_NULL, MilEdgeResult,
            M_DEFAULT);

/* Get the number of edges found. */
MedgeGetResult(MilEdgeResult, M_DEFAULT, M_NUMBER_OF_CHAINS+M_TYPE_MIL_INT,

```

```

&NumEdgeFound, M_NULL);

/* Draw edges in the source image to show the result. */
MgraColor(M_DEFAULT, EdgeDrawColor);
MedgeDraw(M_DEFAULT, MilEdgeResult, MilOverlayImage, M_DRAW_EDGES, M_DEFAULT,
          M_DEFAULT);

/* Pause to show the edges. */
MosPrintf(MIL_TEXT("%d edges were found in the image.\n"), NumEdgeFound);
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Exclude elongated edges. */
MedgeSelect(MilEdgeResult, M_EXCLUDE, M_MOMENT_ELONGATION,
          M_LESS, CONTOUR_MAXIMUM_ELONGATION, M_NULL);

/* Exclude inner chains. */
MedgeSelect(MilEdgeResult, M_EXCLUDE, M_INCLUDED_EDGES, M_INSIDE_BOX, M_NULL, M_NULL);

/* Draw remaining edges and their index to show the result. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);
MgraColor(M_DEFAULT, EdgeDrawColor);
MedgeDraw(M_DEFAULT, MilEdgeResult, MilOverlayImage, M_DRAW_EDGES, M_DEFAULT,
          M_DEFAULT);

/* Pause to show the results. */
MosPrintf(MIL_TEXT("Elongated edges and inner edges of each seal were removed.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Get the number of edges found. */
MedgeGetResult(MilEdgeResult, M_DEFAULT, M_NUMBER_OF_CHAINS+M_TYPE_MIL_INT,
          &NumResults, M_NULL);

/* If the right number of edges were found. */
if ( (NumResults >= 1) && (NumResults <= CONTOUR_MAX_RESULTS) )
{
    /* Draw the index of each edge. */
    MgraColor(M_DEFAULT, LabelDrawColor);
    MedgeDraw(M_DEFAULT, MilEdgeResult, MilOverlayImage, M_DRAW_INDEX,
          M_DEFAULT, M_DEFAULT);

    /* Get the mean Feret diameters. */
    MedgeGetResult(MilEdgeResult, M_DEFAULT, M_FERET_MEAN_DIAMETER,
          MeanFeretDiameter, M_NULL);

    /* Print the results. */
    MosPrintf(MIL_TEXT("Mean diameter of the %ld outer edges are:\n\n"), NumResults);
    MosPrintf(MIL_TEXT("Index   Mean diameter \n"));
    for (i=0; i<NumResults; i++)
    {
        MosPrintf(MIL_TEXT("%-11d%-13.2f\n"), i, MeanFeretDiameter[i]);
    }
}

```

```
    }
else
{
    MosPrintf(MIL_TEXT("Edges have not been found or the number of ")
              MIL_TEXT("found edges is greater than\n"));
    MosPrintf(MIL_TEXT("the specified maximum number of edges !\n\n"));
}

/* Wait for a key press. */
MosPrintf(MIL_TEXT("\nPress <Enter> to end.\n"));
MosGetch();

/* Free MIL objects. */
MbufFree(MilImage);
MedgeFree(MilEdgeContext);
MedgeFree(MilEdgeResult);

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

return 0;
}
```

Chapter

10

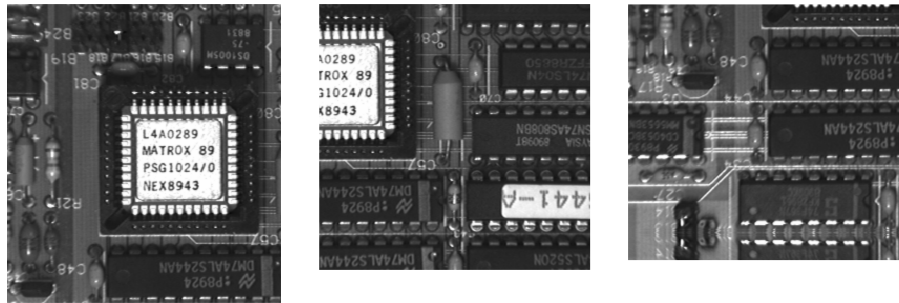
Registration

This chapter explains how to use the MIL Registration module to find the transformations that optimally position images in a common coordinate system; this process is referred to as image registration and the common coordinate system is referred to as the global coordinate system. This chapter also describes how to use the module to perform other imaging operations with the registration results.

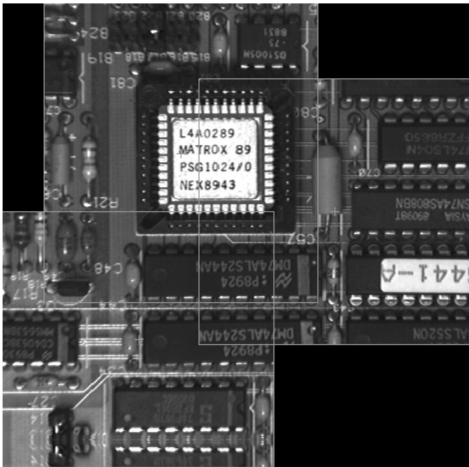
MIL Registration module

The MIL Registration module allows you find the transformations that optimally position images in a common coordinate system; this is referred to as image registration, and the common coordinate system is referred to as the global coordinate system. Registering images is necessary, for example, in industries that need to combine small images and analyze the larger one, recover the geometry of a scene, or superimpose and align two images to see the differences between them.

Three small images



Images placed in their calculated positions and outlined



The module determines exactly how images overlap and fit together, based on the pixels in the images and some preliminary information that you supply. This information includes the rough location of all the images and the type of transformation to use to optimize their alignment. Using this information with advanced algorithms, the MIL Registration module finds the best match between the overlapping regions of the images. It then calculates the transformation that needs to be applied to each image to obtain the optimal match in the overlapping regions.

The Registration module supports mosaicing, which combines images to form one larger image or to create an image with improved resolution (super-resolution). It also supports converting coordinates between two of the following coordinate systems: the global coordinate system, any image's coordinate system, and the mosaic's coordinate system.

Although the registration operation accepts color images, only band 0 is used. In contrast, color images are fully supported with the mosaicing operation. Each band must be 8-bit unsigned.

The Registration module does not take into account the calibration of images.

Steps to performing registration

The following steps provide a basic methodology for using the MIL Registration module:

1. Allocate a registration context, using **MregAlloc()**.
2. Allocate a registration result buffer to hold the results of the registration operation, using **MregAllocResult()**.
3. Verify that the default number of registration elements (256) is sufficient for your application, using **MregInquire()** with **M_NUMBER_OF_ELEMENTS**. A registration element contains the information required to register an image; therefore the context must contain at least the same number of registration elements as images that need to be registered. To change the number of registration elements, use **MregControl()** with **M_NUMBER_OF_ELEMENTS**.
4. Specify the rough location of the images, using **MregSetLocation()** with each of the images' registration elements.
5. Specify the global registration settings, using **MregControl()**.
6. Specify the registration settings for the individual registration elements, using **MregControl()**.
7. Perform the registration, using **MregCalculate()**.
8. Retrieve the required results from the result buffer, using **MregGetResult()**.
9. If necessary, save your registration context or your result buffer, using **MregSave()** or **MregStream()**.
10. If necessary, use the registration results to perform mosaicing, using **MregTransformImage()**.
11. If necessary, use the registration results to convert positions between two of the following coordinate systems: the global coordinate system, any registered image's coordinate system, or a mosaic's coordinate system. This is done using **MregTransformCoordinate()** or **MregTransformCoordinateList()**.

12. Free all your allocated objects, using **MregFree()**.

Basic concepts

The basic concepts and vocabulary conventions for the MIL Registration module are:

- **Global coordinate system.** The coordinate system in which the transformations, resulting from the registration calculation, position the images.
- **Mosaicing.** The process of combining many smaller images to form one larger image, using the results of the registration calculation.
- **Overlapping region.** The region in images that share the same coordinates in a common coordinate system.
- **Reference image.** The image with respect to whose coordinate system you set the rough location of another image. Each image must be positioned with respect to either a reference image's coordinate system or the global coordinate system.
- **Registration.** The registration of images involves finding the transformations that optimally position images in a common (global) coordinate system.
- **Registration element.** A container for the settings used to control the registration of a specific image. Registration elements are located in the registration context.
- **Registration element's image.** The image whose index in the image array is the same as the index of a particular registration element. The registration of an image is controlled by the contents of its associated registration element.
- **Registration context.** The MIL registration context is a container for the registration information and each individual registration element.
- **Registration result element.** A registration result element, located in the registration result buffer, is a container for the registration results of its associated image. It also contains settings to control the mosaicing and coordinate transformation operations.

- **Rough location.** Approximate location of an image in its reference image's coordinate system. The registration calculation finds the transformation that will convert the image's rough location into its optimal location in the global coordinate system.
- **Transformation matrix.** A matrix that maps a set of coordinates from a source coordinate system into a destination coordinate system.

The registration process

MregCalculate() takes a series of images and their rough locations in another image's coordinate system or the global coordinate system, and applies the following three step process to determine the transformations that optimally position each image in the global coordinate system.

1. Each image is first transformed into its rough location in the coordinate system of its reference image (specified using **MregSetLocation()**). This initial alignment of each image with its reference image creates a region in which both images overlap, which is necessary for the registration process to be successful.
2. For each image, the registration operation then finds the transformation that optimizes the match in the overlapping region between the image and its reference image; this is referred to as the optimization step. The algorithm used to find the optimal match depends on the context that you allocate (using **MregAlloc()**). MIL supports a correlation type context and the algorithm it uses is described in the *Correlation contexts* subsection in the *The registration process* section in *Chapter 10: Registration*. You can control the type of transformation that the algorithm can use to reposition the image from its rough location (for example, a translation or a perspective warping), using **MregControl()** with **M_TRANSFORMATION_TYPE**. For more information, see the *Selecting the transformation type* subsection in the *Customizing your registration settings* section in *Chapter 10: Registration*.
3. Once the optimal transformations that place each image in its reference image's coordinate system are found, each transformation is converted so that it maps the image into the global coordinate system instead.

Note that for the image whose reference coordinate system is the global coordinate system, no registration is performed and the optimal transformation is the same as the one set with **MregSetLocation()**.

Correlation contexts

Correlation registration contexts use normalized grayscale correlation to optimize the match in the overlapping regions. Subsections within the overlapping regions are chosen and a pixel by pixel normalized grayscale correlation is calculated on each of the subsections. The **MregCalculate()** function then finds the best possible correlation between the subsections of the overlapping regions and computes the transformations required to obtain this alignment.

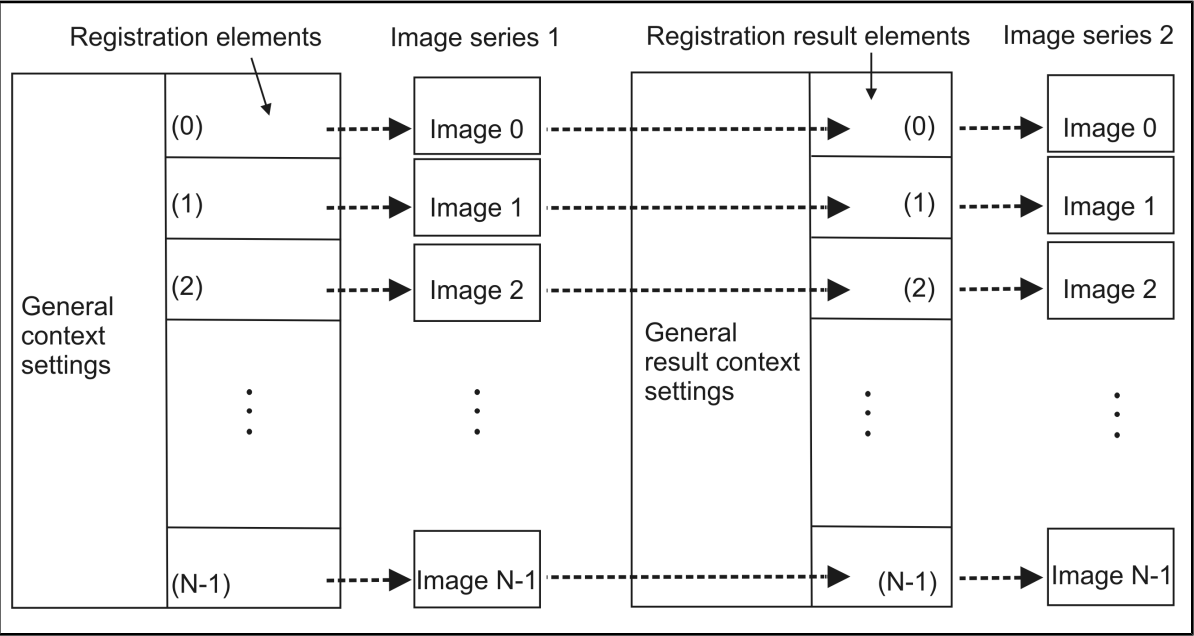
By applying the correlation algorithm on subsections within an overlapping region instead of on the whole overlapping region, the algorithm is more robust to local changes in contrast and intensity within the images. For a better understanding of the normalized grayscale correlation algorithm, see the *The pattern matching algorithm for advanced users* section in *Chapter 7: Pattern matching*.

Registration elements and images

The settings used to control the registration calculation are specified in registration elements, located in the registration context. The results of the calculation are stored in registration result elements, located in the registration result buffer. Registration result elements can also store the settings for operations that use the registration results, such as mosaicing.

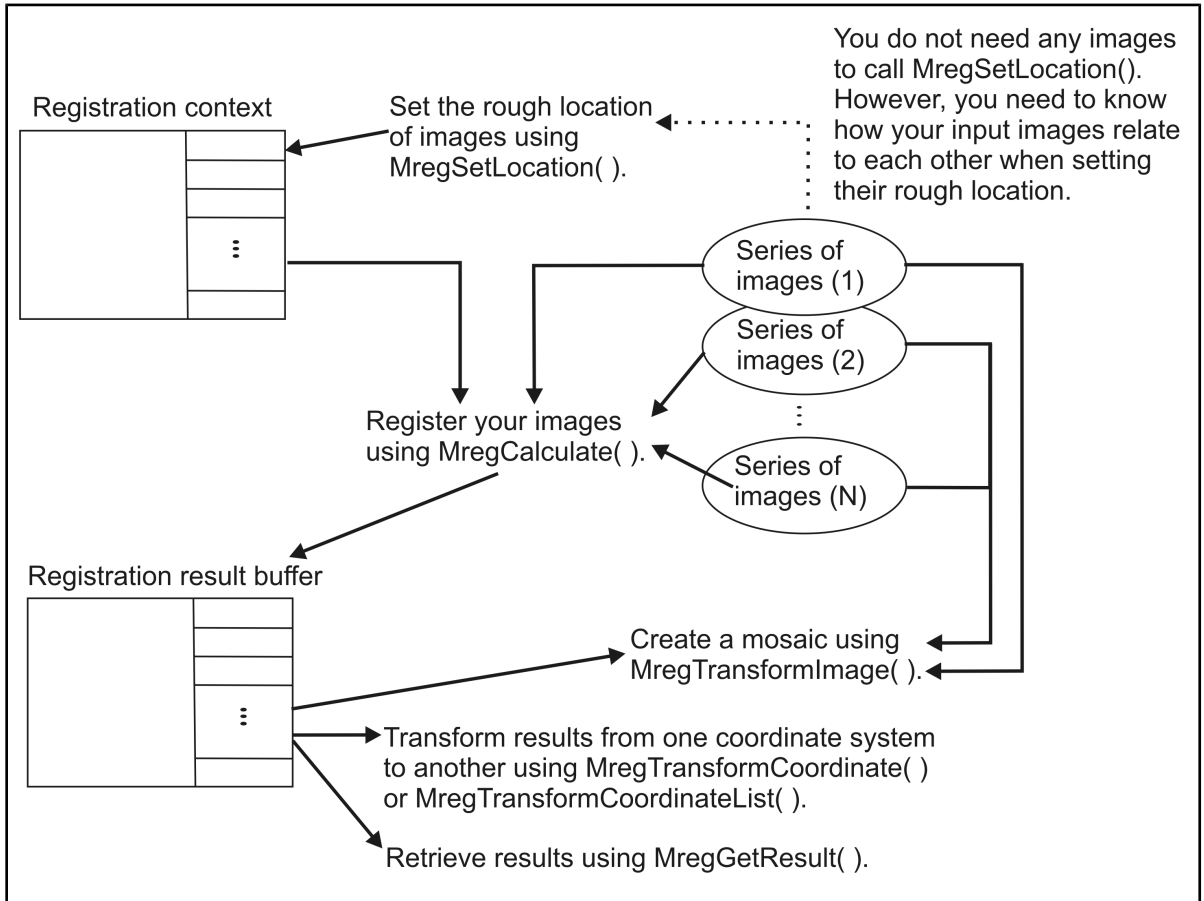
Each image in the image array passed to **MregCalculate()** is associated with the registration element and registration result element whose index matches its own array index. Each registration element controls how the registration of its associated image is performed. Each registration result element stores the results and settings for its associated image. This offers you the flexibility of being able to reuse registration settings and/or results with many different series of input images. For example, when composing a mosaic, images are combined to form one larger image according to the positions calculated during the registration. The freedom to reuse registration results with more than one series of input images allows you to perform the registration calculation once and quickly generate multiple mosaics.

The following diagram illustrates the relationship that exists between the registration elements, the input images, and the registration result elements.



It is important to know the index of the images in the input image array so that you can properly set up their corresponding registration elements. Also, you need to have at least as many registration elements as images. To verify that the number of registration elements is sufficient for your application, use **MregInquire()** with **M_NUMBER_OF_ELEMENTS**. The default number is 256. To increase or decrease the number of registration elements, use **MregControl()** with **M_NUMBER_OF_ELEMENTS**. Note that it is possible to have more registration elements than images; for information on this, see the *Skipping the optimization step* subsection in the *Customizing your registration settings* section in *Chapter 10: Registration*.

The following diagram illustrates how images and registration elements tie in with various functions in the Registration module.



Note that there is no direct link in the diagram between the images and the **MregSetLocation()** function because you set the rough location of your images using registration elements, and not the images themselves. However, knowledge of the relationships between your images is necessary and will be discussed in the *Setting the rough location of your images* section later in this chapter. With the functions that do require input images (**MregCalculate()** and **MregTransformImage()**), you can use more than one series of images with the same registration context or registration result buffer.

Setting the rough location of your images

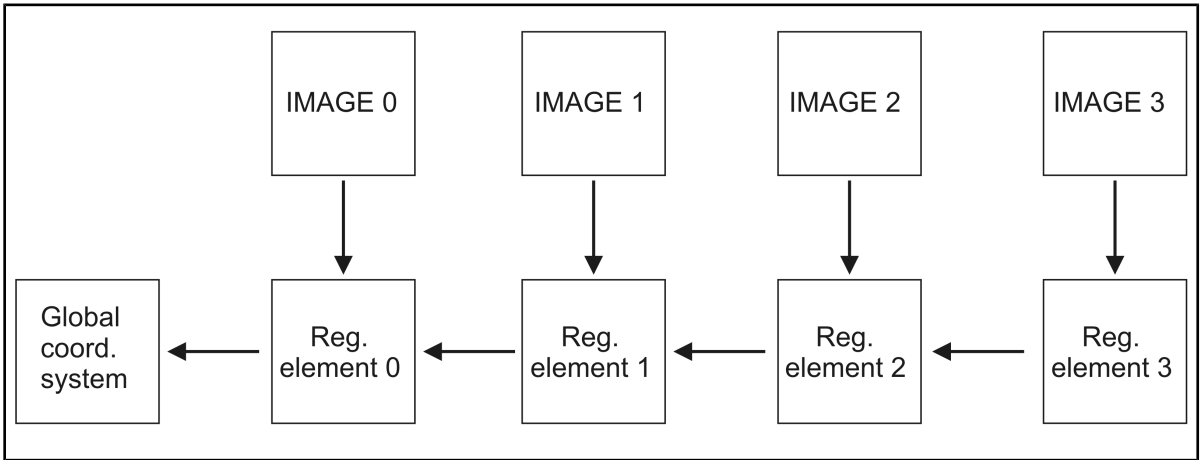
Once you have specified an appropriate number of registration elements, you need to specify the rough location of the images. For every input image, you must call **MregSetLocation()** once and set its rough location by specifying the following:

- The registration element that will store the rough location information for the image.
- Relative to which coordinate system you are specifying the location.
- The transformation that specifies the rough location of the image in this reference coordinate system.

MregSetLocation() also allows you to copy this information from another registration element or from the results of a previous registration operation.

Selecting an image's reference coordinate system

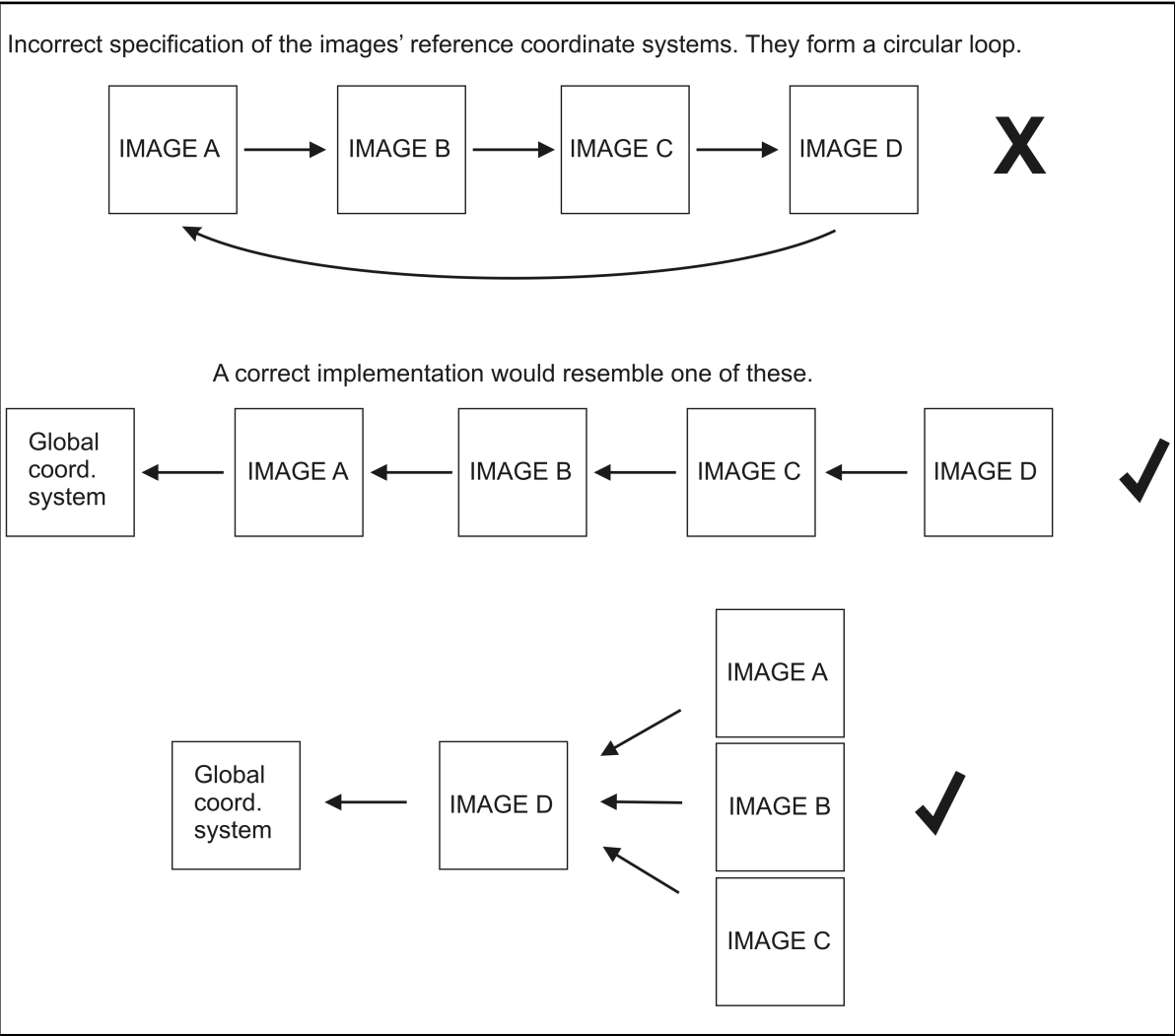
When specifying the rough location, one image must be positioned relative to the global coordinate system; all other images must be positioned relative to another image's coordinate system. This coordinate system is referred to as the image's reference coordinate system. By default, an image's reference coordinate system is that of the registration element's image whose index precedes the current registration element's index. For the image associated with registration element 0, the default reference coordinate system is the global coordinate system. This is illustrated in the image below.



To set an image relative to another image's coordinate system, you must specify the index of that image's registration element when calling **MregSetLocation()**. Also, for the registration to be successful, it is necessary that each image and its reference image overlap. For information on the required minimum overlap between images, see the *Specifying the minimum overlap between images* subsection in the *Customizing your registration settings* section in *Chapter 10: Registration*.

For the image positioned relative to the global coordinate system, the concept of overlapping does not apply. For this image, the transformation that roughly positions it in the global coordinate system is also the optimal transformation. This transformation is copied to the corresponding registration result element when you call **MregCalculate()**.

You must not define the images' reference coordinate systems in a circular manner (see the diagram below).

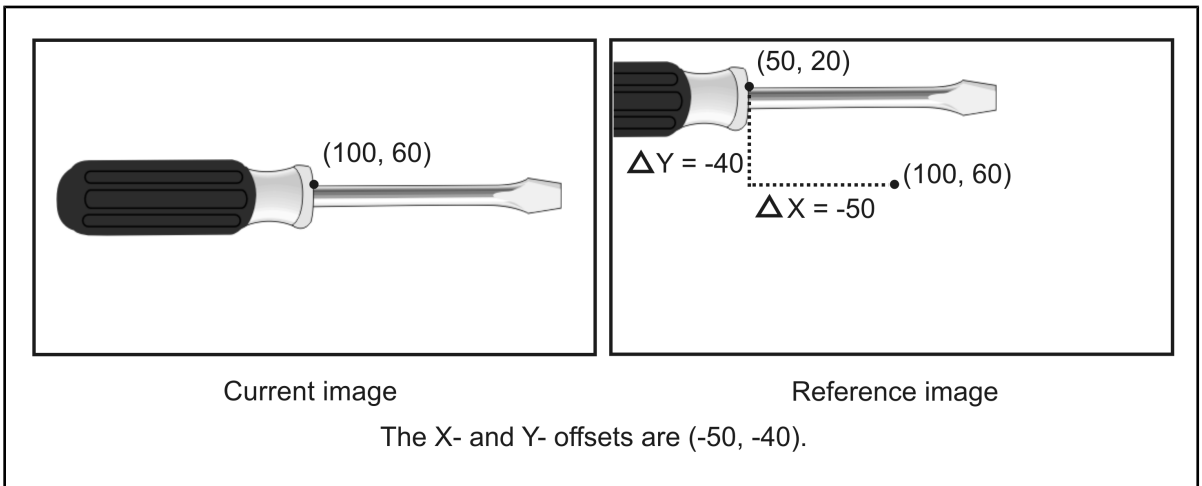


Note that **MregSetLocation()** will not return an error if the reference coordinate systems have been defined in a circular manner. An error will only be generated once you call **MregCalculate()**.

Specifying the type of transformation and its settings

Once you have set each image's reference coordinate system, you must specify the transformation that will map the image into this coordinate system. To do this, you can specify a translation, translation with rotation, or perspective warping transformation.

When you specify a translation transformation (using **MregSetLocation()** with **M_POSITION_XY**), you must specify the X- and Y-offsets between any point in the current image's coordinate system and the same point in its reference coordinate system. These offsets should roughly position the image in its reference coordinate system. The following example illustrates these X- and Y-offsets.



In the case of an image that is rotated relative to its reference coordinate system you can set the rough location via a translation transformation and a rotation transformation (using **MregSetLocation()** with **M_POSITION_XY_ANGLE**).

The other type of transformation that you can specify is a perspective warping, using `MregSetLocation()` with `M_WARP_POLYNOMIAL`, `M_WARP_4_CORNER`, or `M_WARP_4_CORNER_REVERSE`. If you specify a perspective warping, you can specify the mapping between the image's coordinate system and the reference coordinate system in one of the following ways:

- Using a transformation matrix (when using a warping of type `M_WARP_POLYNOMIAL`).
- Using a table containing 4 points in the source image and their corresponding points in the destination image (when using a warping of type `M_WARP_4_CORNER` or `M_WARP_4_CORNER_REVERSE`).

For more information on warping, see the *Warping* section in *Chapter 4: Advanced image processing*.

Note that when you are setting the location of your images, the transformation used to position an image relative to its reference coordinate system can be different from one image to the next. For example, you can position one image using a translation and another image using a perspective warping.

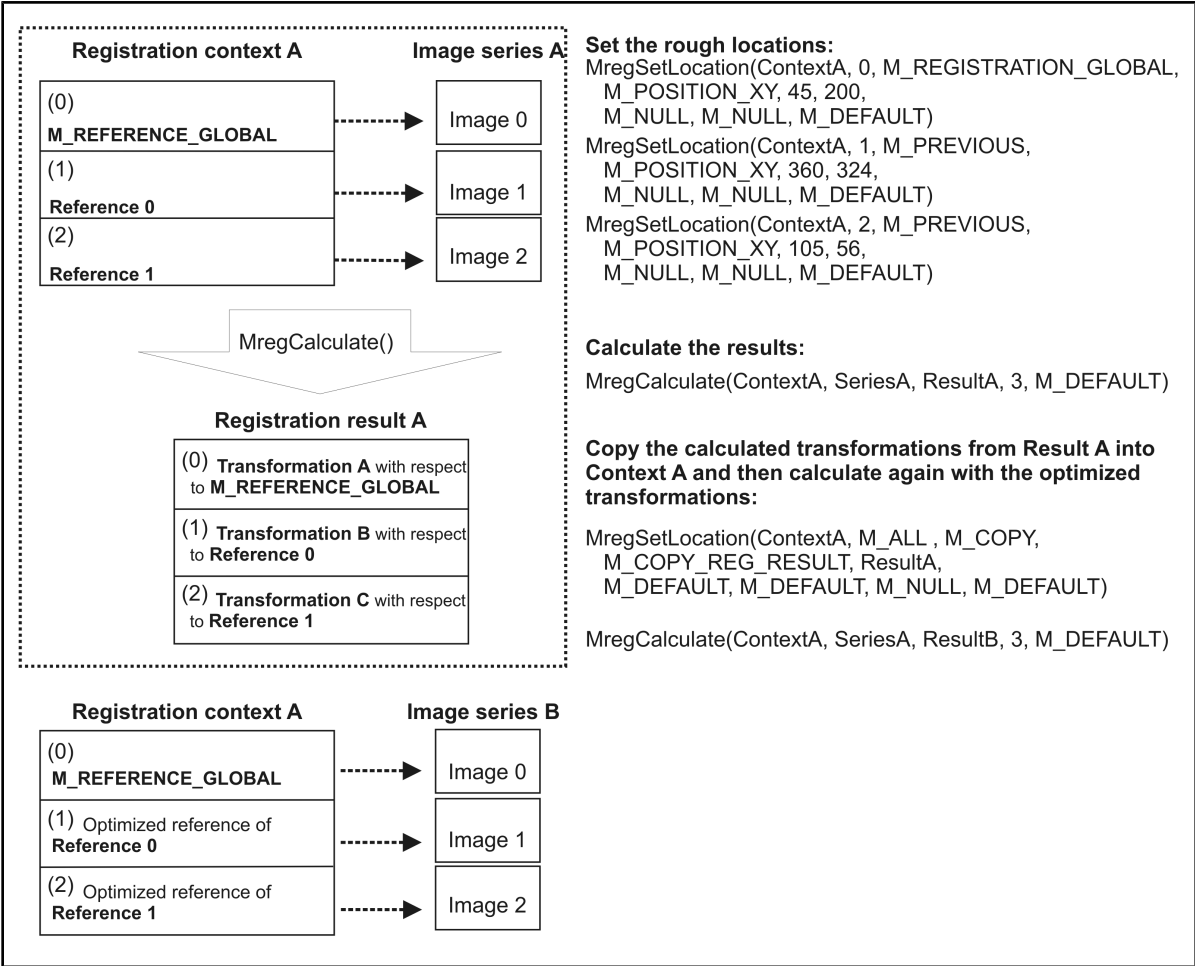
Copying the rough location

In some cases, you might have cameras set up at a fixed distance from one another to take sequential, partially overlapping images of a large object. The transformation information, as it relates to the previous image, is the same for each image. Rather than specifying this information for each image, you can copy this information from one registration element to another, using `MregSetLocation()` with `M_COPY_REG_CONTEXT` or `M_COPY_REG_RESULT`.

In some other cases, you might take several images of the same area to ensure that the registration results are extremely precise. To perform the registration you would probably want to select one of these images as the reference image and obtain registration results for one image in the selected reference image. Then, you would want to use the transformation results of the registration, to specify the rough location of another image relative to the same reference image. Rather than repeatedly getting results to set the rough locations, you can copy this transformation information directly from the registration result element to the registration element using `MregSetLocation()` with `M_COPY_REG_RESULT`.

When copying the transformation information, you can also copy the reference index of the element from which you are copying by setting the **Target** parameter to **M_COPY**. Alternatively, you can specify a different reference index.

The following image shows how you can use **MregSetLocation()** to recursively optimize references by copying optimized transformations into new registration contexts:



The following code snippet demonstrates how using **MregSetLocation()** simplifies copying the transformation information:

```

ElementArray[0] = ReferenceImageId;
ElementArray[1] = GrabImageId;

// WITHOUT M_COPY_REG_RESULT.

// Grab the reference image.
MdigGrab(DigId, ReferenceImageId);

// Set the origin reference
MregSetLocation(ContextId, 0, M_REGISTRATION_GLOBAL, M_POSITION_XY,
                0, 0, M_NULL, M_NULL, M_DEFAULT);

// Continuously grab new images and align them to the reference image.
while (1)
{
    // Grab a new image.
    MdigGrab(DigId, GrabImageId);

    // Calculate the transformation that aligns the grabbed image
    // with the reference image.
    MregCalculate(ContextId, ElementArray, RegResultId, 2, M_DEFAULT);

    // The rough location of the next iteration will be the position
    // that has just been calculated in this iteration.
    MregGetResult(RegResultId, 1, M_POSITION_X+M_REFERENCE, &PositionX);
    MregGetResult(RegResultId, 1, M_POSITION_Y+M_REFERENCE, &PositionY);
    MregSetLocation(ContextId, 1, 0, M_POSITION_XY,
                    PositionX, PositionY, M_NULL, M_NULL, M_DEFAULT);
}

// WITH M_COPY_REG_RESULT.

// Grab a reference image.
MdigGrab(DigId, ReferenceImageId);

// Set the origin reference
MregSetLocation(ContextId, 0, M_REGISTRATION_GLOBAL, M_POSITION_XY,
                0, 0, M_NULL, M_NULL, M_DEFAULT);

// Continuously grab new images and align them to the reference image.
while (1)
{
    // Grab a new image.
    MdigGrab(DigId, GrabImageId);

```

```
// Calculate the transformation that aligns the grabbed image
// with the reference image.
MregCalculate(ContextId, ElementArray, RegResultId, 2, M_DEFAULT);

// The rough location of the next iteration will be the position
// that has just been calculated in this iteration.
MregSetLocation(ContextId, 1, 0, M_COPY_REG_RESULT,
                (MIL_DOUBLE)RegResultId, M_DEFAULT, M_DEFAULT, M_NULL, M_DEFAULT);
}
```

Precision of the rough location and its effect on speed

In general, the more precise the rough location of your images (specified with **MregSetLocation()**), the faster your registration calculation will be. For example, you might know the precise location of your images. If this situation arises, it is possible to skip the optimization step of the registration calculation and only convert the transformations which specify the images' rough locations into the global coordinate system (discussed later in this chapter). This is much faster than having to optimize each transformation in addition to performing the conversion. Alternatively, you might not know anything about your images aside from how they relate to each other. In this case, the rough locations will be very vague and the optimization step of the registration calculation could be relatively long.

Customizing your registration settings

Once you have set the rough location of your images, you can customize how to perform the registration operation, using **MregControl()**. Some of the following settings will affect the registration process in general, whereas others will affect the registration of a specific image.

Selecting the transformation type

You can control the type of transformation that the registration operation can use to optimize the match in the overlapping region, using **MregControl()** with **M_TRANSFORMATION_TYPE**. This is a global setting and it will affect the registration of all your images. The Registration module can optimize the match using a translation, a translation with rotation, a translation with rotation and scale, or a perspective transformation (**M_TRANSLATION**, **M_TRANSLATION_ROTATION**, **M_TRANSLATION_ROTATION_SCALE**, and **M_PERSPECTIVE** respectively). Choosing between the types is not always obvious and will depend on the way the images were taken, as well as the precision used when setting the rough location of the images.

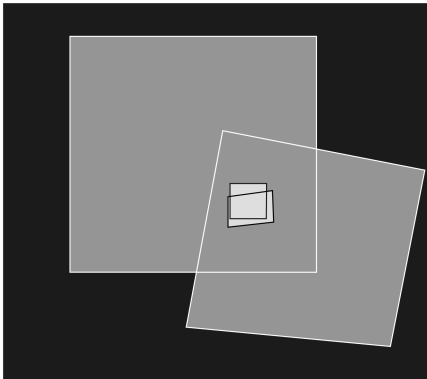
For example, if the camera that captured the images moved only from side to side or up and down, then a translation transformation should be sufficient to optimize the alignment. Alternatively, if the camera made more complex movements when capturing the series of images, such as rotating, tilting, zooming, or panning, a perspective transformation should produce a more accurate alignment.

The optimal transformation depends on how you specified the rough location of your images. Taking the latter example, although the camera made complex movements, if you precisely specified how each image maps into its reference image's coordinate system, you might only need a translation to perfect the alignment.

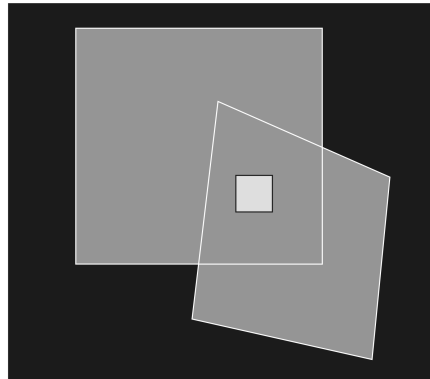
Note that if you are not sure which transformation type to select, select the perspective transformation since it is more general. However, if you are sure a translation will result in an optimal alignment, it is faster to use a translation transformation than it is to use the more general perspective transformation.

The following example shows a pair of images, first placed according to their rough locations and then according to their positions calculated using **MregCalculate()**, in which a perspective warping transformation type was permitted. Notice how the distorted quadrilateral was successfully warped so that it overlaps with the corresponding square in its reference image.

A pair of images in their rough locations.



A pair of images in their calculated positions.



Specifying the maximum allowable displacement

You can set the maximum displacement that the transformation can shift the pixels in the images during registration to optimize the match in the overlapping regions. To do so, use **MregControl()** with **M_LOCATION_DELTA** and specify the maximum displacement as a percentage of the biggest dimension in the images to align. The default setting is 5%. This is a global setting and it will affect the registration of all your images.

In general, the more precise you are in setting the rough location of your images (using **MregSetLocation()**), the faster your registration calculation can be. Being more precise when setting the location of your images allows you to set a smaller value for the maximum displacement, which results in a faster calculation. A more general location means that you must specify a larger maximum displacement; consequently, the algorithm will have to search a much broader area of the images to find an optimal match, making the calculation longer.

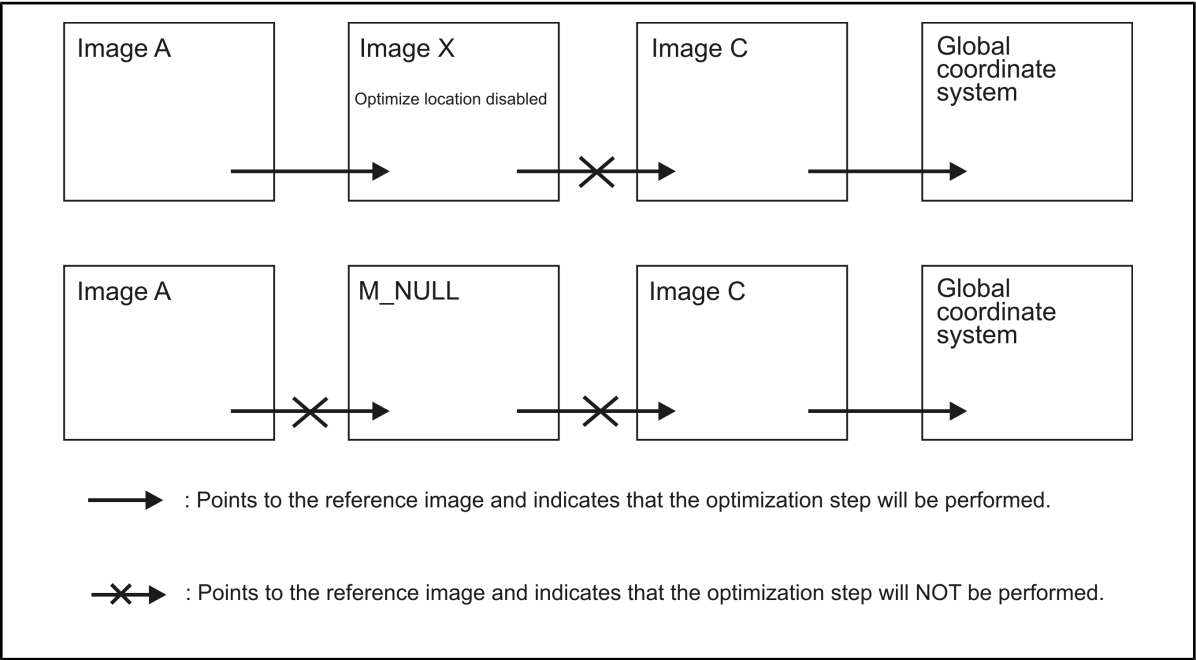
Skipping the optimization step

When the precise location of one or more images is already known, you can bypass the optimization step of the registration calculation. If you choose to do so, for each of these images, the transformation specified in **MregSetLocation()** represents the optimal transformation. When you call **MregCalculate()**, the specified transformation will be converted so that it maps the image into the global coordinate system instead of into its reference image's coordinate system.

There are two ways of bypassing the optimization step of the registration of an image (image X). They differ in how other images using image X as their reference will be registered.

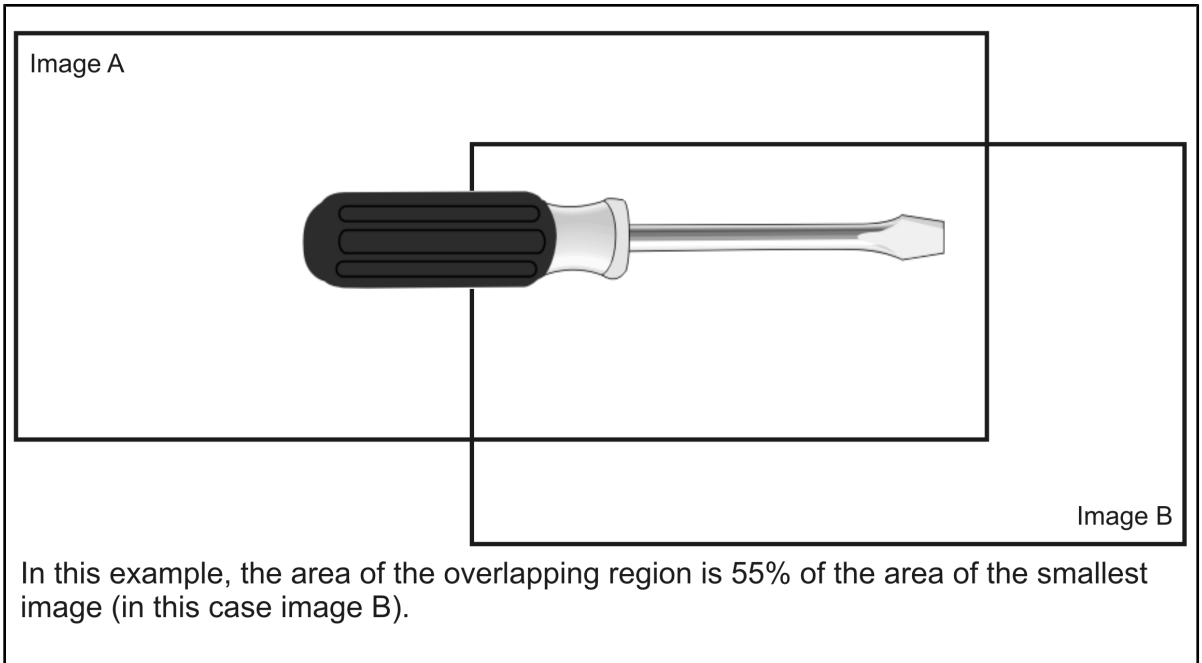
- You can disable the **MregControl()** **M_OPTIMIZE_LOCATION** control type of image X's registration element. If another image (image A) is to be registered with respect to image X, the optimization step of image A's registration will still be performed.
- You can replace image X in the image array by **M_NULL**. Since the registration calculation does not have access to image X, no overlapping region will exist between it and any other images. Therefore, if image A is to be registered with respect to image X, the optimization step of image A's registration will not be performed (even if **M_OPTIMIZE_LOCATION** is enabled). In this case, image A's optimal transformation is the one set with **MregSetLocation()**.

The following illustration gives an example of the two ways of bypassing the optimization step during the registration of an image, and their different effects on other images.



Specifying the minimum overlap between images

To increase the speed and robustness of the registration, you can specify the minimum overlap between each image and its reference image, using **MregControl()** with **M_MIN_OVERLAP**. This is a global setting and affects the registration of all images. Express the overlap as a percentage of the smallest of the two images.



MregSetLocation() will not generate an error if the specified rough location does not comply with the minimum overlap setting. However, when you call **MregCalculate()**, the optimization step of the registration calculation is not performed for this particular image. In addition, you will get **M_FAILURE** if you retrieve the calculation result using **MregGetResult()** with **M_RESULT**, for either the overall registration process or the registration of the individual image.

Accuracy

You can control the accuracy of the registration calculation using **MregControl()** with **M_ACCURACY**. This is a global setting and it will affect the registration of all your images. Registration accuracy can be set to one of the following: **M_LOW** or **M_HIGH**. The default setting is **M_HIGH**. When you set the accuracy to high, the registration calculation is performed with subpixel accuracy. To perform the calculation with pixel accuracy, set **M_ACCURACY** to **M_LOW**.

Setting the origin of an image's coordinate system

In certain cases, it can be useful to individually choose the origin of an image's coordinate system instead of using the image's default coordinate system (the center of the image's top-left pixel). To do so, use **MregControl()** with **M_REFERENCE_X** and **M_REFERENCE_Y** to set the coordinates of the origin to any point inside or outside of the image. An example of when this is useful is when the image is stored in a child buffer, and you know its exact location within the parent buffer. Another example is when images are taken with respect to a moving point, such as the corner of a moving table. In this case, the origin of your images' coordinate systems can be set to that point.

It is very important to be aware of the origin of each image's coordinate system, especially when trying to set the location of your images with **MregSetLocation()**. It is also crucial to know the origin of an image's coordinate system when performing other operations in the Registration module, such as composing a mosaic or transforming coordinates from a source coordinate system to a destination coordinate system.

Retrieving and analyzing results

After having successfully registered your images using **MregCalculate()**, you can extract the required results from your result buffer using **MregGetResult()**.

You can retrieve general results for the entire registration calculation or specific results stored in the registration result elements. The registration result elements are indexed as positive integers, starting from 0, and correspond to the registration elements with the same index. For a complete description of all possible results, refer to the description of **MregGetResult()** in the MIL Reference.

Possible results

Registration produces several types of results which provide information on the calculated positions for the images in the global coordinate system and the success of the registration calculation. Results can be returned for:

- Whether the registration was successful, globally or for an individual image (**M_RESULT**).
- The score of the registration, globally or for an individual image (**M_SCORE**).
- The X- and Y-coordinates of the origin of each image's coordinate system in the global/reference coordinate system (**M_POSITION_X** and **M_POSITION_Y**).
- The coordinates of the four corners of each image in the global/reference coordinate system (**M_TRANSFORMED_...**).
- The identifier of the internal buffer that stores the forward or reverse transformation matrix, used to transform a position in an image into its position in the global/reference coordinate system or vice versa (**M_TRANSFORMATION_MATRIX_ID**, **M_REVERSE_TRANSFORMATION_MATRIX_ID**).

- The forward or reverse transformation matrix data, used to transform a position in an image into its position in the global/reference coordinate system or vice versa (**M_TRANSFORMATION_MATRIX**, **M_REVERSE_TRANSFORMATION_MATRIX**).
- The width and height that the mosaic will have when it is composed (**M_MOSAIC_SIZE_X** and **M_MOSAIC_SIZE_Y**). This information is useful to determine the size of the destination image buffer needed to hold the mosaic.

Using the results

There are several ways that you can use the results of the registration. You can use them to create a mosaic with the images. For more information on mosaicing, see the *Mosaicing* section later in this chapter. Furthermore, you can use

MregTransformCoordinate() or **MregTransformCoordinateList()** to convert a pair or a list of coordinates between two of the following coordinate systems: the global coordinate system, an image's coordinate system, or a mosaic's coordinate system. In addition, you can pass the retrieved identifier of the buffer containing the forward or reverse transformation matrix to **MimWarp()** and use it to warp other images.

Drawing results

Using the **MregDraw()** function, you can draw boxes in a destination image buffer that outline the calculated positions (in the global coordinate system) for the images. You can choose to draw in the display's overlay buffer. By drawing into the display's overlay buffer, you can annotate an image non-destructively (see the *Annotating the displayed image nondestructively* section in *Chapter 20: Displaying an image*).

You can use a previously allocated graphics context (see *Chapter 21: Generating graphics*) to control the drawing color, or use the default graphics context (**M_DEFAULT**).

You can also draw a zoomed version of the boxes that outline the images' positions. To do so, use **MregControl()** with **M_DRAW_RELATIVE_ORIGIN_X**, **M_DRAW_RELATIVE_ORIGIN_Y**, **M_DRAW_SCALE_X**, and **M_DRAW_SCALE_Y**. The relative origin values must be specified in pixels and represent the coordinates in the mosaic's coordinate system that will appear at the top-left corner of the destination buffer. The scale values specify the X- and Y-scaling factors used to fill the destination buffer.

Mosaicing

Once you have performed the registration of images, you can use the results to create a mosaic. Mosaicing combines images to form one larger image or to create an image with improved resolution (super-resolution). To combine the images, mosaicing uses the transformations calculated during the registration. To perform mosaicing, use **MregTransformImage()**.

You can create mosaics from different series of images using the results of one registration calculation. This is useful, for example, if you have a camera that is taking a series of images and the camera's position is moved between each snapshot. If you take another series of images and the camera's position for each individual image is the same as it was for the previous series of images, then you can perform the registration calculation once (for the first series), and re-use the results, stored in the registration result buffer, to compose a mosaic with the subsequent series of images. Note that any slight change between the original camera position and the ones used to take a subsequent series of images will result in a mosaic in which the images are not optimally aligned.

You can also omit certain images from a series of images when producing a mosaic. For example, you might want to add each image to a mosaic as soon as it is grabbed. You can do this by calling **MregTransformImage()**, omitting all the images in the input array except for the grabbed one, and passing the same destination image (the one containing the partial mosaic) upon each call. To omit an image from the input array, replace the identifier of the image by **M_NULL**.

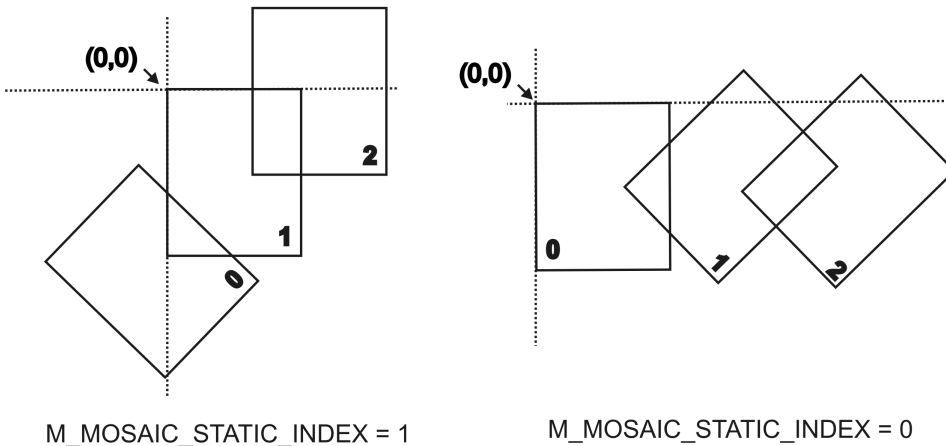
There also exist various registration result buffer settings that affect the composition of your mosaic; you can control their settings using **MregControl()** with **M_GENERAL**. These are discussed in the following subsections.

Positioning and scaling your mosaic in the destination image buffer

You can control the orientation, position, and scale of your mosaic in the destination image buffer using **MregControl()** with **M_MOSAIC_STATIC_INDEX**, **M_MOSAIC_OFFSET_X**, **M_MOSAIC_OFFSET_Y**, and **M_MOSAIC_SCALE** respectively.

You specify the coordinate system relative to which your mosaic will be oriented when it is composed, using **M_MOSAIC_STATIC_INDEX**. To compose the mosaic relative to a specific image, specify the image's registration result element; the origin of the image's coordinate system will be placed at the top-left corner of the destination image (assuming no offsets were specified). In addition, this image will be placed upright in the mosaic, and the other images will be positioned relative to its coordinate system. The other possible choice is to compose the mosaic relative to the global coordinate system, in which case the origin of the global coordinate system will appear at the top-left corner of the destination image buffer (assuming no offsets were specified). Consequently, the images in the mosaic will be oriented according to their calculated positions. The default is the coordinate system of the image associated to the first registration result element.

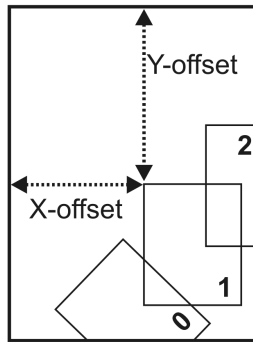
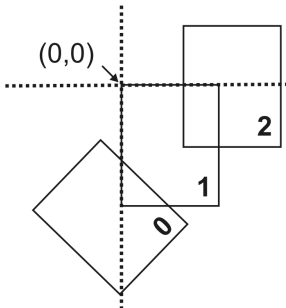
Example of one mosaic oriented in two different ways. The coordinate system of the image specified by **M_MOSAIC_STATIC_INDEX** is shown.



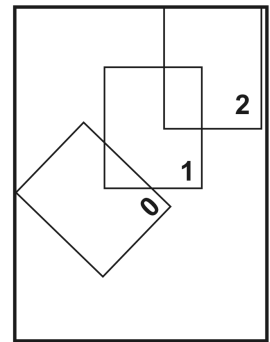
Once you have specified the coordinate system relative to which the mosaic will be composed, you can set the X- and Y-offset between the origin of this coordinate system and the top-left corner of the destination image buffer, using **MregControl()** with **M_MOSAIC_OFFSET_X** and **M_MOSAIC_OFFSET_Y**. This offset allows you to control the location of the mosaic in the destination image buffer. You can compose the mosaic so that the left-most point of the mosaic appears at the complete left of the destination image buffer; to do so, set **M_MOSAIC_OFFSET_X** to **M_ALIGN_LEFT**. Similarly, to place the top-most point of the mosaic at the complete top of the destination image, set **M_MOSAIC_OFFSET_Y** to **M_ALIGN_TOP**. The X- and Y-offsets are illustrated below.

M_MOSAIC_STATIC_INDEX = 1

Destination image buffer

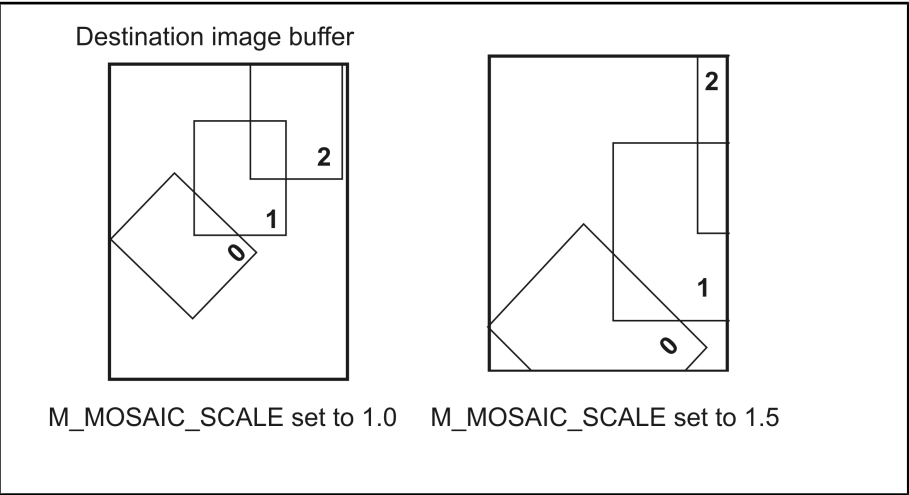


M_MOSAIC_OFFSET_X and
M_MOSAIC_OFFSET_Y set
to positive values.



M_MOSAIC_OFFSET_X and
M_MOSAIC_OFFSET_Y set
to M_ALIGN_LEFT and
M_ALIGN_TOP, respectively.

You can also apply a scale factor to produce an enlarged mosaic, using **MregControl()** with **M_MOSAIC_SCALE**. A value greater than 1.0 produces an enlarged mosaic, while a value less than 1.0 produces a reduced one.

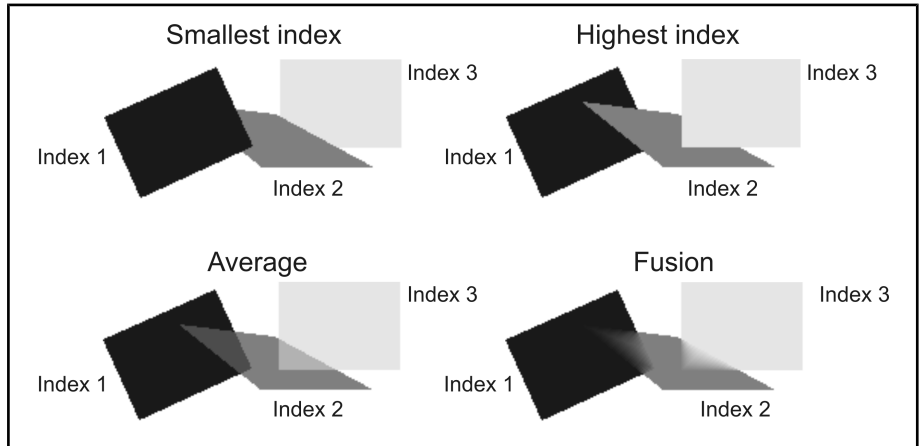


Mosaic composition in the overlapping regions

Although images have been aligned during the registration process to optimize the match in the overlapping regions, the pixels in these overlapping regions are not always superimposed perfectly. Furthermore, the contrast and pixel intensity of one image is not always the same as in the image it overlaps. Therefore, to create a mosaic, you must specify how the Registration module handles the overlapping regions of the images, using **MregControl()** with **M_MOSAIC_COMPOSITION**. You can choose to:

- Use the pixels of the image associated with the registration result element with the lowest index (**M_FIRST_IMAGE**).
- Use the pixels of the image associated with the registration result element with the highest index (**M_LAST_IMAGE**). This is the default.
- Use the average value of the images' pixels in the overlapping region (**M_AVERAGE_IMAGE**).
- Fuse the images by progressively blending overlapping pixels (**M_FUSION_IMAGE**). That is, overlapping pixel values are modified (blended) to form a transitional portion. Blending is based on the distance between each pixel and the edges of the images in the mosaic.
- Use the pixels of all the images to create a new image with enhanced resolution (**M_SUPER_RESOLUTION**).

The following example illustrates the types of mosaic compositions that you can specify to create a larger image:

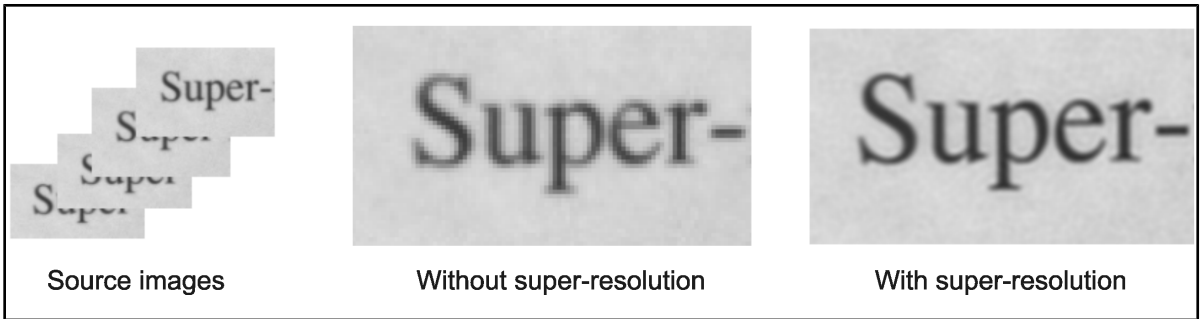


The following section explains how to combine images of the same area to increase their resolution.

Mosaic composition using super-resolution

Super-resolution is a process where multiple source images are used to create a new image with enhanced resolution. Details that are difficult to see in the original source images are extracted and can be seen in the mosaic when it is enlarged. To compose a super-resolution mosaic, use **MregControl()** with **M_MOSAIC_COMPOSITION** set to **M_SUPER_RESOLUTION**. Set **M_MOSAIC_SCALE** to a value greater than one to enlarge the mosaic image.

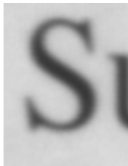
The following shows the comparison of an image resized using bilinear interpolation and the result of a super-resolution mosaic composition. Four source images are taken to create the mosaic and the result is scaled up three times with **M_MOSAIC_SCALE**.



The source images need to be very accurately aligned before the mosaic composition can take place. To ensure that the images are aligned with subpixel accuracy, call **MregControl()** with **M_ACCURACY** set to **M_HIGH** before using **MregTransformImage()**.

During super-resolution calculation, the Registration module tries to remove the effect of blurring caused by your image acquisition setup. For optimal results, you should set **M_SR_PSF_TYPE** to the point spread function (PSF) that best models how a point of light is blurred by the lens and CCD of your acquisition setup. An **M_CIRCULAR** PSF assumes that the setup blurs each point of light as a uniform circle. An **M_GAUSSIAN** PSF assumes the setup blurs each point of light according to a radially symmetric Gaussian function. You must specify a radius

for both of these PSFs using **M_SR_PSF_RADIUS**; for an **M_GAUSSIAN** PSF, the radius corresponds to the standard deviation of the Gaussian function. A correctly estimated PSF radius can enhance the super-resolution mosaic further, while a radius which is estimated too large can create large artifacts.



PSF disabled or
radius set to 0.0.



Correctly estimated
PSF radius



PSF radius estimated
too large

If your source images have noise, super-resolution mosaicing will enhance the noise in the final image. To avoid the amplification of noise, you can specify a smoothness value. However, a smoothing value which is too high can blur the image.



Specified smoothness
value too small



Correctly estimated
smoothness value



Too much smoothing

Registration example

The registration example *Mreg.cpp* illustrates how the module can be used to register images and compose a mosaic.

```

/*****
/*
* File name: MReg.cpp
*
* Synopsis: This program uses the registration module to form a mosaic of
*           three images taken from a camera at unknown translation.
*/
#include <mil.h>
#include <stdlib.h>

/* Number of images to register. */
#define NUM_IMAGES_TO_REGISTER 3

/* MIL image file specifications. */
#define IMAGE_FILES_SOURCE M_IMAGE_PATH MIL_TEXT("CircuitBoardPart%d.mim")

int MosMain(void)
{
    MIL_ID      MilApplication,          /* Application identifier.      */
               MilSystem,                /* System identifier.          */
               MilDisplay,               /* Display identifier.         */
               MilSourceImages           /* Source images buffer identifiers.*/
               [NUM_IMAGES_TO_REGISTER], /* Mosaic image buffer identifier. */
               MilMosaicImage = M_NULL, /* Overlay image.              */
               MilRegContext,            /* Registration context identifier.*/
               MilRegResult;             /* Registration result identifier.*/
    MIL_INT     i;                       /* Iterator.                   */
    MIL_INT     Result;                  /* Result of the registration.   */
    MIL_INT     MosaicSizeX,             /* Size of the mosaic.          */
               MosaicSizeY;
    MIL_INT     MosaicSizeBand,          /* Characteristics of mosaic image.*/
               MosaicType;
    MIL_TEXT_CHAR ImageFilesSource[NUM_IMAGES_TO_REGISTER][260];

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

    /* Load source image names. */
    for (i = 0; i < NUM_IMAGES_TO_REGISTER; i++)
    {
        MosSprintf(ImageFilesSource[i], 260, IMAGE_FILES_SOURCE, i);
    }

    /* Print module name. */

```

```

MosPrintf(MIL_TEXT("\nREGISTRATION MODULE:\n"));
MosPrintf(MIL_TEXT("-----\n\n"));

/* Print comment. */
MosPrintf(MIL_TEXT("This program will make a mosaic from many source images.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Restore the source images. */
for (i = 0; i < NUM_IMAGES_TO_REGISTER; i++)
    MbufRestore(ImageFilesSource[i], MilSystem, &MilSourceImages[i]);

/* Display the source images and prepare for overlay annotations. */
for (i = 0; i < NUM_IMAGES_TO_REGISTER; i++)
{
    MdispSelect(MilDisplay, MilSourceImages[i]);

    /* Pause to show each image. */
    MosPrintf(MIL_TEXT("image %ld.\n"), i);
    MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
    MosGetch();
}

/* Allocate a new empty registration context. */
MregAlloc( MilSystem, M_CORRELATION, M_DEFAULT, &MilRegContext);

/* Allocate a new empty registration result buffer. */
MregAllocResult(MilSystem, M_DEFAULT, &MilRegResult);

/* Set the transformation type to translation. */
MregControl(MilRegContext, M_CONTEXT, M_TRANSFORMATION_TYPE, M_TRANSLATION);

/* By default, each image will be registered with the previous in the list
   No need to set other location parameters. */

/* Set range to 100% in order to search all possible translations. */
MregControl(MilRegContext, M_CONTEXT, M_LOCATION_DELTA, 100);

/* Calculate the registration on the images. */
MregCalculate(MilRegContext, MilSourceImages, MilRegResult,
              NUM_IMAGES_TO_REGISTER, M_DEFAULT);

/* Verify if registration is successful. */
MregGetResult(MilRegResult, M_GENERAL, M_RESULT + M_TYPE_MIL_INT, &Result);
if( Result == M_SUCCESS )
{
    /* Get the size of the required mosaic buffer. */
    MregGetResult(MilRegResult, M_GENERAL, M_MOSAIC_SIZE_X + M_TYPE_MIL_INT,
                  &MosaicSizeX);
    MregGetResult(MilRegResult, M_GENERAL, M_MOSAIC_SIZE_Y + M_TYPE_MIL_INT,
                  &MosaicSizeY);

    /* The mosaic type will be the same as the source images. */

```



```

MbufInquire(MilSourceImages[0], M_SIZE_BAND, &MosaicSizeBand);
MbufInquire(MilSourceImages[0], M_TYPE, &MosaicType);

/* Allocate mosaic image. */
MbufAllocColor(MilSystem, MosaicSizeBand, MosaicSizeX, MosaicSizeY, MosaicType,
               M_IMAGE+M_PROC+M_DISP, &MilMosaicImage);

/* Compose the mosaic from the source images. */
MregTransformImage(MilRegResult, MilSourceImages, MilMosaicImage,
                  NUM_IMAGES_TO_REGISTER, M_BILINEAR+M_OVERSCAN_CLEAR, M_DEFAULT);

/* Display the mosaic image and prepare for overlay annotations. */
MdispSelect(MilDisplay, MilMosaicImage);
MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);
MgraColor(M_DEFAULT, M_RGB888(0, 0xFF, 0));

/* Pause to show the mosaic. */
MosPrintf(MIL_TEXT("mosaic image.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Draw the box of all source images in the mosaic. */
MregDraw(M_DEFAULT, MilRegResult, MilOverlayImage, M_DRAW_BOX, M_ALL, M_DEFAULT);

/* Draw a cross at the center of each image in the mosaic. */
for (i = 0; i < NUM_IMAGES_TO_REGISTER; i++)
{
    MIL_DOUBLE   SourcePosX,
                 SourcePosY,
                 MosaicPosX,
                 MosaicPosY;
    MIL_INT      MosaicPosXMilInt,
                 MosaicPosYMilInt;

    /* Coordinates of the center of the source image. */
    SourcePosX = 0.5 * (MIL_DOUBLE)MbufInquire(MilSourceImages[i], M_SIZE_X, NULL);
    SourcePosY = 0.5 * (MIL_DOUBLE)MbufInquire(MilSourceImages[i], M_SIZE_Y, NULL);

    /* Transform the coordinates to the mosaic. */
    MregTransformCoordinate(MilRegResult, i, M_MOSAIC, SourcePosX, SourcePosY,
                           &MosaicPosX, &MosaicPosY, M_DEFAULT);
    MosaicPosXMilInt = (MIL_INT)(MosaicPosX + 0.5);
    MosaicPosYMilInt = (MIL_INT)(MosaicPosY + 0.5);

    /* Draw the cross in the mosaic. */
    MgraLine(M_DEFAULT, MilOverlayImage, MosaicPosXMilInt - 4, MosaicPosYMilInt,
             MosaicPosXMilInt + 4, MosaicPosYMilInt);
    MgraLine(M_DEFAULT, MilOverlayImage, MosaicPosXMilInt, MosaicPosYMilInt - 4,
             MosaicPosXMilInt, MosaicPosYMilInt + 4);
}

MosPrintf(MIL_TEXT("The bounding boxes and the center of all source images\n"));

```

```

        MosPrintf(MIL_TEXT("have been drawn in the mosaic.\n"));
    }
else
{
    MosPrintf(MIL_TEXT("Error: Registration was not successful.\n"));
}

/* Pause to show results. */
MosPrintf(MIL_TEXT("\nPress <Enter> to end.\n\n"));
MosGetch();

/* Free all allocations. */
if (MilMosaicImage != M_NULL)
    MbufFree(MilMosaicImage);
MregFree(MilRegContext);
MregFree(MilRegResult);
for (i = 0; i < NUM_IMAGES_TO_REGISTER; i++)
    MbufFree(MilSourceImages[i]);

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

return 0;
}

```

Chapter

11

Optical character recognition

This chapter presents the features of the Optical Character Recognition (OCR) module, which performs template-based character recognition.

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 is template-based and provides a powerful and easy to use function set for reading and verifying mechanically generated character strings in 8-bit grayscale images, providing results such as quality (match) scores and validity flags. The module is especially designed to operate on character strings in degraded images, with up to $\pm 180^\circ$ of rotation in the target string. The OCR module can be used in conjunction with other MIL functions to develop hardware independent OCR programs for machine vision applications.

The module offers the choice of loading a set of grayscale character representations from a MIL font file, or gathering the needed grayscale character representations from image buffers and/or other files to create a MIL font. 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 and its ASCII value is returned. You can adjust the value at which this match is considered a success by setting the acceptance level. A read operation yields a string of ASCII characters, a confidence score for each character, and its position in the target string, as well as a confidence score for the whole string. A verify operation also provides a validity flag for each character.

For applications requiring custom font types, the OCR module supports the creation of custom MIL fonts that can be saved to disk and restored as needed. Two predefined MIL font files are provided to read and verify semiconductor wafer serial numbers of standard SEMI font character types.

The module also supports user-specified character constraints, the automatic resizing of a font to better match the target characters (automatic calibration), and the reading of texts that span multiple lines. To ensure recognition accuracy, checksum calculations are performed when analyzing standard SEMI font character strings. For strings of custom font types, user-defined functions can validate the read character string automatically (for example, using custom checksum functions).

Steps to reading or verifying a string in an image

The following steps provide a basic methodology for using the MIL Optical Character Recognition module:

1. Create a custom OCR font context using **MocrAllocFont()** and then use either **MocrCopyFont()** or **MocrImportFont()** to add character representations to the OCR font context.
2. If necessary, specify the type of characters (alphabetic, numeric, or other) that should appear at specific positions in the string, using **MocrSetConstraint()**.
 - ❖ With custom fonts, you can hook a custom validation check function to the read/verify operations using **MocrHookFunction()**. After checking character constraints on a found string, the validity function will be executed. If using an OCR font context allocated for a semi font, checksum calculations are performed automatically.
3. If necessary, adjust general processing controls to fit your application, using successive calls to **MocrControl()**.
4. Calibrate the OCR font context automatically to match the target image's character size and spacing, using **MocrCalibrateFont()**. Alternately, you can set these values manually using **MocrControl()**.
 - ❖ To physically modify the polarity and sizing of the font of an OCR font context, use **MocrModifyFont()**.
5. Preprocess the OCR font context with **MocrPreprocess()** once all the constraints and processing controls are set.

6. Allocate a result buffer using **MocrAllocResult()**. This buffer will be used to store subsequent read or verify result values.
7. Acquire or load a target image. Optionally, limit the area to be read and/or improve the quality of the image using some other MIL module. Note that OCR can only process 8-bit unsigned buffers.
- ❖ The cleaner the background of a target image, the better the results of a read/verify operation.
8. Read or verify the string in the target image, using **MocrReadString()** or **MocrVerifyString()**, respectively. These functions are performed according to the defined character constraints and processing controls.
9. Obtain the OCR results using **MocrGetResult()**.
10. Free all your allocated OCR objects using **MocrFree()**.

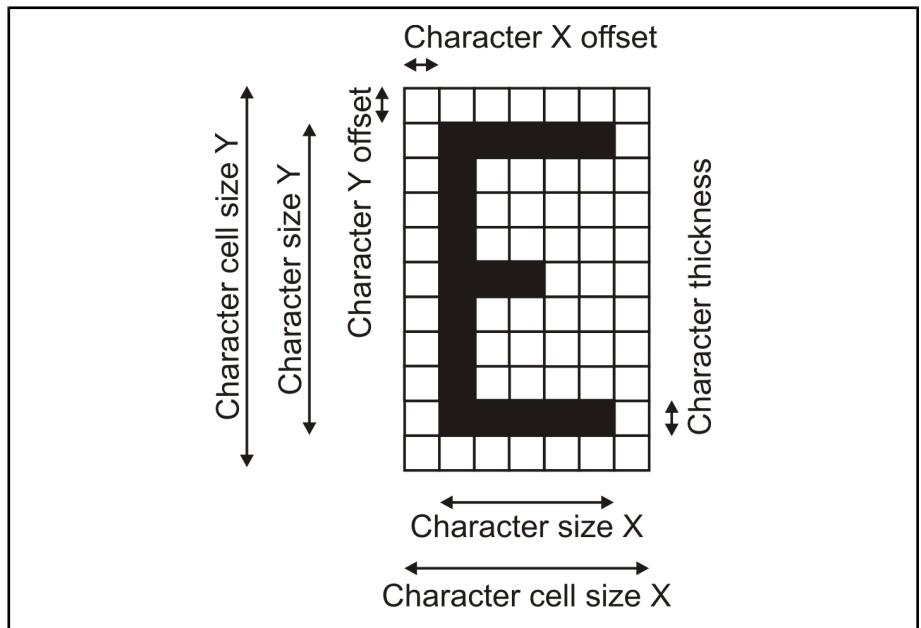
In general, the first five steps are performed once, while steps 6 to 10 are repeated as required. Since you can save the calibration results, character constraints, and processing controls with the character representations (with **MocrSaveFont()**), steps 1 to 5 can be replaced by a single step which loads an OCR font context from disk using **MocrRestoreFont()**.

Before performing a read/verify operation, if any font-specific control or target constraint changes, the OCR font context should be preprocessed. Use **MocrInquire()** to establish if the OCR font context requires preprocessing.

Basic concepts

The basic concepts and vocabulary conventions for the MIL OCR module are:

- **Acceptance level.** The user-defined level against which the match score is compared to determine if the match is valid.
- **Character representations.** The grayscale bitmap (image) or ASCII representation (drawing) of a character used by MIL OCR.
- **Entire text.** One or many strings contained within the target image.
- **Fixed size font.** A series of character representations that each has the same character *Y* size and character *X* size. Characters of a fixed size font tend to be all upper-case.



- **Match score.** The result of a read/verify operation that shows the level of correlation between the character representations in the OCR font and the characters found within the image, taking into account character constraints.

- **OCR font.** The MIL OCR font contains the character representations.
- **OCR font context.** The MIL OCR font context is a container for the OCR font information, target image character size and spacing, constraints, and processing controls.
- **Target string(s).** The text to be found within the target image. This can be as little as a single character or multiple lines of characters.

Guidelines for choosing context types

An OCR font context is a container for the OCR font information, target image character size and spacing, constraints, and processing controls. When allocating your font context (using **MocrAllocFont()**), you can choose 1 of 2 types:

- The **M_GENERAL** OCR font context type.
- The **M_CONSTRAINED** OCR font context type.

General OCR font context type

The **M_GENERAL** OCR font context type requires less information about the target string but works best on clean target images. If you need to specify the location of the string to eliminate erroneous results, use a child buffer to create a smaller region in which to search.

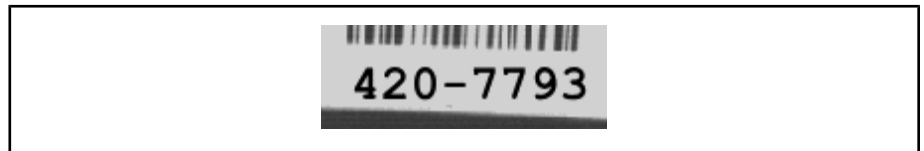
The target image should have a clearly visible threshold (binarization) between the characters and their background. The threshold must preserve the shape of the characters. Broken and/or touching characters can degrade the results.

When using an **M_GENERAL** OCR font context, the cleaner the background of a target image, the better the results of a search. The backgrounds should not have blobs of similar size and intensity as the characters.

The following **MocrControl()** control types are available only when using an **M_GENERAL** OCR font context:

- Enabling blank character recognition (use **M_BLANK_CHARACTERS**).
- Finding the string length automatically (use **M_STRING_SIZE** with **M_ANY**).
- Finding the character width automatically (use **M_TARGET_CHAR_SIZE_X** with **M_SAME**) when using a fixed size font.
- Finding the character height automatically (use **M_TARGET_CHAR_SIZE_Y** with **M_SAME**) when using a fixed size font.
- Finding the inter-character spacing automatically (use **M_TARGET_CHAR_SPACING** with **M_SAME** if it is the same between characters or **M_ANY** if it differs between characters).

An example image best suited to using an **M_GENERAL** OCR font context is:



If your image is greatly degraded or if you know a lot about the location of your target string(s), it is recommended to use the **M_CONSTRAINED** font context type.

Constrained OCR font context type

The **M_CONSTRAINED** OCR font context type works well with degraded target images. This type of context requires more information about the target string, but provides a more robust search. **M_CONSTRAINED** is best used when all the details are known about the target string. Especially, the more known about the target string's location and the size and spacing of its characters, the better the results of a search. Since more precise information is required to calibrate your font with that of the target string, you can automatically calibrate the font with the target string using **MocrCalibrateFont()**. Note that the automatic calibration of a font can only be done when using an **M_CONSTRAINED** OCR font context type.

An example image best suited to using an **M_CONSTRAINED** OCR font context is:

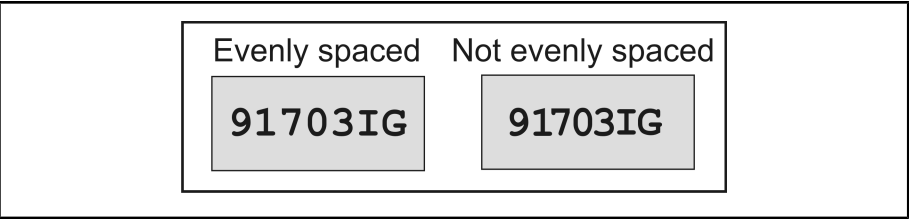


If your image contains a string comprised of unevenly spaced characters or a string that has blanks or broken characters, it is recommended to use the **M_GENERAL** OCR font context type.

Switching between the two

Defining a good OCR font context takes time. Modifying an existing OCR font context is faster than creating a new OCR font context from scratch. When dealing with two sets of images, for example, with similar OCR font requirements with regards to character representation, but differing in some other aspect (for example, sizing, spacing, processing controls, and/or quality), it might be faster to switch OCR font context types, reset the required constraints and controls, preprocess, and then read the new images. You can switch the type of context using the **MocrControl()** function with **M_CONTEXT_CONVERT**.

For example, after reading a series of images with an evenly spaced font (depicted below), to read a group of images where the font is not equally spaced, requires a switch from **M_CONSTRAINED** to **M_GENERAL**.



❖ Note, when you switch between context types, any unsupported setting is reset to its default.

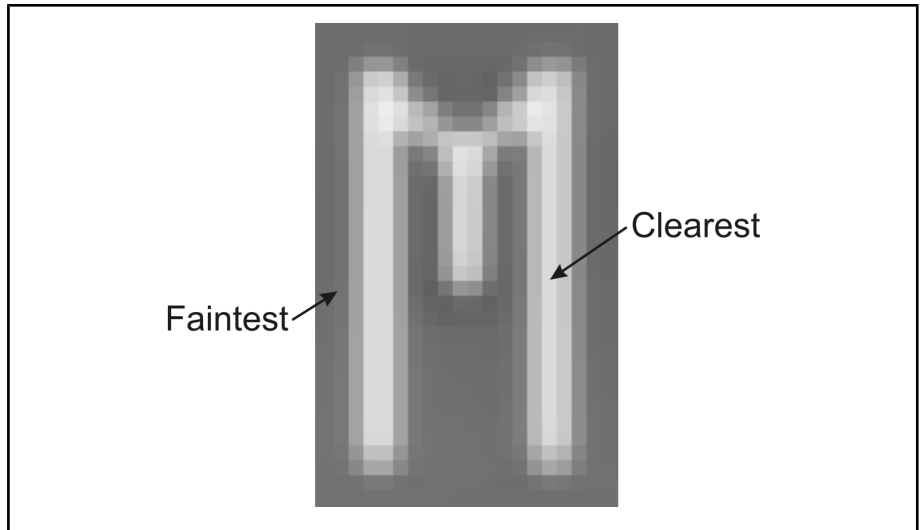
Deciding which OCR font context type to use

The following decision tree should help you decide when you should use an **M_GENERAL** or an **M_CONSTRAINED** OCR font context type. Note that, if a question does not apply to your situation, skip to the next question in the list.

1. Can the target image be thresholded? That is, can a clear distinction be made between the background of the target image and the characters to be read?
 - **Yes.** Use an **M_GENERAL** OCR font context.
 - **No.** Use an **M_CONSTRAINED** OCR font context.
2. Does the target image have intensity problems; that is, does it have reflections or uneven lighting, or is it so heavily degraded that bits of the string are difficult to read with the naked eye?
 - **Yes.** Use an **M_CONSTRAINED** OCR font context.
 - **No.** Use an **M_GENERAL** OCR font context.
3. Is the target string made up of characters that vary in size? Does the target string contain variable spacing?
 - **Yes.** Use an **M_GENERAL** OCR font context.
 - **No.** Use an **M_CONSTRAINED** OCR font context.

Note that variable spacing does not include "jitter", which refers to tiny variations in the position of the target characters relative to the expected character position. Jitter can be corrected by calibrating your OCR font context (with **MocrCalibrateFont()**) to match the target string.

Set **TargetCharSizeXMin** and **TargetCharSizeYMin** to measure to the clearest edge of the target characters to be read. Set **TargetCharSizeXMax** and **TargetCharSizeYMax** to measure to the faintest edge of the target character to be read - just before the shading (if any) matches the background color.



4. Are any of the characters to be read broken?

When there are broken characters, the best way to determine which OCR font context to use is to try both the **M_GENERAL** and the **M_CONSTRAINED** contexts. In a specific situation, one or the other can produce better results.

If using an **M_GENERAL** OCR font context, use **MocrControl()** with **M_BROKEN_CHAR** set to **M_ENABLE**.

5. Are any of the characters to be read touching?

When there are touching characters, the best way to determine which OCR font context to use is to try both the **M_GENERAL** and the **M_CONSTRAINED** contexts. In a specific situation, one or the other can produce better results.

If using an **M_GENERAL** OCR font context, use **MocrControl()** with **M_TOUCHING_CHAR** set to **M_ENABLE**.

If your answers were a combination of both **M_GENERAL** and **M_CONSTRAINED** OCR font contexts, use the OCR font context most often recommended. You will have to further manipulate your target image or your OCR font (mostly using **MocrControl()**), to improve results.

OCR font

You must specify the MIL OCR font to read/verify the character strings in target images. MIL uses fonts (or typesets) to specify the style and size of characters in the images to be read or verified.

An OCR font contains the following information:

- The grayscale representations of the characters.
- Codes identifying each character (ASCII codes for the characters).
- The number of characters in the OCR font.
- Character dimensions.

The above information can be calibrated (with **MocrCalibrateFont()** or **MocrControl()**), modified (with **MocrModifyFont()**), and saved (with **MocrSaveFont()**) for later restoration.

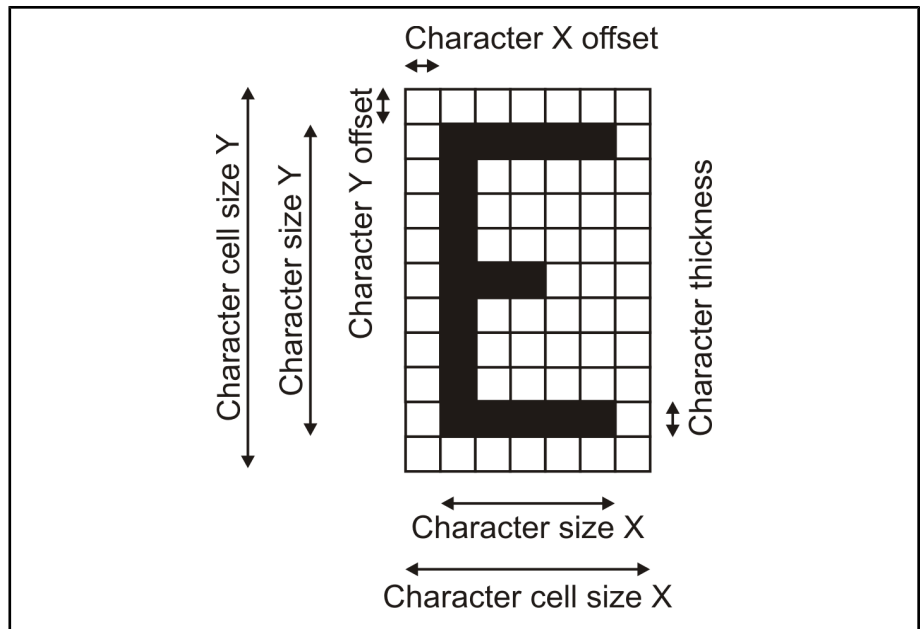
User-defined MIL OCR fonts

Unless using one of the two provided predefined OCR fonts, you must define a custom OCR font. You can create a user-defined OCR font from scratch, or make minor modifications to an existing OCR font, already saved to disk.

To create a custom font:

1. Allocate an OCR font context, using **MocrAllocFont()**.
 - ❖ When allocating an OCR font context, you must specify the maximum number of characters that can be stored in the font, and the dimensions of the font's character representations and their character cells.
2. Grab/create the grayscale character representations of the font in a MIL image buffer and then copy them from the image buffer to the OCR font context, using **MocrCopyFont()**. Alternatively, you can import grayscale character representations from a text file or an image file (for example, a TIFF) into an OCR font context using **MocrImportFont()**.
 - ❖ Note that, for best effect, try using the MIL OCRReader utility, or an interactive tool such as Matrox Inspector, to determine font character widths and heights. Interactive graphic tools are also good for creating new characters to add to an existing font.
 - ❖ When importing, or copying, character representations to the OCR font context, the OCR font context must have sufficient space to hold the representations of all the specified characters. You can use **MocrInquire()** to determine the maximum number of characters that can be stored in the font and the size of each character.
3. Once copied or imported into an OCR font context, the entire OCR font context can be saved on disk using **MocrSaveFont()**, and then later restored using **MocrRestoreFont()**.

The following is an example of a character in its MIL OCR font character cell and the dimensions that you will have to specify during OCR font context allocation. Values are to be specified in pixels. Each square in the grid represents one pixel.



The parameters **CharCellSizeX**, **CharCellSizeY**, **CharOffsetX**, **CharOffsetY**, **CharSizeX**, and **CharSizeY** of function **MocrAllocFont()** must comply with the following restrictions:

- $2 * \text{CharOffsetX} + \text{CharSizeX} \leq \text{CharCellSizeX}$.
- $2 * \text{CharOffsetY} + \text{CharSizeY} \leq \text{CharCellSizeY}$.
- **CharCellSizeY**, **CharCellSizeX**, **CharSizeY**, **CharSizeX** must be ≥ 6 pixels and ≤ 256 pixels.

When copying the character representations from an image buffer, or importing them from an image file, the characters must have the dimensions specified during OCR font context allocation.

This information breaks down into the following:

Row	Description
01	Specifies ASCII file format.
02	Blank row.
03	Specifies the start of a new character representation and its associated (generally ASCII) character.
04 to 36	Specifies the alpha-numerical representation of the character.
37	Blank row.
38	Specifies the start of a new character representation and its associated (generally ASCII) character.
39 to 71	Specifies the alpha-numerical representation of the character.
etc.	This pattern is repeated for every character in the font.

An example

For an example on how to create a user-defined MIL OCR font, see the *Optical character recognition examples* section later in this chapter.

Existing MIL OCR fonts

Once created, a MIL OCR font can be saved and restored as needed. Restoring this information (with **MocrRestoreFont()**) rather than creating the MIL OCR font from scratch saves time, especially if the restored font requires no further modifications. Note that the entire OCR font context is restored when restoring the font using **MocrRestoreFont()**.

Semi fonts

MIL OCR comes with two ISO compatible Semi fonts (**M_SEMI_M12_92** or **M_SEMI_M13_88**) and one generic Semi font that has no constraints and no checksum (*SEMI.mfo*). These can be used directly or modified to suit your needs.

Using a Semi font

To use a Semi font directly, restore it using **MocrRestoreFont()** with the **Filename** parameter set to "*SEMI_M12-92.mfo*", "*SEMI-M13-88.mfo*", or "*SEMI.mfo*". These files are located in directory "*\contexts*" under the MIL installation folder. Once restored, it can be modified using the MIL OCR functions.

Creating a Semi font

To create a new font based on a SEMI font:

1. Create an OCR font context using **MocrAllocFont()** with:
 - The **FontType** parameter set to either **M_SEMI_M12_92** or **M_SEMI_M13_88**.
 - The **StringLength** parameter must be set to 12 when using **M_SEMI_M12_92** and 18 when using **M_SEMI_M13_88**.
 - The **CharNumber** parameter must be set to 38. This allows for capital letters (A-Z), digits (0-9), hyphen (-), and period (.).
2. Use either **MocrCopyFont()** or **MocrImportFont()** to add character representations from an existing Semi font.

Quality and scale are important

Using high-quality character representations will result in the best results. OCR processing relies on using the cleanest font characters possible.

When using an **M_GENERAL** OCR font context, broken characters and spaces, even if expected in the target string, should not be defined in the font. Instead, you should enable the ability to read broken characters using **MocrControl()** with **M_BROKEN_CHAR**, and/or enable the ability to read spaces using **MocrControl()** with **M_BLANK_CHARACTERS**.

When using an **M_GENERAL** font context type, the threshold between the characters and the background must preserve the shape of the characters and have a clearly-visible point of differentiation (binarization).

If the characters in the target image are brighter than the background (for example, white on black), then the character representations included in your font must also be of characters that are brighter than the background. The foreground is specified at context allocation time (**MocrAllocFont()**) and can be changed later using **MocrModifyFont()** with **M_INVERT**. This changes both the character representations and the setting specified at allocation time.

If the size of the character representations in the font is not the same as those in the target string, you can calibrate the font (discussed later). Alternatively, when the physical size of the character representations of the OCR font differ from those in the target image, changing the size of the character representations of the OCR font could improve the robustness of the search. To change the size, use **MocrModifyFont()** with **M_RESIZE**. Changing the size of the font permanently in the OCR font can be faster than resizing the font before each read/verify operation, as is done when the font is calibrated.

Visualizing

It might be necessary, at some point during application development, to display the character representations of your MIL OCR font. To do so, use **MocrCopyFont()** to copy the character representations to a displayable image buffer.

Erasing characters

To remove a character from the OCR font, use **MocrControl()** with **M_CHAR_ERASE** and specify the ASCII code associated with the character representation to remove. An OCR font can contain a limited number of characters; this number is set during OCR font context allocation. Removing unused or erroneously added characters is the easiest way to assure that these characters will not be used when looking for matches in the target string and that there is space for new characters to be added.

Defining the target strings

Define the target strings in the following ways:

- Calibrating your font helps to match the characters in the target image to those within the MIL OCR font.
- Setting the appropriate processing controls and string information helps better define the search criteria for best results.
- Constraints allow you to limit the search to specific characters within the MIL OCR font.

Calibrating your font

You should either manually or automatically calibrate your MIL OCR font to better match the size of the characters and the inter-character spacing in the target image. The more MIL OCR knows about your target image, the faster it can search for the required string(s) and the more robust the results.

- ❖ Note that in some cases, it might be more efficient to change the size of the actual font than to calibrate.

Automatic calibration

Automatic calibration is only available when you are using an **M_CONSTRAINED** OCR font context.

Use **MocrCalibrateFont()** with a small, given range of sizes in which to search, and it will test all the possible sizes in that range, returning the best match of the width, the height, and the spacing of the characters in your sample target image. The sample target image should be the best possible image with an angle, string length, and number of strings that are representative of the target images.

Automatic calibration is always performed using only the first string with the highest match score found in the sample target image. To calibrate multiple strings, create a child buffer around each. If the string cannot be located in the image, an error is generated. Skipping the string locator step (using **MocrControl()** with **M_SKIP_STRING_LOCATION**) might produce undesirable results. This is discussed later in this chapter.

When you are dealing with long strings, the spacing in your target image should be as accurate as possible.

- ❖ Automatic calibration resets values set during a manual calibration.

Manual calibration

Manual calibration can be done for either **M_GENERAL** or **M_CONSTRAINED** OCR font context types. With an **M_GENERAL** OCR font context type, manual calibration does not require exact numbers.

Use **MocrControl()** to manually set the width (**M_TARGET_CHAR_SIZE_X**), the height (**M_TARGET_CHAR_SIZE_Y**), and spacing (**M_TARGET_CHAR_SPACING**) of the target image characters, in pixels. If the exact measurements are known, manually setting the size of the target characters is faster than an automatic calibration.

When using an **M_GENERAL** OCR font context with a fixed size font and reading a string that has the same spacing between characters, but the size of the spacing is unknown, use **MocrControl()** with **M_TARGET_CHAR_SPACING** set to **M_SAME**. If the spacing is unknown and/or not the same between characters, use **MocrControl()** with **M_TARGET_CHAR_SPACING** set to **M_ANY**.

Note that the scale factors between target character sizes and font character sizes must be between 0.25 and 4.0, inclusive. The following restrictions apply:

- **TargetCharSizeXMin** ($/ \text{CharSizeX}$) ≥ 0.25 .
- **TargetCharSizeYMin** ($/ \text{CharSizeY}$) ≥ 0.25 .
- **TargetCharSizeXMax** ($/ \text{CharSizeX}$) ≤ 4.0 .
- **TargetCharSizeYMax** ($/ \text{CharSizeY}$) ≤ 4.0 .

Setting appropriate processing controls

Each of the following controls can improve the robustness of the search. Note, however, that these controls increase the complexity of the operation and reduce its overall speed.

Blanks

By default, MIL OCR ignores blanks in the target image.

When using an **M_GENERAL** OCR font context type, you can use **MocrControl()** with **M_BLANK_CHARACTERS** to enable the ability to read blank characters. Blank characters should not be included in your MIL OCR font. Once enabled, you must modify your string length to include the number of blanks that you expect in the target image. Blanks before the target string are not counted. A blank space will have the same size and inter-character spacing as all other characters in the MIL OCR font, unless **M_TARGET_CHAR_SPACING** is set to **M_ANY**.

- ❖ Note that **M_BLANK_CHARACTERS** is available only when using an **M_GENERAL** OCR font context.

Broken characters

If the target image contains characters that contain scratches or breaks, the target image contains a broken character. To force MIL OCR to identify these characters as a possible match of the characters within the MIL OCR font, use **MocrControl()** with **M_BROKEN_CHAR** enabled. This will reduce the speed of subsequent read/verify operations but can increase robustness. Broken characters should not be included in a MIL OCR font.

- ❖ Note that, when using an **M_GENERAL** OCR font context, **M_BROKEN_CHAR** must be enabled to read a broken character. In an **M_CONSTRAINED** OCR font context, MIL OCR will automatically try to identify broken characters.

Morphological filtering

When the difference between the foreground and the background is so slight that it causes read or verification errors, increase or decrease the value for morphological filtering, using **MocrControl()** with **M_MORPHOLOGIC_FILTERING**, to improve MIL OCR's chances of finding and identifying the characters within the target string. Experimentation is required to determine the exact number that should be passed to this control.

Thickness and dots

You can thicken the target characters using **MocrControl()** with **M_THICKEN_CHAR** when reading thin characters made up of dots to make the characters easier to find and identify.

Touching characters

If characters in the target image touch each other, or if they are connected to blobs of a similar intensity, enable touching characters, using **MocrControl()** with **M_TOUCHING_CHAR**. This will improve MIL OCR's chances of finding and identifying these characters. This will reduce the speed of subsequent read/verify operations but can increase robustness.

- ❖ Note that, when using an **M_GENERAL** OCR font context, **M_TOUCHING_CHAR** must be enabled to read touching characters. When using an **M_CONSTRAINED** OCR font context, MIL OCR will automatically try to identify touching characters.

Specifying other string information

In some cases, additional information about the target string is required. For example, to search for multiple strings, or to search at an angle, certain control types should be set. In other cases, additional information will increase the robustness of the search.

Number of strings

You can set the search to find multiple strings using **MocrControl()** with **M_STRING_NUMBER**. Multiple strings can only be found if each string resides on a different line within the target image and each line of text does not overlap the previous.

- ❖ Note that, for best results, all strings in an image should be of similar length, have a consistent inter-line spacing, and should start at a similar location along the X-axis.

Identifying the number of strings to read/verify is important when dealing with a target image containing multiple lines. It does not have to be specified for single-string images since the default value is 1.

String lengths

You should specify the length of the strings to be read, to receive reliable results using MIL OCR. The maximum string length must be set at the time of OCR font context allocation. With an **M_GENERAL** OCR font context type, the maximum string length can be set to **M_ANY**. Note, however, that this increases the processing time since the OCR module must then try to calculate the string length based on successful matches between the target image and the MIL OCR font.

The default string length is set to the maximum string length of the OCR font context. When the OCR font context constraints are set (discussed later), specifying a string length can also improve the speed of a following read/verify operation.

- ❖ Note that if **M_SEMI_M12_92** is used, the string length must be 12. If **M_SEMI_M13_88** is used, the string length must be 18.

If the entire text of the target image contains one or more strings that differ significantly in length, use an **M_GENERAL** OCR font context for fast results. For more robust results, allocate separate **M_CONSTRAINED** OCR font contexts for each line of text to be read/verified. Experimentation with both OCR font context types is the only way to determine which provides the best solution for each case.

Often, when using an **M_CONSTRAINED** OCR font context, errors in string length result in unpredictable results.

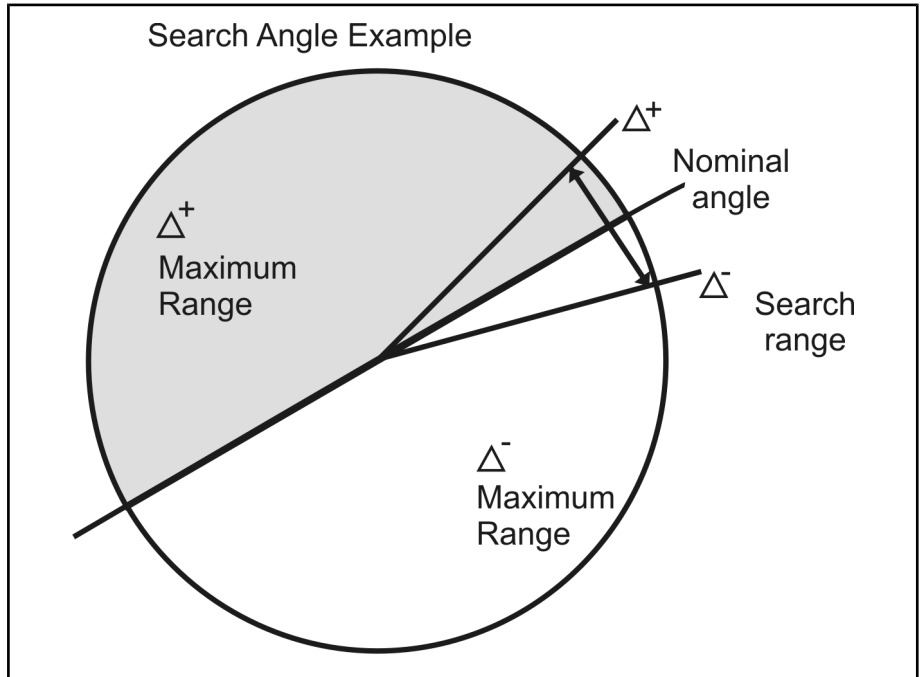
Angle

You can search for the strings within the target image at a specific angle, or through an angular range. The angle of the search is used to locate the string or strings before a read/verify operation. If individual lines in the target image are set at different angles, create a child buffer containing each string at a different angle and read them using a different OCR font context.

- ❖ These child buffers should not overlap.

For each OCR font context, you can specify the specific angle of the search using **MocrControl()** with **M_STRING_ANGLE**. By default, the search angle is 0.0°.

To search through an angular range, use **MocrControl()** with **M_STRING_ANGLE_DELTA_POS** to specify the positive range and/or **M_STRING_ANGLE_DELTA_NEG** to specify the negative range in which to search. A search for a string within the target image through an angular range always starts at the string angle, specified with **M_STRING_ANGLE**. Note that the range in which to search, either in the positive or negative direction, should never be greater than 180°.



If you were searching for a string at approximately 15°, you might want to search between 30° and 0°. To do this, you would set **M_STRING_ANGLE** to 15°, and **M_STRING_ANGLE_DELTA_POS** and **M_STRING_ANGLE_DELTA_NEG** would both be set to 15°. This results in a 30° arc that will be searched for your string.

By default, searching through a range of angles is enabled only if one of the **M_STRING_ANGLE_DELTA...** control types is set to a value other than 0.0°.

❖ Note that, after setting **M_STRING_ANGLE...**, you must use **MocrPreprocess()**.

Positional variation

If the inter-character spacing is not even, you can increase the robustness of the search by specifying the maximum variation in position of the characters. The position variation increases the region being searched for characters within the target image (using **MocrControl()** with **M_CHAR_POSITION_VARIATION_X** and **M_CHAR_POSITION_VARIATION_Y**). These values are relative to the expected position of each 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 for each character in the string, using **MocrSetConstraint()**:

- A digit (**M_DIGIT**): ASCII codes 48 to 57.
 - A letter (**M_LETTER**): ASCII codes 65 to 90 and 97 to 122.
 - An uppercase letter (**M_LETTER + M_UPPERCASE**): ASCII characters 65 to 90.
 - A lowercase letter (**M_LETTER + M_LOWERCASE**): ASCII characters 97 to 122.
 - A character from a specific list of characters, for example A, 1, b, 2. This includes special characters and punctuations, for example ampersand (&), hyphen (-), ellipsis (...).
- ❖ Note that MIL OCR will search for the number of characters specified by the target string length. For best results, the number of characters to be found in your target image should match your string length.

The constraints are stored with the OCR font context as part of its information set and can be inquired, using **MocrInquire()**.

The following is an example of how character constraints are set. For example, the character in the first position should be the letter K and 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, MIL_TEXT("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, MIL_TEXT("12")); /* Must be 1 or 2. */
MocrSetConstraint(OcrFont, 4, M_DIGIT, M_NULL); /* Any digit. */
MocrSetConstraint(OcrFont, 5, M_DEFAULT, M_NULL); /* Any character. */
```

Locating your text

MIL OCR tries to locate the strings to read/verify automatically. Both the angle of the string and the string length information are used to perform the location.

The location step can be skipped to improve the speed of a read/verify operation, using **MocrControl()** with **M_SKIP_STRING_LOCATION**.

When using an **M_CONSTRAINED** OCR font context, you can skip the location step if you create a child buffer or mask that contains only the target string and a minimum amount of space around it. Use other modules in MIL to determine the required buffer size. For more information, refer to the *Manipulating and controlling certain data buffer areas* section in *Chapter 18: Specifying and managing your data buffers*.

When using an **M_GENERAL** OCR font context, string location can be skipped if you are using a clean target image.

Determining what is a match

Acceptance levels

You can set the acceptance level of a successful read/verify operation:

- For each character (**MocrControl()** with **M_CHAR_ACCEPTANCE**).

If the correspondence (also known as the match score) between a character in an image and a character and its constraints in an OCR font context is less than the specified acceptance level, that character is considered invalid and the associated validity flag for that character is set to false.

- For the entire string of characters and the entire text (**MocrControl()** with **M_STRING_ACCEPTANCE**).

If the match score for the entire string passes the acceptance level set for the string, the string is considered valid and its validity flag is set to true. The match score for the entire string is determined by taking the average of the match scores for all characters in that string. The match score for the entire text is the average of the match scores for all the read/verified strings.

A perfect match has a match score of 100%, and no correlation has a match score of 0%. If your images have a lot of noise or distortion, set a lower acceptance level. Poor-quality images increase the chance of false readings and will probably increase the time required to read/verify the character.

- ❖ Perfect matches are highly improbable due to noise obtained during image acquisition.

Unrecognized characters

You can specify the symbol for unrecognized (or invalid) characters (**MocrControl()** with **M_CHAR_INVALID**). If a 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%
Match scores:	70% 15% 53% 24% 80%
Result:	H*L*O

Since the match 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.

Retrieving and analyzing the results

After having potentially located the string in your target image using **MocrReadString()** or **MocrVerifyString()**, you can extract the required results from your result buffer using **MocrGetResult()**. All OCR results are based on the concept of a character. A string is a group of characters that reside along the same line along the same principal axis and the same angle. Multiple strings are a group of strings that are to be read together.

Results are returned in the order in which they are read/verified, from left to right. Always check the validity of the string to ensure that the match score is greater than or equal to the acceptance level, using **MocrGetResult()** with **M_STRING_VALID_FLAG**. Additional checks against the actual length read and the intended length could be made to assure accuracy.

Invalid characters in the resulting strings can be replaced with an ASCII character using **MocrControl()** with **M_CHAR_INVALID**. Note that this control type must be set before performing the read/verify operation.

A character

Character data is returned for each character in the string or strings. This data includes:

- Position along the X- and Y-axis.
- The height and width of the character.
- The match score of the character.
- The spacing of the character.
- The validity of the character (based on the character's acceptance level).
- The ASCII version of the character.

A string

String data is returned for each string. String data includes:

- The contents of the string.
- The angle of the string.
- The length of the string.
- The match score of the string.
- The validity of the string (based on the string's acceptance level).

A text

When reading/verifying multiple strings, text data is returned for all the strings. The text data includes:

- All the characters read, even if they are on different lines in the target image.
 - The number of characters read in total.
 - The match score for the entire text.
 - The threshold value used to binarize the target image. Note that this is only available when using an **M_GENERAL** OCR font context, multiple strings, and/or the string has a few degrees of rotation.
- ❖ All the results for a single string are also available for each string in the entire text.

Understanding odd results

Unexpected results can come from one of the following scenarios.

- If characters are vertically overlapping (or touching).

Create a child buffer that contains the characters below the point where they overlap. This effectively removes the overlapping portion of the characters.

- ❖ The more that characters overlap vertically, the less chance for a valid result.

To read a target image containing multiple lines of text, MIL OCR requires that there is enough space between lines. If lines are too close together (vertically), the robustness of the search will suffer.

Use **MocrControl()** with **M_TOUCHING_CHAR** to read horizontally touching characters.

- If the string or strings have non-uniform inter-character spacing.

Read/verify the string with an **M_GENERAL** OCR font context type and use **MocrControl()** with **M_TARGET_CHAR_SPACING** set to **M_ANY**.

- ❖ This setting is best suited to read non-uniformly spaced characters.

Create a child buffer to contain the most extreme spacing examples for best results.

- If the string or strings in the target image are greatly disparate.

Create a child buffer for each different region in position, angle, and/or length. Read each region individually for best results.

- ❖ The more similar the strings in angle, position, and length, the faster and more robust the search.

If the spacing, length or angle differ, use different MIL OCR font contexts for each child buffer, with their operational and processing controls configured appropriately for each situation. For more information, see the *Defining the target strings* section earlier in this chapter.

- If the target image contains multiple lines with different lengths.

Create a child buffer that contains the area with the shorter strings and read the shorter strings, using a different OCR font context, into a different OCR result buffer.

- ❖ Strings of similar length are easiest to locate.

Make sure that the target image has a consistent inter-line spacing and that all lines start at a similar location along the X-axis, for best results.

- If the target image contains blanks.

Verify that your string length is correct.

- ❖ Your string length determines the number of characters sought.

If the problem persists and the ability to read blank spaces is enabled (use **MocrControl()** with **M_BLANK_CHARACTERS**) remember to include them in your string length (use **MocrControl()** with **M_STRING_SIZE**).

- If the target image contains a lot of non-character blobs.

Try to improve your target image before trying to read/verify again. For more information, see the *Defining the target strings* section earlier in this chapter.

- ❖ An image that does not include blobs of equal intensity as the foreground will return best results.

Use **MocrControl()** with **M_MORPHOLOGIC_FILTERING** to internally enhance the contrast of the image.

- If the target image contains a string longer or shorter than what was expected.
- ❖ MIL OCR will try to return the number of characters that you set as existing in the target image using **MocrControl()** with **M_STRING_SIZE**.

If your string length is set to a value larger than the actual length of the string in the target image, and if using **M_GENERAL** OCR font context, MIL will return the best matches found and stop searching after it has found the specified number of characters. If using an **M_CONSTRAINED** OCR font context, the string length determines the exact number of characters found, even if this means returning characters with low match scores.

- If the target string is at an angle.

Verify that the principal axis of the entire text is equal to the expected angle (nominal angle), specified using **MocrControl()** with **M_STRING_ANGLE**.

- ❖ A string at an angle greater or less than 0 takes longer to find.

Create a target image with a string angle of 0. If the string is properly read, your original application did not follow the expected use of string angle.

Hooking functions

MIL OCR allows you to attach or detach a user-defined function to a MIL OCR event when the specified OCR font context is used. This can be used to impose global string constraints, and can be used to implement custom checksum functions or to reject strings that would have otherwise met the character constraints imposed.

Once your user-defined function is created, use **MocrHookFunction()** to attach it to the validation of a string. It will then execute during the last stage of either **MocrReadString()** or **MocrVerifyString()**. When **MocrHookFunction()** is used in conjunction with an **M_SEMI...** OCR font context type, the function being hooked will replace the default validation function.

Improving search speed

To ensure the fastest possible read or verify operation:

- Use as clean a target image as possible when using an **M_GENERAL** OCR font context.
- Calibrate the OCR font context's character size and target spacing, to match the characters in the target images.
- Preprocess once all the processing controls and constraints are set and before the application's first read/verify operation.
- Reduce the area to read/verify in the target image by creating a child buffer around the target string using **MbufChild...**; the search time is roughly proportional to the area searched.
- Set character constraints using **MocrSetConstraint()**.
- Skip the location step, when possible.

- Set the speed and reliability of the algorithm by setting the robustness factor (**MocrControl()** with **M_SPEED**). For instance, when reading larger characters, robustness can be sacrificed for speed.
- Disable the **M_BLANK_CHARACTERS**, **M_BROKEN_CHAR**, and **M_TOUCHING_CHAR** control types unless absolutely necessary.
- Adjust your image to minimize the angular search range of the read/verify operation.

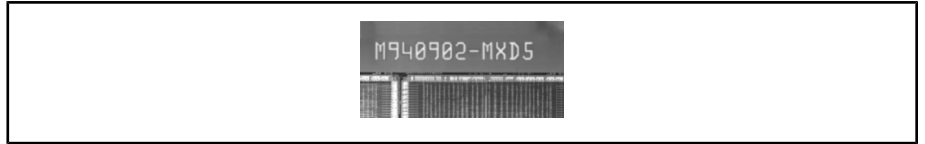
The more you know about the target image, the faster it can be read/verified. This includes:

- Knowing where the string is located within the target image.
- Knowing the height, width, and the inter-character spacing of the string within the target image.
- Knowing the exact number of lines to be read from the target image.
- Knowing the exact string length or at least that all the strings are of similar length.
- Having good examples of each character to be found in the target image so a high-quality MIL OCR font can be made.

MIL OCR can help determine some of this information, for example, **MocrCalibrateFont()** can automatically calibrate the OCR font context's character size to match the characters in the sample target image, and the string location step with **MocrControl()** can find a string within the image, barring any problems.

Optical character recognition examples

The Optical Character Recognition example, *MOcr.cpp* illustrates how the module can be used with the following source image:



Specifically, *MOcr.cpp* shows 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 M12 92 font character string, containing a checksum.

```

/*****
/*
* File name: MOcr.cpp
*
* Synopsis: This program uses the OCR module to read a SEMI font string:
*           the example calibrates a predefined OCR font (semi font)
*           and uses it to read a string in the image. The string read
*           is then printed to the screen and the calibrated font is
*           saved to disk.
*/
#include <mil.h>

/*****
OCR SEMI font read example.
*****/

/* Target image character specifications. */
#define CHAR_IMAGE_FILE      M_IMAGE_PATH MIL_TEXT("OcrSemi1292.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

```

```

/* Font file names. */
#define FONT_FILE_IN          M_IMAGE_PATH MIL_TEXT("Semi1292.mfo")
#define FONT_FILE_OUT        M_TEMP_DIR MIL_TEXT("Semi1292Calibrated.mfo")

/* Length of the string to read (null terminated) */
#define STRING_LENGTH        13L
#define STRING_CALIBRATION   MIL_TEXT("M940902-MXD5")

int MosMain(void)
{
    MIL_ID MilApplication,          /* Application identifier.      */
        MilSystem,                /* System identifier.           */
        MilDisplay,               /* Display identifier.          */
        MilImage,                 /* Image buffer identifier.     */
        MilSubImage,              /* Sub-image buffer identifier. */
        MilFontSubImage,          /* Font display sub image.      */
        MilOverlayImage,          /* Overlay image.               */
        OcrFont,                  /* OCR font identifier.         */
        OcrResult;                /* OCR result buffer identifier.*/
    MIL_TEXT_CHAR String[STRING_LENGTH]; /* Array of characters to read. */
    MIL_DOUBLE Score;             /* Reading score.               */
    MIL_INT  SizeX, SizeY, Type;   /* Source image dimensions.     */

    MosPrintf(MIL_TEXT("\nOCR MODULE (SEMI font reading):\n"));
    MosPrintf(MIL_TEXT("-----\n\n"));

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

    /* Load and display the source image into a new image buffer. */
    MbufAlloc2d(MilSystem,
        MbufDiskInquire(CHAR_IMAGE_FILE, M_SIZE_X, &SizeX),
        MbufDiskInquire(CHAR_IMAGE_FILE, M_SIZE_Y, &SizeY)*3/2,
        MbufDiskInquire(CHAR_IMAGE_FILE, M_TYPE, &Type),
        M_IMAGE+M_PROC+M_DISP,
        &MilImage);
    MbufClear(MilImage, 0);
    MbufLoad(CHAR_IMAGE_FILE, MilImage);
    MdispSelect(MilDisplay, MilImage);

    /* Restrict the region of the image where to read the string. */
    MbufChild2d(MilImage, READ_REGION_POS_X, READ_REGION_POS_Y,
        READ_REGION_WIDTH, READ_REGION_HEIGHT, &MilSubImage);

    /* Define the bottom of the image as the region where to copy the font representation.*/
    MbufChild2d(MilImage, 50, SizeY+10, SizeX-100, (SizeY/3)-10, &MilFontSubImage);

    /* Restore the OCR character font from disk. */
    MocrRestoreFont(FONT_FILE_IN, M_RESTORE, MilSystem, &OcrFont);

    /* Show the font representation. */
    MocrCopyFont(MilFontSubImage, OcrFont, M_COPY_FROM_FONT+M_ALL_CHAR, M_NULL );

```

```

/* Pause to show the original image. */
MosPrintf(MIL_TEXT("The SEMI string at the top will be read using the ")
          MIL_TEXT("font displayed at the bottom.\n\n"));
MosPrintf(MIL_TEXT("Calibrating SEMI font...\n\n"));

/* Calibrate the OCR font. */
MocrCalibrateFont(MilSubImage, OcrFont, STRING_CALIBRATION,
                 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);

/* 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, MIL_TEXT("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, MIL_TEXT("-")); /* - only */
MocrSetConstraint(OcrFont, 8, M_LETTER, MIL_TEXT("M")); /* M only */
MocrSetConstraint(OcrFont, 9, M_LETTER, MIL_TEXT("X")); /* X only */
MocrSetConstraint(OcrFont, 10, M_LETTER, MIL_TEXT("ABCDEFGH")); /* SEMI checksum */
MocrSetConstraint(OcrFont, 11, M_DIGIT, MIL_TEXT("01234567")); /* SEMI checksum */

/* Pause before the read operation. */
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* 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_STRING_SCORE, &Score);

/* Print the result. */
MosPrintf(MIL_TEXT("\nThe string read is: \"%s\" (score: %.1f%%).\n\n"), String, Score);

/* Draw the string in the overlay under the reading region. */
MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);
MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);
MgraFont(M_DEFAULT, M_FONT_DEFAULT_LARGE);
MgraColor(M_DEFAULT, M_COLOR_YELLOW);
MgraText(M_DEFAULT, MilOverlayImage, READ_REGION_POS_X+(READ_REGION_WIDTH/4),
         READ_REGION_POS_Y+READ_REGION_HEIGHT+50, String);

/* Save the calibrated font if the reading score was sufficiently high. */

```

```

    if (Score > READ_SCORE_MIN)
    {
        MocrSaveFont(FONT_FILE_OUT, M_SAVE, OcrFont);
        MosPrintf(MIL_TEXT("Read successful, calibrated OCR font was saved to disk.\n"));
    }
    else
    {
        MosPrintf(MIL_TEXT("Error: Read score too low, calibrated OCR font not saved.\n"));
    }
    MosPrintf(MIL_TEXT("Press <Enter> to end.\n\n\n"));
    MosGetch();

    /* Clear the overlay. */
    MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

    /* Free all allocations. */
    MocrFree(OcrFont);
    MocrFree(OcrResult);
    MbufFree(MilSubImage);
    MbufFree(MilFontSubImage);
    MbufFree(MilImage);
    MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);

    return 0;
}

```


Chapter

12

String Reader

This chapter explains how to perform feature-based character recognition with the MIL String Reader module.

MIL String Reader module

The MIL String Reader module is a set of powerful functions that allow you to perform feature-based character recognition. Unlike the MIL Optical Character Recognition module, which is template-based, String Reader's feature-based technology makes it invariant to changes in scale, aspect ratio, and contrast. It is also considerably tolerant towards variations in perspective and angle, and allows you to quickly determine why your application is not reading the string you expect. In addition, String Reader is designed to support multiple user-defined grammar rules, multi-font definitions, and fontless string reading, making applications simple to write and maintain.

Given its feature-based technology, String Reader proves itself most useful when solving problems that have a limited amount of information, or information that tends to vary. For example, when writing an Automatic Number Plate Recognition (ANPR) application, you will probably have to contend with bad lighting, poor contrast, and variations in scale, perspective, and font. String Reader provides you with the tools to solve such problems.



Font defined in source image



License plate read in target image

String Reader uses a set of specified string models to locate and read strings. Each string model serves as a template, defining the rules a string must follow for it to be read. String Reader reads strings in grayscale images, and provides numerous results, such as the string's score and the character's value. String Reader also offers many settings that allow you to tailor the read algorithm to meet your specific needs. For example, by adjusting the character's acceptance, you can have precise control over which strings are accepted by the read operation. You can also specify a font that the read string must match to be valid or you can specify that the read strings do not have to match any specific font.

Steps to reading a string in an image

The following steps provide a basic methodology for using the MIL String Reader module:

1. For a font-based context, allocate a String Reader context to hold your string models and fonts, using **MstrAlloc()**. For a fontless context, restore a predefined context, using **MstrRestore()**.
2. For a font-based context, add the required fonts to the context and customize them, using **MstrControl()** and **MstrEditFont()** respectively. For a fontless context, specify the general characteristics of the characters to search for in the target image, using **MstrControl()**.
3. Allocate a String Reader result buffer to hold the results of the read operation, using **MstrAllocResult()**.
4. Add a string model to the String Reader context, using **MstrControl()** with **M_STRING_ADD**. Note that you can add more than one string model to the context.
5. Set the maximum number of strings to read, using **MstrControl()** with **M_STRING_NUMBER**.
6. Set the minimum and maximum number of expected characters for the string models, using **MstrControl()** with **M_STRING_SIZE_MIN** and **M_STRING_SIZE_MAX**.
7. If necessary, set character constraints, using **MstrSetConstraint()**.

8. If necessary, adjust general context controls, using **MstrControl()**.
 9. Preprocess the String Reader context, using **MstrPreprocess()**.
 10. If necessary, fix read problems, using **MstrExpert()**.
 11. Perform the read operation in the specified target image, using **MstrRead()**.
 12. Retrieve the required results from the String Reader result buffer, using **MstrGetResult()**.
 13. If necessary, draw the results, using **MstrDraw()**.
 14. If necessary, save your String Reader context, using **MstrSave()**.
 15. Free all your allocated objects, using **MstrFree()**.
- ❖ Several String Reader functions require that you pass a null-terminated string array of the required characters. These characters can either be ASCII or Unicode. To specify which, use **MstrControl()** with **M_ENCODING**. Whenever you pass a string array, you must ensure that it is of the right type for the encoding scheme selected. For more information, see the *Encoding* subsection in the *Global context settings* section in *Chapter 12: String Reader*.

Basic concepts

The basic concepts and vocabulary conventions for the MIL String Reader module are:

- **Character.** Any ASCII or Unicode symbol, such as a letter or a digit. A group of characters is used to define a font. Note that a space is not considered a character.
- **Punctuation character.** Typically categorized as characters that are not letters or numbers, such as the hyphen ('-'). To be considered part of the string, a punctuation character must fall within the range of at least one regular character's Y-size.

- **Read region.** A region in the target image enclosing the string that has been read. This region can be defined as the string's box, which includes the maximum number of spaces before and after the string.
- **Regular character.** A character that is either a letter or a number.
- **Source image.** The image from which a font can be defined. Note that fonts can also be defined from fonts installed on your system (for example, Times New Roman and Arial).
- **Space.** The space is used to delimit multiple strings that are on the same line. Strings on separate lines are automatically considered separate strings. Note that String Reader does not consider a space a normal character. A space is therefore never read.
- **String.** A linear sequence of regular characters. If the distance between two successive characters is greater than the maximum space allowed, these two characters are considered part of two separate strings. Note that characters on separate lines can never be part of the same string.
- **String model.** A container, within the String Reader context, that holds all the settings and constraints that control how to read one or many specific strings in the target image.
- **String Reader context.** The container for all string models and fonts. The String Reader context also contains global read settings that apply to the character recognition algorithm. Based on the settings of the context, string models, and fonts, strings in the target image are either accepted or rejected by the read operation.
- **String Reader font.** A container, within the String Reader context, that holds a list of characters (ASCII or Unicode) with each character's corresponding representation. In a given font, each character must be unique. Only strings containing characters that adhere to a String Reader font can be read.
- **String result.** A string, read from the target image, that respects all the settings and constraints of a string model.
- **Target image.** The image in which to read the strings.

Creating and customizing the fonts for a font-based context

Once you have allocated a String Reader font-based context, using **MstrAlloc()** and **M_FONT_BASED**, you must add at least one font to the font list of that context, and at least one character to the font. By default, the font list and the font are empty.

You can add characters to the font from one of the following:

- **System font.** These characters are referred to as system characters (for example, characters added from your system's Arial font).
- **Source image.** These characters are referred to as user-defined characters (for example, characters added from a mosaic of license plates).

Once a font has been added to the font list of the context, the font is automatically assigned default settings. Typically, these settings are sufficient for many applications. It is recommended that you try the defaults before changing them. However, if the default font settings do not suit your needs, you can:

- Set the font's character type.
- Normalize the font's characters to an appropriate size.
- Set the space size.
- Set the font's character baseline.
- Sort the font's characters.

You can check the font settings by drawing the font characters in an image buffer, using **MstrDraw()**.

Adding and deleting fonts to the context

To add a font to the font list in the String Reader context, use **MstrControl()** with **M_FONT_ADD**. The fonts that you add to the String Reader context are empty; that is, they do not contain any characters. You must therefore add them. To do so, see the *Adding and deleting characters to the font* subsection in the *Creating and customizing the fonts for a font based context* section in *Chapter 12: String Reader*.

When a font has been added to the context, it is assigned an index. Fonts are indexed using positive, consecutive integers starting at zero, and increasing by one. You can use the index to access a font. The maximum number of fonts that you can add to a String Reader context is 255.

To control a font setting, you must use **MstrControl()** with the **M_FONT_INDEX** macro set to the index of a specific font, or to all fonts (**M_ALL**).

You can also delete a font or all fonts from the font list. To do so, use **MstrControl()** with **M_FONT_DELETE** set to the index of a specific font, or to all fonts (**M_ALL**). Note that when a font is deleted, its index is reassigned; that is, index values greater than that of the removed font are reduced by one.

Fonts can also be given a user-defined numeric label, using **MstrControl()** with **M_FONT_USER_LABEL**. A user label can be used as a means of identifying your font, independently from its index in the String Reader context. However, user labels cannot be used as a direct replacement for the index; to retrieve the index of a font from the user label, use **MstrInquire()** with **M_FONT_INDEX_FROM_LABEL**. The user label, once converted to an index value, can be used with any String Reader 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 font, set the user label to **M_NO_LABEL**.

Adding and deleting characters to the font

Once you have added a font to the context, you must add characters to that font. To do so, use **MstrEditFont()** with **M_FONT_INDEX** and **M_CHAR_ADD**. You cannot add a character that is smaller than 6x6 pixels. Characters can either be added from your system or from a source image. For more information, see the *Character representation* subsection in the *Creating and customizing the fonts for a font based context* section in *Chapter 12: String Reader*. Note that all operations that apply to the characters of a specific font are done using **MstrEditFont()**.

After a character is added to a font, it can be accessed with its respective font's index and its own character value (ASCII or Unicode). These two values uniquely identify the character. For example, to access the character 'A' in font 0, you must use **MstrEditFont()** with the **FontIndex** parameter set to **M_FONT_INDEX**, and the **Param2** parameter set to 'A'.

Since characters are partly identified based on their own value, two characters in one font cannot have the same value. If you try to add a character with a value that already exists in a font, the existing character will be replaced by the newly added one, unless you use **MstrEditFont()** with **M_NO_OVERWRITE**. However, multiple fonts within the context can have characters with the same value, each of which is distinguished by its respective font index.

To delete a character in a font, use **MstrEditFont()** with **M_CHAR_DELETE**, specifying the font's index and the character's value. Instead of passing the character value, you can also pass a null-terminated string array of all the characters to delete, or **M_NULL**, which will delete all characters in the font.

When you add a character to a font, you can also decide its type. Its type determines whether String Reader treats the character as regular (**M_REGULAR**) or as punctuation (**M_PUNCTUATION**). Regular characters can typically be categorized as letters or numbers. A linear sequence of regular characters forms a string; that is, each regular character in a string must fall within the Y-size range of every character in that string. Punctuation characters can typically be categorized as characters that are not letters or numbers, such as the quotation ("). To be considered part of the string, a punctuation character must fall within the range of at least one regular character's Y-size. To have String Reader automatically establish the type of the characters, use **M_AUTO_COMPUTE**. With this setting, String Reader will decide whether the type should be **M_REGULAR** or

M_PUNCTUATION, based on the character's shape and numerical code. When you add characters to a font, their default character type is **M_AUTO_COMPUTE**. In the vast majority of cases, **M_AUTO_COMPUTE** will choose the correct character type.

Character representation

Characters in the font are referred to as either system characters or user-defined characters. This is based on how you specify their character representation when you add the characters to the font.

System characters

System characters are characters that have been added from a given system font, such as Arial. To add characters from a system font, use **MstrEditFont()** with **M_CHAR_ADD** and **M_SYSTEM_FONT**. You must also specify a null-terminated string of the characters to add from the system font, the name of the system font, and the size of the characters to add, in points (for example size 10 font). You can add the regular characters of the font (that is, 'A' to 'Z', 'a' to 'z', and '0' to '9'), as well as punctuation (such as the hyphen or the comma). The system font's name must be provided in standard U.S. English; for example, "Arial", "Arial Bold", "Times New Roman Italic", and "Courier New Bold Italic". You can find the full list of system font names with the MIL StringReader utility.

User-defined characters

User-defined characters are characters that have been defined from source images. When the required system font is not available, user-defined characters are the ideal solution, since all you need is an image of the characters in the required font. To add user-defined characters to the font, use **MstrEditFont()** with **M_CHAR_ADD** and **M_USER_DEFINED**. You must also specify the identifier of the characters' source image, and a null-terminated string with the list of characters to associate with the character representations in the image. Note that the size of the characters is automatically determined from the source image.

- ❖ It is recommended that you use an interactive graphic tool, such as the MIL StringReader utility or Matrox Inspector, to create new characters to add to an existing font.

Typically, the characters in your list represent one string that you want to read. However, you can specify multiple strings to read by adding a space between groups of characters. Generally, you can use multiple strings when one or more of the following situations in the source image are encountered: the characters are not all on the same line, the contrast is clearly different between characters, or there is clearly a horizontal space between groups of characters.

When adding user-defined characters, the character representations in the source image are taken from left to right and from top to bottom and are associated to the corresponding characters in your character list. String Reader is able to locate the characters in the source image because it assumes a Latin-based font. For example, if you want to define an 'A', String Reader will look through the source image from left to right and from top to bottom until the Latin-based character 'A' is located. If it is not located, an error is returned.

For the most part, String Reader's Latin-based font definition is very useful, since it can, for example, tell the difference between a real character and background noise. However, this type of automatic font definition makes it problematic to define non-Latin-based characters, like Chinese letters, or to define an 'I' as a 'W'.

You can, however, choose to add a single user-defined character that bypasses String Reader's automatic Latin-based font definition. To do so, you must combine **M_USER_DEFINED** with **M_SINGLE**. In this case, the entire source image is automatically defined as one character. For example, if your source image depicts the letter 'I', and the character in the character list is the letter 'W', then you have just added a 'W' that looks like an 'I'.

Although this is not mandatory, when adding user-defined characters it is best to provide a high quality source image that ideally contains only the characters to add to the font. A good source image is especially important for a single user-defined character, which is typically added because it is, in some respect, out of the ordinary. The image should also be binarized exactly as you want. Otherwise, it will be binarized by MII.

Each potential character in the source image must be entirely connected (except for the accentuated characters, like "è") and should not be merged with other characters or other image objects. Also, the characters you want to add from the source image must all be approximately the same size. Even if the operation

succeeds, it is recommended that you draw the characters of the font, using **MstrDraw()**, to ensure that every character is defined as expected. For more information, see the *Annotation* subsection in the *Retrieving results and annotation* section in *Chapter 12: String Reader*.

Locating the specified characters in the source image is typically very robust. For example, you do not need to specify the specific size or location of the characters. However, if you are experiencing problems, or if you want to speed up this process, you can create a child buffer around the region that contains the characters to add. For more information, see the *Manipulating and controlling certain data buffer areas* section in *Chapter 18: Specifying and managing your data buffers*.

Source image foreground

When defining characters from a source image and adding them to a font, you must be certain that the foreground is set correctly. By default, the foreground is black. You can, however, change it to white. To alter the foreground, you can combine **M_USER_DEFINED** with either **M_FOREGROUND_BLACK** or **M_FOREGROUND_WHITE**. This foreground setting only allows you to set the foreground of the character representations in the source image. This setting does not in any way affect the read operation, as fonts themselves do not have any foreground at all. To read characters in a target image, where the characters have a white foreground, you must explicitly set the read foreground control type of the string model to white; otherwise, you will not be able to read the characters. By default, the foreground used for the read operation is black. For more information, see the *Target image foreground* subsection in the *Adding and deleting string models to the context* section in *Chapter 12: String Reader*.

Normalize characters

Whether you are working with system fonts or user-defined fonts, it is not unusual for the characters to have different dimensions. In fact, fonts added from multiple system fonts, or multiple images, often create a rather chaotic group of character sizes; this can make altering some String Reader settings difficult, such as reading within a specific scale range. In this case, you should normalize the characters, with respect to a given X-size or Y-size. Since the other dimension is always scaled to maintain the same aspect ratio, this results in all the characters having the same width or height.

To normalize your characters, you must first decide on the appropriate dimension. This is typically done by obtaining the size of a character with an appropriate height or width, using **MstrInquire()** with either **M_CHAR_SIZE_X** or **M_CHAR_SIZE_Y**. You can then use **MstrEditFont()** with **M_CHAR_NORMALIZE** and either the X-size or Y-size information to normalize the characters in the font. You can normalize a specific character in the font or all characters. In the former case, pass a null-terminated string of the characters to normalize. To normalize all the characters, pass **M_NULL** instead. The reference X-size or Y-size must be greater than 8 pixels.

Space size

The font is partly responsible for establishing the space size required to delimit multiple strings. Specifically, the space size threshold that ultimately delimits strings depends on two settings: the space width (**MstrControl()** with **M_SPACE_WIDTH**) and the maximum number of consecutive spaces (**MstrControl()** with **M_SPACE_MAX_CONSECUTIVE**). The space width is a font setting, while the maximum number of consecutive spaces is a string model setting. By default, the space width is equal to the average X-size (width) of the characters in the font, and the maximum number of consecutive spaces is equal to 3. The space size is calculated using the following equation:

SpaceWidth x (*MaximumNumberOfConsecutiveSpaces*+1)-1.

For example, if the average width of the characters is 30, the default space size would be 119 (30 x (3 + 1) - 1). Therefore, two adjacent characters that are greater than 119 pixels apart are considered part of two separate strings. Note that the space size is relative to the scale of the string.

For more information, see the *Space* section later in this chapter.

Baseline

The font establishes the string's baseline, which is the imaginary horizontal line on which the majority of the characters in the string rest, as shown below:



Before you can read your string, you must customize your font such that the baseline of each character within the string (the imaginary horizontal line on which each character rests) is correctly aligned with the other characters in the font.

As soon as you add a font to the String Reader context, all the characters within the font are automatically assigned default baselines. When you define a character from a system font (**M_SYSTEM_FONT**, such as Arial or Times New Roman), String Reader will automatically compute appropriate baselines for all the characters. However, when you define a user-defined font (**M_USER_DEFINED**), there is no inherent baseline, since the font is defined from an image. Although String Reader will assign appropriate baselines for most of the characters in the font, there are instances where it cannot. For example, String Reader will not be able to assign a baseline to a hyphen (-), and will assign **M_NONE** as the baseline.

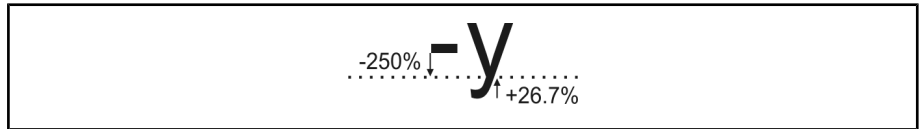
When choosing what baseline to give a character, keep in mind that the character's baseline is computed as the distance between the bottom of the character and the baseline of the string, and is measured as a percentage of the character's height. This distance can be a negative value (for characters that rest above the string's baseline, such as '-') or a positive value (for characters that rest below the string's baseline, such as 'p'). The bottom of each character is considered to be 0, the top of each character is considered to be 100, and the middle of each character is considered to be 50.

Typically, the baseline of the string intersects the bottom of your characters; therefore, a character's baseline is usually 0. All digits, for example, have a baseline of 0. Most uppercase characters, except possibly Q, have a baseline of 0. Similarly, the majority of lowercase characters, except those with descenders (g, j, p, q, and y), have a baseline of 0.

However, some characters have a non-0 baseline value. For example, the bottom of each character with a descender (g, j, p, q, y, and sometimes Q) typically falls below the baseline of the string. That is, the 0 value for characters with descenders is usually lower than the 0 value for most other characters in the string. Similarly, the 0 value for characters that rest above the baseline (such as the hyphen, "-") is usually higher than the 0 value for most other characters in the string. Notice the following string, where the baseline of every character is set to 0:

.....AbcPqRSTWX.Y.Z.....

You can see that not all the characters are aligned properly with the other characters. The characters that normally rest either below or above the baseline of the string need a baseline value that properly aligns them with the other characters in the string. As you can see, the correct baseline for the hyphen ('-') would be approximately -250% of its height, while the correct baseline for the 'y' would be approximately 26.7% of its height:



To set the baseline for both system and user-defined characters, use **MstrEditFont()** and pass the baseline value, as a percentage of the character's height. Any value between -1000 and 1000 is valid. Note that the baseline need not intersect the character.

To reset characters to their automatically computed baseline, use **MstrEditFont()** with **M_CHAR_BASELINE** and **M_AUTO_COMPUTE**, and specify a null-terminated string of the characters.

You can also decide to give a character no baseline, by setting the baseline to **M_NONE**. In this case, String Reader ignores the character's baseline value when reading strings. This is a useful setting because, when characters are being read, their baseline is one of the conditions that determines whether they are part of a string. For characters whose baseline is set to **M_NONE**, the baseline condition is ignored. The character must still respect the minimum definition of a string (a linear sequence of regular characters), and it must also respect all other possible restrictions that have been set for the string, such as the character's scale range.

Sorting characters

Characters are not always added to the font in an orderly fashion; particularly user-defined characters. It can therefore be difficult, when displaying these characters, to quickly tell which ones are missing. Also, you might want to have your characters ordered by sub-lists, such as all the digits, followed by all the letters, followed by all the symbols. To solve these problems, and any others regarding the order of your characters, String Reader provides a sort functionality.

To sort the characters in the font, according to their character values, in either ascending or descending order, use **MstrEditFont()** with **M_CHAR_SORT** and either **M_ASCENDING** or **M_DESCENDING**, and a null-terminated string of the characters.

Using a fontless context

Characters have distinct features which distinguish them from one another. The String Reader module can make use of these characteristics to read strings in a target image without making use of a font. To do so, it requires that you use a predefined fontless context. You might choose to use a fontless context over a font-based context in cases where you expect to read strings in images with variable fonts. A fontless context is faster to set up than a font-based one.

To use a fontless context, you need to restore, using **MstrRestore()**, one of the predefined context files located under the "*\\Matrox Imaging\\contexts*" MIL installation folder. The String Reader module comes with three predefined fontless contexts:

- *"FONTLESS_ANPR.msr"*. A generic context useful to read a wide variety of licence plate types written in Latin-based alphabets and Arabic numerals.
 - *"FONTLESS_EUROPEAN_ANPR.msr"*. A context useful to read European licence plates.
 - *"FONTLESS_MACHINE_PRINT.msr"*. A special context that reads machine printed characters in Arial, Ocr-B, or other sans-serif fonts.
- ❖ The String Reader module can only use a fontless context to read uppercase characters and numbers.

Enabling and disabling characters

When you restore a fontless context, it contains information about all uppercase characters and numbers. By default, the read operation searches the image for this entire range. However, for some applications, it is preferable to search for a subset of these characters, especially when you know one of two similar looking characters will not appear. In a fontless context, characters are not added or deleted; instead, they are enabled or disabled. You can disable characters using **MstrControl()** with **M_DISABLE_CHAR** or re-enable them with **M_ENABLE_CHAR**.

When only a few characters are needed, you can disable all characters (**M_DISABLE_CHAR** set to **M_ALL**) and then re-enable the required characters, rather than disabling characters one at a time.

Customizing a fontless context

After you have restored a fontless context, you need to specify the size of the characters to read. Use **MstrControl()** to specify the following values in pixels:

- A reference height (**M_REF_CHAR_SIZE_Y**).
- The width of the narrowest character (**M_MIN_CHAR_SIZE_X**) at the reference height.
- The width of the widest character (**M_MAX_CHAR_SIZE_X**) at the reference height.
- The thickness (stroke width) of a typical character (**M_REF_CHAR_THICKNESS**) at the reference height.

If the characters vary in size, set the reference height (**M_REF_CHAR_SIZE_Y**) to the average value of the character set's heights. Then, specify minimum and maximum allowable scale factors based on the reference height with **M_STRING_SCALE_MIN_FACTOR** and **M_STRING_SCALE_MAX_FACTOR**.

When setting **M_MIN_CHAR_SIZE_X**, **M_MAX_CHAR_SIZE_X** and **M_REF_CHAR_THICKNESS**, specify the width and thickness that the characters have when they have a height of **M_REF_CHAR_SIZE_Y**. When the read operation finds characters of a different height, size values are scaled by the ratio between the height found and the reference height.

In the following example, the characters have a reference height of 41 pixels. The widest character, Q, has a width of 20 pixels, the narrowest character, 1, has a width of 11 pixels, and the thickness of the characters is 6 pixels.



Adding and deleting string models to the context

Once you have added a font to the font list in the String Reader context, you must add at least one string model to the string model list in that context. By default, the string model list is empty.

Each string model in the string model list represents a template, a way of locating a string in the target image. The settings in this template (string model) have been configured with defaults that are typically sufficient for many applications. It is recommended that you try the defaults first. The following sections in this chapter describe the various string model settings and how to change them. For example, you can set the minimum and maximum number of characters in the string, the maximum number of strings to locate (for an individual string model and for the String Reader context), the acceptance level for the string, and the valid range of scale. You can also set various types of character constraints, which can be used with all fonts in the String Reader context, or can be restricted to a specific font.

After you have adjusted all your string model settings, you must preprocess the context before calling **MstrRead()**. Note that a string in the target image will only be read if it satisfies every string model criteria, and if it respects the minimum definition of a string (that is, a linear sequence of regular characters).

Adding and deleting the string models

To add a string model to the string model list in the String Reader context, use **MstrControl()** with **M_STRING_ADD**. The maximum number of string models that you can add to a String Reader context is 255. The read operation time can increase with the number of string models in the context.

When a string model has been added to the context, it is assigned an index. String models are indexed as positive, consecutive integers starting at zero, and increasing by one. You can use the index to access a string model.

To control a string model setting, you must use **MstrControl()** with the **M_STRING_INDEX** macro set to the index of a specific string model, or to all string models.

You can also delete a string model or all string models from the string model list. To do so, use **MstrControl()** with **M_STRING_DELETE** set to the index of a specific string model, or to all string models (**M_ALL**). Note that when a string model is deleted, its index is reassigned; that is, index values greater than that of the removed string model are reduced by one.

To return the index of the string model read, use **MstrGetResult()** with **M_STRING_MODEL_INDEX**.

String models can also be given a user-defined numeric label, using **MstrControl()** with **M_STRING_USER_LABEL**. A user label can be used as a means of identifying your string model, independently from its index in the String Reader context. However, user labels cannot be used as a direct replacement for the index; to retrieve the index of a string model from the user label, use **MstrInquire()** with **M_STRING_INDEX_FROM_LABEL**. The user label, once converted to an index value, can be used with any String Reader 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 string model, set the user label to **M_NO_LABEL**.

Preprocess and read

After you add string models and fonts to your String Reader context and before reading a string from the target image, the String Reader context must be preprocessed, otherwise an error will occur. It is during this preprocessing stage that String Reader prepares the context for the read operation by executing a number of preread calculations. To preprocess the String Reader context, use **MstrPreprocess()**. To read a string from the target image, use **MstrRead()**.

Note that you might have to preprocess the String Reader context again if you change some of its control settings, such as the nominal angle of the string (**M_STRING_ANGLE**). To check if you need to preprocess the String Reader context, use **MstrInquire()** with **M_PREPROCESSED**.

Saving and restoring

When you save a String Reader context, its preprocessing information is not saved. You must therefore preprocess the context when it is restored, even if you haven't changed any of its settings. To save a String Reader context to a file, use **MstrSave()**. To restore the context, use **MstrRestore()**.

Target image foreground

By default, strings are read assuming that the foreground is black. If it is white, or if it can be either white or black, you must specify this information in the string model, using **MstrControl()** with **M_FOREGROUND_VALUE**. Each string model can have a different foreground color. Note that only this string model setting affects what the read operation assumes is the foreground color; the foreground set when adding a user-defined font does not. For more information, see the *Source image foreground* subsection in the *Creating and customizing the fonts for a font based context* section in *Chapter 12: String Reader*.

Space

When using the MIL String Reader module, the space is not considered a normal character. For example, you can never read a space with String Reader, nor can you set a space as a constraint to which a valid character must adhere, to be read (for more information on character constraints, see the *Rules for character placement* section later in this chapter).

The space is used to delimit multiple strings in the target image when performing a read operation (**MstrRead()**). That is, the space size establishes the maximum distance between two adjacent characters before each character is considered part of two separate strings. The space size is calculated using the space width (a font setting) and the maximum number of consecutive spaces (a string model setting). For more information, see the *Space size* subsection in the *Creating and customizing the fonts for a font based context* section in *Chapter 12: String Reader*.

Note that, like the space, the carriage return also signifies a new string. If a character is on another line, it is automatically part of another string.

If the space was treated like a normal character, there would be tricky issues to contend with, such as distinguishing the space from noise. By defining a space as the marker that separates strings, there is less chance for an incorrect read operation. For example, consider an ANPR application that must read the following target image:



Since most license plates do not have any symbol between characters, and the ones that do cannot be ignored, multiple types of spaces might have to be considered for this string to be read correctly. You would not only have to account for the space that occurs between strings, but how this space might vary when noise exists (in this case, the flag). Essentially, it is simpler to decide whether or not the characters exist, rather than deciding whether or not a space exists.

When a space is encountered in the target image, it is marked by String Reader and can be artificially inserted to display formatted results. For more information, see the *Formatted and non-formatted results* subsection in the *Retrieving results and annotation* section in *Chapter 12: String Reader*.

Space width

To set the width of the space character in the font, use **MstrControl()** with **M_SPACE_WIDTH**. By default, the space width is set to **M_MEAN_CHAR_WIDTH**, which is equal to the average X-size (width) of the characters in the font. You can also set the space width to **M_INFINITE**, which is equal to an infinite amount of space. That is, all characters on the same line are always part of the same string. In this case, however, there will never be a space character in the formatted string since the distance can never be large enough to

constitute a space character. Other settings for the space width include the maximum X-size or minimum X-size character in the font, and a quarter of the maximum character width in the font. You can also set the space width to a specific value, in pixels.

The space width that you set is relative to the font. Therefore, the space width is considered to be at the size of the font (that is, the point size of the characters in the font), and adjusted according to the scale and aspect ratio of the string.

Maximum number of consecutive spaces

To set the maximum number of consecutive spaces required for two adjacent characters to be part of two separate strings, use **MstrControl()** with **M_SPACE_MAX_CONSECUTIVE**. The default value is 3 consecutive spaces.

Note that **M_SPACE_MAX_CONSECUTIVE** is a string model setting, while **M_SPACE_WIDTH** is a font setting. Therefore, each string model can have a maximum number of consecutive spaces, while all string models must adhere to their font's space width.

The notion of consecutive spaces is important, since non-consecutive spaces do not cause a split in the string. Consider the following Quebec license plate.



In this example, the first 3 characters are separated by a single space each, which makes them part of the same string. However, following the letter 'N', there are several consecutive spaces; if you set a maximum of one consecutive space, the '9' would be part of a different string. Based on the maximum number of spaces, this license plate can either be read as two strings, or one string with a large space between 'N' and '9'.

Number of strings to read

You can set the maximum number of strings to read in the target image for both the String Reader context and for each individual string model in the String Reader context.

For each individual string model, use **MstrControl()** with **M_STRING_NUMBER**, specifying the index of the particular string model. The default value is 1. To read all strings that adhere to a string model, set **M_STRING_NUMBER** to **M_ALL**.

The number set for the String Reader context specifies the maximum total of all strings that can be read, for all string models in the context together. To set the number for a String Reader context, use **M_STRING_NUMBER** with **M_STRING_NUMBER**, specifying **M_CONTEXT** as the index. The default value is **M_ALL**, which reads the maximum number of strings specified for each string model.

Note that if the context's **M_STRING_NUMBER** value is reached, the read operation stops, regardless of any individual string model's outstanding **M_STRING_NUMBER** setting. Similarly, once every individual string model's **M_STRING_NUMBER** value is reached, the read operation stops, regardless of the context's outstanding **M_STRING_NUMBER** setting.

For example, if you are writing an application that reads three name tags, you need three string models, one for each name. If you set **M_STRING_NUMBER** for the context to 1, and you set **M_STRING_NUMBER** for each string model to 1, then as soon as one of the name tags is read, both string number settings will be satisfied. However, if you set **M_STRING_NUMBER** for the context to 3 and you set **M_STRING_NUMBER** for each string model to 1, then the string number settings will only be satisfied when one of each name tag is read.

Degrees of freedom

By default, the String Reader module has been configured to conduct a fast and robust read operation, with a reasonable degree of tolerance for potential strings in the target image. However, to read unusually sized and positioned strings, you might have to change the tolerance defaults. Specifically, you might have to modify the default angle, scale, aspect ratio, baseline, and skew settings with which the strings are read. Increasing these tolerances can increase the time of the read operation.

Tolerance values are always relative to applicable angle, scale, aspect ratio, and baseline values that you have set. For example, the baseline's tolerance depends on the size of the character; therefore, if a character has been scaled, the tolerance is calculated at that scale.

String angle and character angle

The angle of the string is the angle of the best-fit line that falls on the baseline of all the characters in that string. For example, the angle of the following string is -10° :

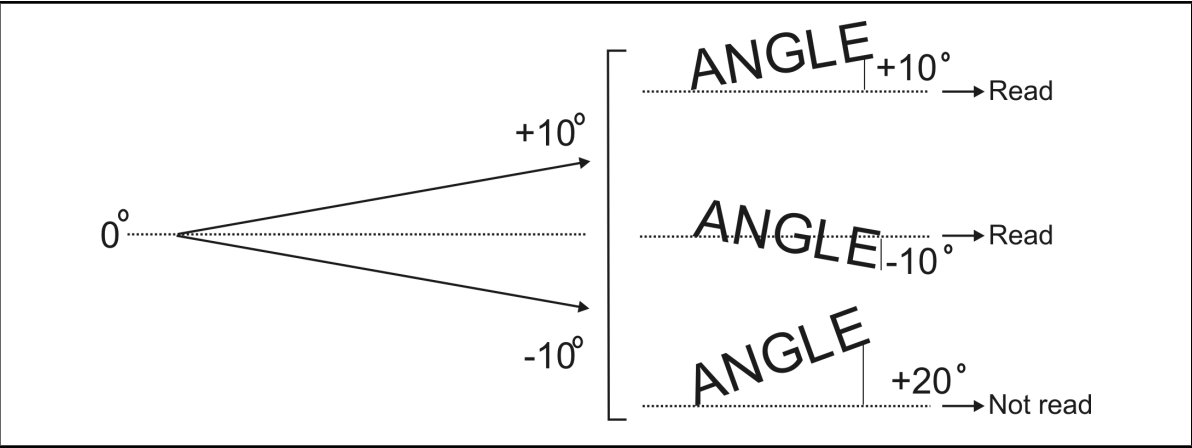


By default, strings are located in the target image at a nominal angle of 0.0° , and with an angular tolerance of $\pm 10.0^\circ$. This means that, only those strings in the target image that have an angle between $+10.0^\circ$ and -10.0° can actually be retrieved as results.

To change the angular tolerance, use **MstrControl()** with **M_STRING_ANGLE_DELTA_POS** and **M_STRING_ANGLE_DELTA_NEG**. Valid values are between 0.0° and 10.0° . These tolerance values are relative to the nominal angle of the string, which you can set using **MstrControl()** with **M_STRING_ANGLE**.

Essentially, `M_STRING_ANGLE_DELTA_POS` and `M_STRING_ANGLE_DELTA_NEG` set the possible upper (clockwise) and lower (counter-clockwise) limits of the string's angular range (relative to `M_STRING_ANGLE`). The upper limit can be described as `M_STRING_ANGLE + M_STRING_ANGLE_DELTA_POS`, while the lower limit can be described as `M_STRING_ANGLE - M_STRING_ANGLE_DELTA_NEG`. Strings with angles outside this angular range will not be returned as results.

For example, if you know that a string in the target image will typically be located at 0.0° , but you want to allow for an angular tolerance of $\pm 10^\circ$, you would set `M_STRING_ANGLE_DELTA_NEG` and `M_STRING_ANGLE_DELTA_POS` to 10° .



Each character within the string also has an angle. Typically, this angle value is the same as the string's angle value; however, this is not mandatory. The following example illustrates the difference between a string angle and a character angle; the characters in the string have an angle of 10° , while the angle of the string itself is 0° .



To retrieve the string angle, use **MstrGetResult()** with **M_STRING_ANGLE**. To retrieve the character angle of the string, use **MstrGetResult()** with **M_CHAR_ANGLE**.

Note that the String Reader algorithm is naturally robust to variations in a character's angle. However, if you expect your characters to be located between a wide angular range, it is recommended to enable calculations specific to angular-range search strategies. To do so, use **MstrControl()** with **M_SEARCH_CHAR_ANGLE**. Enabling this setting might increase the read operation's processing time.

String scale and character scale

The scale of the string is the median scale, in the X-direction, of each individual character in that string. By default, strings are located in the target image at a nominal scale of 1.0, with a maximum permitted scale of 2.0 and a minimum permitted scale of 0.5. This means that strings in the target image that have a scale between 0.5 and 2.0 can be returned as results.

By default, the scale of each individual character within the string can vary from the scale of the string by a maximum factor of 1.1 and a minimum factor of 0.9. This means that characters that have a scale between 1.1 and 0.9 (from the string scale) can be considered part of the string.

To alter the maximum (upper limit) and minimum (lower limit) permitted string and character scales, use **MstrControl()** with **M_STRING_SCALE_MAX_FACTOR**, **M_STRING_SCALE_MIN_FACTOR**, **M_CHAR_SCALE_MAX_FACTOR**, and **M_CHAR_SCALE_MIN_FACTOR**. Valid values for the maximum string and character scales are between 1.0 and 2.0. Valid values for the minimum string and character scales are between 0.5 and 1.0. String scales are relative to the nominal scale of the string, that is, **M_STRING_SCALE** x

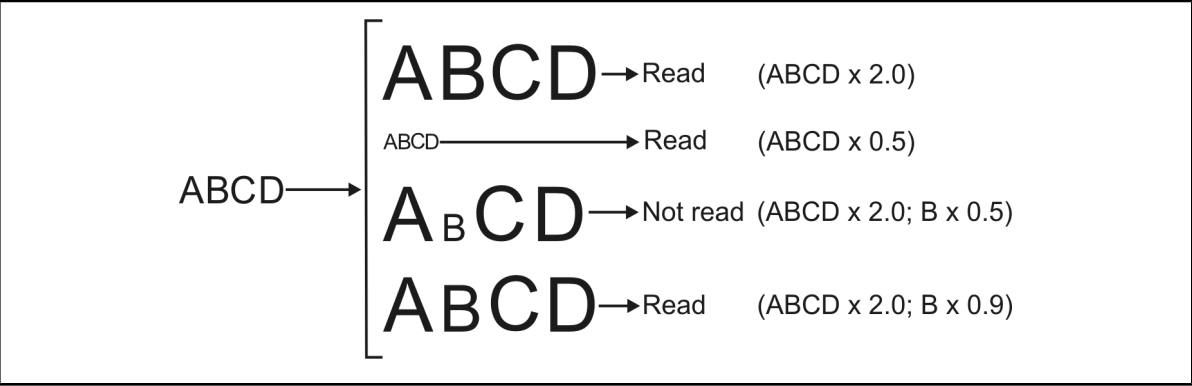
M_STRING_SCALE_MAX_FACTOR or **M_STRING_SCALE_MIN_FACTOR**.

You can set **M_STRING_SCALE** with **MstrControl()**. Valid string scale values are between 0.25 and 4.0. Character scales are relative to the scale of the string in the target image, as calculated by String Reader. That is, *CalculatedStringScale* x

M_CHAR_SCALE_MAX_FACTOR or **M_CHAR_SCALE_MIN_FACTOR**. Note

that **M_CHAR_SCALE_MAX_FACTOR** and **M_CHAR_SCALE_MIN_FACTOR** define how much an individual character's scale can deviate from the string's scale.

For example, you want to write an application that reads the string 'ABCD' within a scale range of 0.5 to 2.0. However, you want to ensure that the scale of each individual character in the string only ranges between 0.9 and 1.1. To do so, set **M_STRING_SCALE** to 1.0, **M_STRING_SCALE_MAX_FACTOR** and **M_STRING_SCALE_MIN_FACTOR** to 2.0 and 0.5, and **M_CHAR_SCALE_MAX_FACTOR** and **M_CHAR_SCALE_MIN_FACTOR** to 1.1 and 0.9.



To retrieve the string scale, use **MstrGetResult()** with **M_STRING_SCALE**. To retrieve the character scale, use **MstrGetResult()** with **M_CHAR_SCALE**. Strings and characters with scales outside their respective range will not be returned as results. Note that characters that are outside the string's upper and lower limits will not be read.

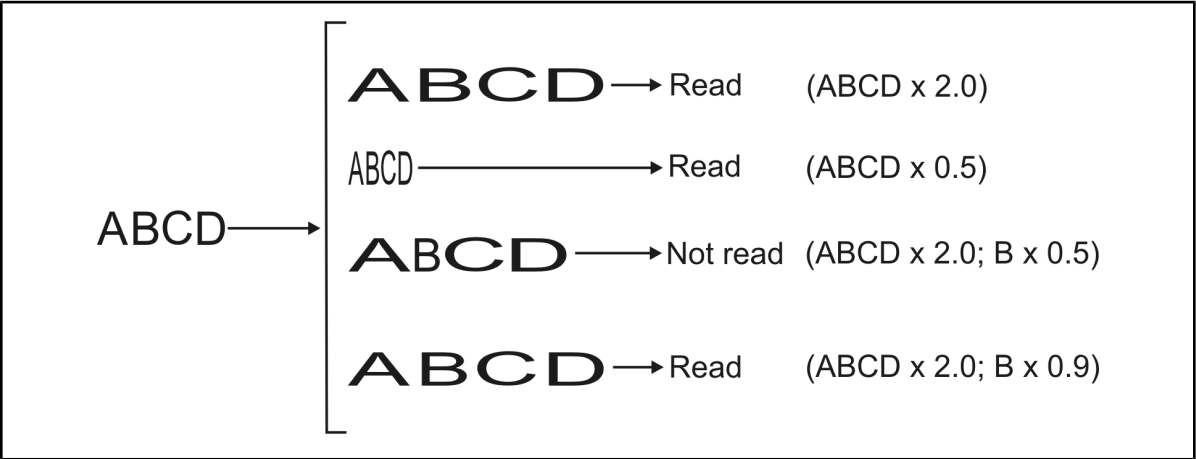
String aspect ratio and character aspect ratio

The aspect ratio of a string is the median aspect ratio of each individual character in that string. The aspect ratio of a character is the ratio of its scale in the X-direction by its scale in the Y-direction. By default, strings are located in the target image at a nominal aspect ratio of 1.0, with a maximum permitted aspect ratio of 1.25 and a minimum permitted aspect ratio of 0.8. This means that strings in the target image that have an aspect ratio between 1.25 and 0.8 can be returned as results.

By default, the aspect ratio of each individual character within the string can vary from the aspect ratio of the string by a maximum factor of 1.1, and a minimum factor of 0.9. This means that only characters that have an aspect ratio between 1.1 and 0.9 (from the string aspect ratio) can be considered part of the string.

To alter the maximum (upper limit) and minimum (lower limit) permitted string and character aspect ratios, use **MstrControl()** with **M_STRING_ASPECT_RATIO_MAX_FACTOR**, **M_STRING_ASPECT_RATIO_MIN_FACTOR**, **M_CHAR_ASPECT_RATIO_MAX_FACTOR**, and **M_CHAR_ASPECT_RATIO_MIN_FACTOR**. Valid values for the maximum string and character aspect ratios are between 1.0 and 2.0. Valid values for the minimum string and character aspect ratios are between 0.5 and 1.0. String aspect ratios are relative to the nominal aspect ratio of the string, that is, **M_STRING_ASPECT_RATIO** x **M_STRING_ASPECT_RATIO_MAX_FACTOR** or **M_STRING_ASPECT_RATIO_MIN_FACTOR**. You can set **M_STRING_ASPECT_RATIO** with **MstrControl()**. Valid string aspect ratios are between 0.5 to 2.0. Character aspect ratios are relative to the aspect ratio of the string in the target image, as calculated by String Reader. That is, *CalculatedStringAspectRatio* x **M_CHAR_ASPECT_RATIO_MAX_FACTOR** or **M_CHAR_ASPECT_RATIO_MIN_FACTOR**. Note that **M_CHAR_ASPECT_RATIO_MAX_FACTOR** and **M_CHAR_ASPECT_RATIO_MIN_FACTOR** define how much an individual character's aspect ratio can deviate from the string's aspect ratio.

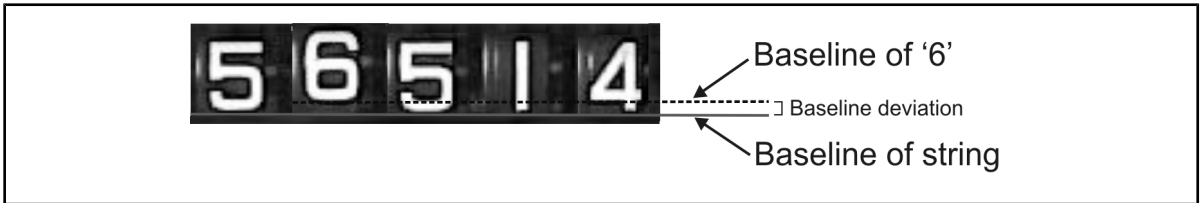
For example, you want to write an application that reads the string 'ABCD' within an aspect ratio range of 0.5 to 2.0. However, you want to ensure that the aspect ratio of each individual character in the string only ranges between 0.9 and 1.1. To do so, set **M_STRING_ASPECT_RATIO** to 1.0, **M_STRING_ASPECT_RATIO_MAX_FACTOR** and **M_STRING_ASPECT_RATIO_MIN_FACTOR** to 2.0 and 0.5, and **M_CHAR_ASPECT_RATIO_MAX_FACTOR** and **M_CHAR_ASPECT_RATIO_MIN_FACTOR** to 1.1 and 0.9.



To retrieve the string aspect ratio, use **MstrGetResult()** with **M_STRING_ASPECT_RATIO**. To retrieve the character aspect ratio, use **MstrGetResult()** with **M_CHAR_ASPECT_RATIO**. Strings and characters with aspect ratios outside their respective range will not be returned as results. Note that characters that are outside the string's upper and lower limits will not be read.

Character's maximum baseline deviation

In a target image, most characters in a string do not rest on significantly different baselines. However, you might encounter a target image where certain characters are significantly misaligned. In some cases, these strings are valid and you want to be able to read them. For example, when dealing with odometers, certain digits might be considerably misaligned with the other digits, yet the string is still valid, such as in the following image:



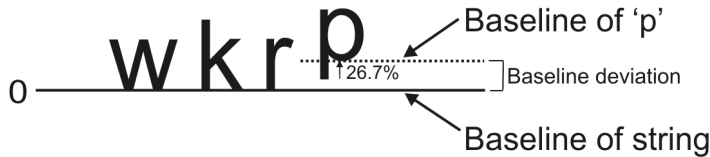
The image above shows that '6' has a baseline deviation. A character's baseline deviation is the difference between the character's baseline and the baseline of the string in which it is found. A character's maximum baseline deviation is the maximum tolerable baseline deviation that a character in a string can have before it is rejected from the string. In most cases, a character's baseline deviation is calculated as a percentage of the character's height. For the example above, a baseline deviation of 15 for '6' means that 6's baseline deviates from the string's baseline by 15% of 6's height. However, for punctuation characters, the baseline deviation is calculated as a percentage of the height of the character with the largest Y-size in the font.

By default, all characters are assigned a default maximum baseline deviation of 10. This value is a sufficient tolerance for most applications because characters in a string in a target image do not usually rest on significantly different baselines. However, in certain situations such as the odometer case, the default value might not be sufficient for the String Reader to accept the misaligned character. If the misaligned character's baseline deviation is more than the character's set maximum baseline deviation, the character will be rejected from the string. This is why in some cases, you must manually change the characters' maximum baseline deviation.

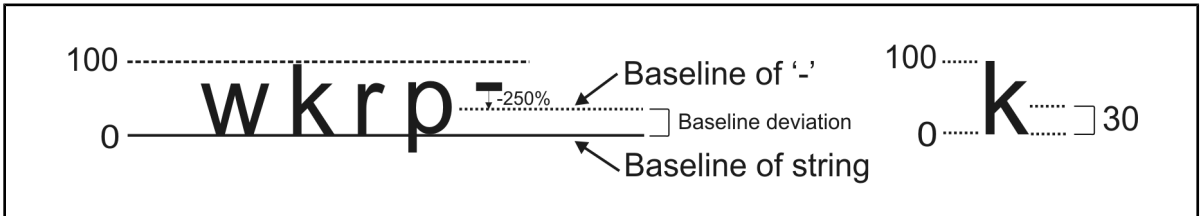
To set the maximum baseline deviation that all characters in the string can have, use `MstrControl()` with `M_CHAR_MAX_BASELINE_DEVIATION`. Valid values are between 0 and 100. The default value is 10.

To retrieve every read character's baseline deviation, use **MstrGetResult()** with **M_CHAR_BASELINE_DEVIATION**.

Consider now the target image shown below. Let us assume that in the font, 'p' had its baseline defined as 26.7%, which is illustrated by the dotted line (for more information on baselines, see the *Baseline* subsection in the *Creating and customizing the fonts for a font based context* section in *Chapter 12: String Reader*). We can see that in the target image, 'p' is misaligned with the other characters, and therefore there is a baseline deviation. If this baseline deviation is larger than the set maximum baseline deviation, the character will be rejected from the string. If this is the case and you want the String Reader to accept 'p', you need to increase the characters' maximum baseline deviation.



As mentioned previously, the baseline deviation for punctuation characters is represented as a percentage of the height of the character with the largest Y-size in the font. This is used because the Y-size of punctuation characters is typically too small to calculate baseline deviation as a percentage of the character's height. The image below shows the string, 'wkrp-'. Suppose the hyphen's baseline was defined in the font as -250%. The target image shown has the hyphen's baseline misaligned from the other characters' baseline; therefore, there is a baseline deviation. Let us assume that 'k' is the character with the largest Y-size in the font. To calculate the baseline deviation, instead of using the height of the hyphen, String Reader uses 'k' to measure the baseline deviation. From the right side of the image below, you can see that the baseline deviation of the hyphen corresponds to 30% of k's height. If 30 is larger than the set maximum baseline deviation, the hyphen will be rejected from the string. In this case, to accept '-', you need to increase the characters' maximum baseline deviation.



Note that a character's baseline deviation is represented at the scale of the target image. With non-punctuation characters, the baseline deviation is represented as a percentage of the character's height at the scale of the target image. With punctuation characters, the baseline deviation is represented as a percentage of the character with the greatest Y-size within the font, but at the scale of the string in the target image.

Skew angle

The term skew can be defined as a rotational deviation from the correct horizontal and vertical orientation. Typically, when dealing with characters, a skew presents itself as a lateral or horizontal lean.



The String Reader algorithm is naturally robust to variations in a character's skew angle. However, if you expect your characters to be skewed with a wide angular range (for example, when dealing with significant shifts in perspective), it is recommended to enable calculations specific to angular-range skew search strategies. To do so, use **MstrControl()** with **M_SEARCH_SKEW_ANGLE**. Note that enabling this setting might increase the read operation's processing time.

The skew angle for each character within a string must always be the same. To retrieve the character's skew angle, use **MstrGetResult()** with **M_SKEW_ANGLE**.

Rules for character placement

The read operation (**MstrRead()**) compares each character in the target image to each character in the font and locates the best string that satisfies all your constraints. You can place constraints on the size of the string, and on the characters themselves. Used together, these constraints form the grammar rules. By setting up proper grammar rules, you can restrict the character comparison process, thereby increasing the robustness of the read operation, and lowering the possibility of false positive results.

Every grammar rule that you set for a string model is referred to as the string model's grammar, while every grammar rule that you set for all string models in the String Reader context is referred to as the context's grammar. Note that you do not explicitly set grammar rules for the context. The context's grammar is formed by all the grammar rules for all the string models within that context.

- ❖ The space character cannot be set as a constraint and is not counted in the string size. For more information, see the *Space* section earlier in this chapter.

String size

String Reader will read the number of characters specified by the string size. For best results, the number of characters to be read in your target image should match your string size.

You can set the maximum and minimum number of characters a string must have for it to be read using **M_STRING_SIZE_MAX** and **M_STRING_SIZE_MIN**. You can set these to any value greater than or equal to 1. The maximum string size can also be set to infinite (**M_INFINITE**).

Character constraints

To ensure that only a certain type of character (or a particular character) appears at a specific position in the string, you can apply character constraints, using **MstrSetConstraint()**. You can specify that a character at a specific position falls into one of the following types of characters:

- A digit (**M_DIGIT**): '0' to '9'.
- A letter (**M_LETTER**): 'a' to 'z' and 'A' to 'Z'.

- An uppercase letter (**M_LETTER** + **M_UPPERCASE**): 'A' to 'Z'.
- A lowercase letter (**M_LETTER** + **M_LOWERCASE**): 'a' to 'z'.
- A combination of the above types (for example, **M_DIGIT** + **M_LETTER**).
- Any character (**M_ANY**).

For example, if you set the character constraint for the character at position 1 to **M_DIGIT**, then only if the character at position 1 is between '0' and '9' can it be considered part of the string.

You can further restrict the character constraint to a specific list of characters of the specified type. This list can include special characters and punctuations (for example, 'a', '1', 'aA1', '&', '-', '...'). For example, if you set the character constraint for the character at position 1 to '1', then only if the character at position 1 is '1' can it be considered part of the string.

When providing a specific list of characters, you must ensure that the characters are of the correct constraint type. For example, if the specific character in the list is '1', and the constraint type is **M_LETTER**, an error will be generated. Also, you cannot repeat letters in your character list; that is, each letter must be unique. For example, 'AABB' is not allowed.

Note that a constraint can be set for any position, even when the string to read does not contain a character at that position. For example, you can set a constraint for a position that is greater than the number of characters in the string. Such constraints are always ignored.

How constraints are used with string models and fonts

The constraint can either be applied to a specific string model in the context, or to all string models in the context. To apply the constraint to a specific string model, set **M_STRING_INDEX** to the index value of the string model. To apply the constraint to all string models, set **M_STRING_INDEX** to **M_ALL**.

If it is appropriate, you can set a default (**M_DEFAULT**) constraint for the string model. This constraint will be used for any character in the string for which a constraint has not been explicitly set. This is useful if the majority of the characters must respect the same constraint. For example, you can have an application that

reads 26 characters, all of which are digits, except for the character at position three, which is a letter. Rather than having 26 lines of code with almost identical **MstrSetConstraint()** calls (one for each character position), you can simplify your code by:

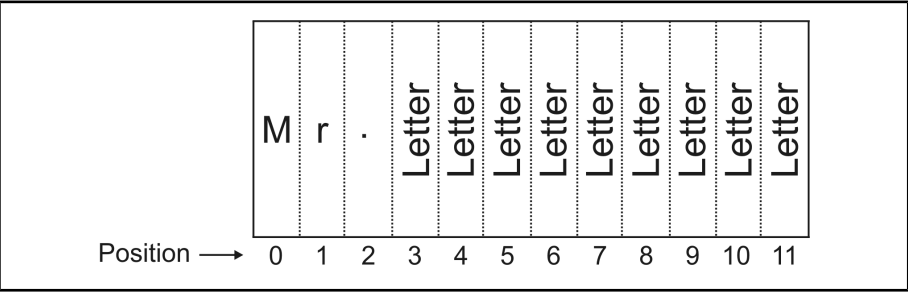
- Setting the default constraint for the string model to **M_DIGIT**.
- Setting the constraint for the character at position three to **M_LETTER**.

The constraint is automatically used with any font in the context. To restrict the constraint to a specific font, set **M_FONT_INDEX** to the index value of a specific font, and add it to the constraint (for example, **M_DIGIT + M_FONT_INDEX**). This allows you to, for example, write an ANPR application that reads old and new license plates that use different fonts. Your application can specify an "OldPlates" string model with *Font1* and a "NewPlates" string model with *Font2*.

Note that the specified font index can be for a font that does not currently exist. However, if the font still does not exist at preprocessing time, an error will be generated. At preprocessing time, each character in an explicit character list constraint must exist in at least one of the fonts, otherwise an error will be generated.

Grammar rules

Grammar rules refer to both character constraints and string size constraints, which are typically used together in concert. For example, you have an application that reads name tags. Your grammar rules specify that the minimum string size is set to 6 and the maximum string size is set to 12. Your grammar rules also specify the following character constraints for each of your string's 12 positions:



In this case, the following table lists some strings that would be either accepted or rejected for the read operation:

String	Read state
Mr. White	Accepted
Mr. Orange	Accepted
Mr. Blonde	Accepted
Mr. Pink	Accepted
Joe Cabot	Rejected
Mr. Lawrence Bender	Rejected
Mr. X	Rejected

As previously mentioned, each string model in a String Reader context can have a different set of grammar rules. This gives the String Reader module a lot of versatility; it allows you to read many different types of stings within a single context. For example, you want to read two types of telephone numbers, one that starts with '514', and another that starts with '450'. You can therefore create two string models, one for each area code. In this case, the grammar rules for both string models would be exactly the same, except for the constraint placed on the first three characters:

String(0)→	5	1	4	-	Digit	Digit	Digit	-	Digit	Digit	Digit	Digit
String(1)→	4	5	0	-	Digit	Digit	Digit	-	Digit	Digit	Digit	Digit
Position→	0	1	2	3	4	5	6	7	8	9	10	11

The advantage of having grammar rules can also be seen when writing an ANPR application. For example, the majority of Quebec license plates have two types of strings: three letters followed by three digits, or three digits followed by three letters.



If you want to write an application that reads these license plates, you would need two string models. Each would have a maximum and minimum size of 6. One model would read letter, letter, letter, digit, digit, and another model would read digit, digit, digit, letter, letter, letter.

Note that, if you developed such an application using the OCR module instead of the String Reader module, it would be more complex to write, and less able to reject incorrect strings. Since the OCR module does not allow for grammar rules, the state of all characters cannot be inferred by the state of some. For example, you know that for Quebec license plates, strings that start with a digit must have three digits followed by three letters. Similarly, strings that start with a letter must have three letters followed by three digits. While String Reader can make this assumption, OCR cannot. With OCR, you would have to specify, for each character position, digit or letter. Although this would read the license plates, it might also read some strings that are not license plates (false positives). That is, if the character at position 0 is a letter, it is simply not true that the character at position 1 can be either a letter or a digit; it must be a letter. Having the ability to apply grammar rules helps you avoid such inaccuracies.

Global context settings

The global settings of a String Reader context can determine the strategy used to read all of the context's string models, which in turn, could affect the speed and robustness of the read operation (**MstrRead()**). To adjust the global context settings to fit your individual application's needs, use **MstrControl()** with **M_CONTEXT**.

The default value for each global context setting is typically good. However, if you want to find the optimum balance between speed and robustness for a particular application, it is recommended that you experiment with different global context setting values.

The global context settings include:

- Minimum contrast.
- Speed.
- Timeout.
- Maximum number of strings to read.
- Encoding.
- Space character.
- String separator.
- Scale.
- Skew.

Unlike other global context settings of StringReader, you can set the maximum number of strings to read for both the String Reader context and each string model within the context. For more information, see the *Number of strings to read* section earlier in this chapter.

Scale and skew have been previously discussed. For more information, see the *String scale and character scale* subsection in the *Degrees of freedom* section in *Chapter 12: String Reader* and the *Skew angle* subsection in the *Degrees of freedom* section in *Chapter 12: String Reader*. Space characters and string separators are discussed in the *Formatted and non-formatted results* subsection in the *Retrieving results and annotation* section in *Chapter 12: String Reader*.

Minimum contrast

String Reader is contrast invariant. Therefore, if a valid character exists in the target image, it will be read, regardless of whether the contrast between that character and its background is high or low. However, you might be able to speed up the read time by specifying the minimum contrast value, using **MstrControl()** with **M_MINIMUM_CONTRAST**. Valid values are between 1 and 255. The default value is 15. Any character in the target image that does not have a contrast greater than the specified minimum value will be ignored, thereby possibly increasing the speed of the read operation.

Increasing the minimum contrast might make a readable character unreadable. For example, if the contrast between a valid character and its background is 25, and you set the minimum contrast to 50, that character will not be read. It is therefore important to make sure that the minimum contrast value is not causing the read operation to ignore acceptable characters.

Speed

You can specify the algorithm's read speed, using **MstrControl()** with **M_SPEED**. The speed can be set to **M_MEDIUM** or **M_HIGH**. The default is **M_MEDIUM**.

Higher read speeds cause the read operation to take a greater number of shortcuts, which typically results in shorter read times, though it can also skip important information.

Timeout

In time critical applications, you can set a time limit, in msec, for String Reader to read using the specified string models. To do so, use **MstrControl()** with **M_TIMEOUT**. The default value is 2000 msec. Due to certain application-dependent calculations, the actual maximum read time might vary slightly from the timeout specified. You can disable the timeout setting using **M_DISABLE**.

After the time limit has been reached, the read operation will terminate, even if the required number of strings has not been read. Results are returned for all strings read up to the timeout. It is not possible to predict which strings will be read beforehand.

To check whether the timeout limit has been reached, use **MstrGetResult()** with **M_TIMEOUT_END**.

Encoding

You can set the type of character encoding used by the String Reader context. To do so, use **MstrControl()** with **M_ENCODING**.

The selected encoding scheme will affect the expected data type of the information that you pass to or retrieve from String Reader. For example, when adding characters to the font, using **MstrEditFont()** with **M_CHAR_ADD**, the expected data type is based on the **M_ENCODING** setting.

The encoding schemes supported are **M_ASCII** and **M_UNICODE**. **M_ASCII** specifies an 8-bit ASCII standard character type, and corresponds to the *char* data type. This is the default value. **M_UNICODE** specifies a 16-bit Unicode standard character type, and corresponds to the *short* data type.

Scores and acceptances

When String Reader performs a read operation, it analyzes the target image and localizes potential character candidates for each string model. The best candidates that meet the minimum definition of a string (a linear sequence of regular characters) and all user-specified constraints will be read.

Every character candidate, and the string candidate to which it belongs, is given a score between 0 and 100. The higher the score, the better the candidate. These scores quantify:

- The similarity between the character in the target image and the corresponding character in the font.
- The similarity between the character in the target image and the other characters of the string in the target.
- The number of unread characters within the read-region of the string in the target image.

String Reader allows you to set various acceptance values, which act as a threshold for the candidate's scores. If the candidate's score is equal to or above its respective acceptance value, the candidate can be read; otherwise, it cannot be read. Note that the string candidates with the highest scores above all the acceptance values will be read, up to the maximum number of strings specified. For more information on setting the maximum number of strings to read, see the *Number of strings to read* section earlier in this chapter.

String Reader also allows you to set various string certainty values. Any string candidate with scores greater than or equal to the certainty levels is considered a certain match. Therefore, if the read operation localizes the required number of string candidates that meet all the certainty values, the read operation stops, without verifying the rest of the image for candidates with higher scores.

The acceptance level represents a threshold for a score that is adequate, though you would like to check the rest of the image for better scores. The certainty level represents a threshold for a score that is so good, you need not verify that there is a better one. Since lowering the certainty usually results in less image analysis, it can speed up the read operation. However, you should set the certainty carefully, since a high certainty can stop **MstrRead()** prematurely, and allow the best candidates to remain unread.

Acceptance and certainty values can be set for the characters, the string, and the read-region of the string in the target image. Characters also have a similarity and homogeneity setting.

The default value for each of these settings is typically sufficient. However, to fit the specific needs of your application, it is recommended that you try experimenting with the values. For example, trying various acceptance values might be necessary if you have an image with an unusual amount of text that you want to ignore. Note that, although each value represents a kind of threshold for a specific score, when used in concert, they provide a great deal of versatility for accepting and rejecting strings as a whole.

String acceptance, certainty, and score

The string score is the average score of all the characters in the string. To set the acceptance threshold for the string score, use **MstrControl()** with **M_STRING_ACCEPTANCE**. To set the certainty threshold for the string score, use **MstrControl()** with **M_STRING_CERTAINTY**. To get the actual score of the string, use **MstrGetResult()** with **M_STRING_SCORE**.

The following ANPR example illustrates an image used to define a font, and a target image where the string 'ZEY 877' must be read. In this example, the string has a score of 91.9%. Therefore, if **M_STRING_ACCEPTANCE** is set to 91.9 or below, the string 'ZEY 877' will be accepted by the read operation.



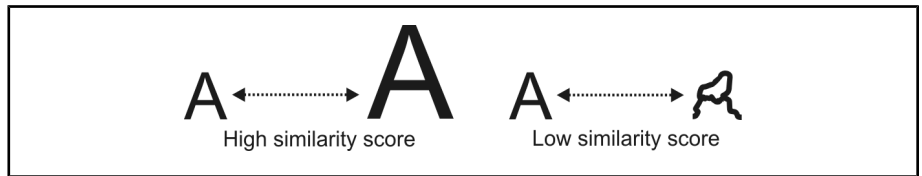
Character acceptance and score

The character score is calculated as the average value of the character's similarity and homogeneity scores. To set the acceptance threshold for the character score, use `MstrControl()` with `M_CHAR_ACCEPTANCE`. To get the actual score of the character, use `MstrGetResult()` with `M_CHAR_SCORE`.

Note that characters with character scores below the character acceptance value will not be considered part of the string, and therefore might cause the string not to be read. Note that if a character does not meet either the character similarity or homogeneity acceptance thresholds, these scores are ignored when calculating the string score (`M_STRING_SCORE`).

Character similarity

The character's similarity score quantifies how closely the character in the target image resembles the character in the font. This resemblance is based solely on how the character "looks," disregarding variations in contrast, size, aspect ratio, scale, and baseline. For example, an 'A' is still an 'A', even if the 'A' in the target image is twice as big as the 'A' in the font.



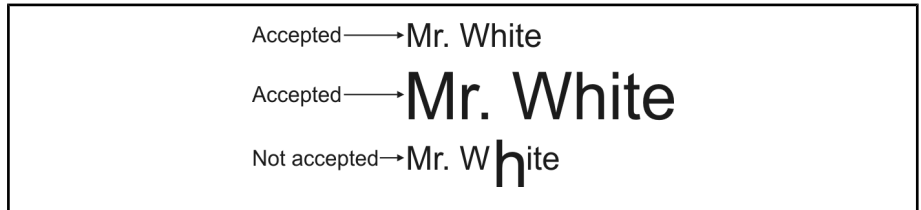
To set the acceptance threshold for the character's similarity score, use **MstrControl()** with **M_CHAR_SIMILARITY_ACCEPTANCE**. To get the actual similarity score of the character, use **MstrGetResult()** with **M_CHAR_SIMILARITY_SCORE**.

Character homogeneity

The character's homogeneity score quantifies the similarity between the character in the target image and the other characters of the string in the target image. To calculate the character's homogeneity score, many character traits are taken into account, such as variations in contrast, size, aspect ratio, scale, and baseline. If each of these characteristics (and others not mentioned) is the same between the character and all other characters of the string in the target image, the resulting homogeneity score for that character would be 100. Note, however, that this is seldom the case.

To set the acceptance threshold for the character's homogeneity score, use **MstrControl()** with **M_CHAR_HOMOGENEITY_ACCEPTANCE**. To get the actual homogeneity score of the character, use **MstrGetResult()** with **M_CHAR_HOMOGENEITY_SCORE**.

Since the character's homogeneity is calculated by how well the character compares to the other characters in the string, rather than how well it compares to the font, it can solve problems that cannot be resolved solely by similarity with the font. For example, in a target image, you might have characters that have high similarity scores and that, individually, are all acceptable. However, if their characteristics cause them to be out of context with the other characters in the string, you might want to reject them, based on their homogeneity score.



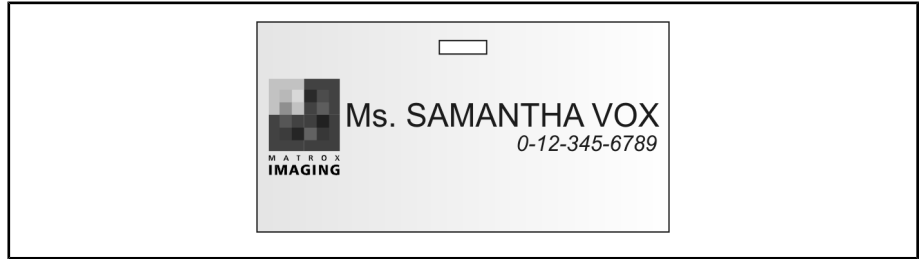
String target acceptance, certainty, and score

String Reader computes a score for the read-region of the string in the target image. This region can be defined as the string's box, which includes the maximum number of spaces before and after the string. This distance is calculated using the space width (**MstrControl()** with **M_SPACE_WIDTH**) and the maximum number of consecutive spaces (**MstrControl()** with **M_SPACE_MAX_CONSECUTIVE**). For more information, see the *Space size* subsection in the *Creating and customizing the fonts for a font based context* section in *Chapter 12: String Reader*.

Any unread character within the read-region in the target image will lower the string target score. An unread is any character that would ordinarily be read, if not for user-specified constraints (such as scale and grammar restrictions). The greater the number of unread readable characters in the read-region of the string in the target image, the lower the string target score will be.

To get the string target score, use **MstrGetResult()** with **M_STRING_TARGET_SCORE**. To set the acceptance threshold for the string target score, use **MstrControl()** with **M_STRING_TARGET_ACCEPTANCE**. To set the certainty threshold for the string target score, use **MstrControl()** with **M_STRING_TARGET_CERTAINTY**.

For example, to read the string "SAM VOX", you have set 'S', 'A', 'M' as the first three character constraints, and 'V', 'O', 'X' as the last three character constraints. In the following target image, it is likely that "SAM VOX" would be read.



However, you might want the extra letters at the beginning and in the middle of the read-region of the string to cause this string not to be read. To do so, set an adequate acceptance level for the string target score (for example, 50.0). The extra letters in the string's read-region will cause this string to have a very low string target score.

Retrieving results and annotation

After adding at least one string model and one font to the String Reader context, and performing a successful preprocess and read operation, you would typically retrieve and draw various features of your results. To do so, use **MstrGetResult()** and **MstrDraw()**. These functions offer many different types of results and draw operations. For example, you can retrieve the aspect ratio of the characters, using **M_CHAR_ASPECT_RATIO**, or draw a cross at the center of gravity of each string's character, using **M_DRAW_STRING_CHAR_POSITION**.

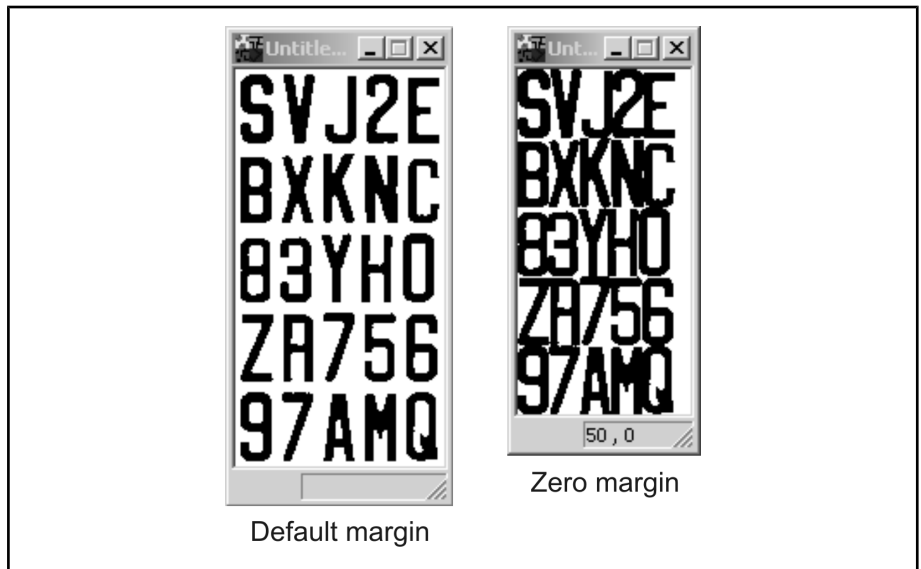
To return the index of the font of each individual character within the string, use **MstrGetResult()** with **M_CHAR_FONT**. To return the index of the character in the font for each individual character within the string, use **MstrGetResult()** with **M_CHAR_INDEX**. In either case, the index returned is the index value at the moment the **MstrRead()** operation is performed. Therefore, if the font list changes after the read operation has been called (for example, you add or remove a character), the index value that you retrieve might not be valid.

Annotation

The String Reader module allows you to draw specific features of the String Reader context or String Reader results, using **MstrDraw()**. For example, you can draw your string results to ensure that the read operation has read the correct characters.

After each drawing operation, the optimal image size that was needed for that operation is updated. This information can be inquired using **MstrInquire()** with **M_DRAW_LAST_SIZE_X** and **M_DRAW_LAST_SIZE_Y**. You can also update these size constants without actually performing the drawing operation; to do so, set the destination image of **MstrDraw()** to **M_NULL**. This can be useful if you, for example, want to get the optimal image buffer size for the drawing operation.

All characters are drawn according to their draw box margin, which you can set using **MstrControl()** with **M_DRAW_BOX_MARGIN_X** and **M_DRAW_BOX_MARGIN_Y**. These margins determine the amount of white space between each character that is drawn. The default value is 3. Note that a character has no inherent margin; that is, if these margin values are set to 0, the characters are drawn, one after the other and one above the other, with no white space.



For a String Reader context, you can draw a character representation of the font in the destination image. This is useful to ensure that the characters added to the font are correct. To do so, you must specify the explicit list of characters to draw and verify. Characters are drawn from left to right, and from top to bottom. Characters that fall outside the destination image are clipped. To draw the characters on multiple lines, a space character must be inserted to separate the strings included in the null-terminated string.

It might be useful to sort the characters before you draw them, thereby making it easier to visually ascertain if all the required characters are there. For more information, see the *Sorting characters* subsection in the *Creating and customizing the fonts for a font based context* section in *Chapter 12: String Reader*.

String Reader offers numerous drawing operations, which, if required, can be combined to draw multiple features simultaneously. For example, you can draw the characters of the string read, a box around the string read, and a cross at the center of the bounding box of each string's character (the character position). The following image shows how drawing the position of characters of a string (**M_DRAW_STRING_CHAR_POSITION**) for a typical Quebec license plate is displayed:



For a full list of drawing operations, see **MstrDraw()**. Note that characters are always drawn at the location read in the target with the correct angle, scale and aspect ratio.

Transformation coefficients

In addition to many other kinds of results, String Reader provides you with forward and reverse transformation coefficients. These results allow you to transform any point from the character's source image (the image from which the characters were defined) to its corresponding position in the target image (forward), or from the read position in the target image to its corresponding position in the character's source image (reverse). With this information, the equivalent position of any point of any character can be drawn.

For more information, see the transformation coefficients table in **MstrGetResult()**.

Formatted and non-formatted results

Unlike String Reader, humans have a hard time processing a long string of characters uninterrupted by any space, or any indication of where one string is separated from another. To make strings easier to read, you can display formatted results.

To retrieve the string as a formatted result, use **MstrGetResult()** with **M_FORMATTED_STRING**. A space character is inserted in the string each time the distance between two adjacent characters exceeds the space width (**MstrControl()** with **M_SPACE_WIDTH**), at the scale of the string.

You can also retrieve the entire text read as a formatted result, using **MstrGetResult()** with **M_TEXT**. In addition to inserting space characters, string separators are also inserted in the text each time the distance between two adjacent characters exceeds the space size, which depends on the space width (at the scale of the string) and the maximum number of consecutive spaces (**MstrControl()** with **M_SPACE_MAX_CONSECUTIVE**). For more information, see the *Space size* subsection in the *Creating and customizing the fonts for a font based context* section in *Chapter 12: String Reader*. Both the space character and the string separator must be set before the string is read.

To retrieve the non-formatted string (no spaces or string separators added), use **MstrGetResult()** with **M_STRING**.

Regardless of which result is being returned, strings are always ordered in the natural Latin-based reading order. This is the order that you read text written in a Latin-based language, such as Canadian English.

As mentioned earlier in this chapter, the space is not considered a character, and is not a part of any string that has been read. However, String Reader does mark the existence of a space, and can therefore artificially insert a space and/or a string separator at the correct place. For more information, see the *Space* section earlier in this chapter.

The default character used as the space character is ASCII 32, which is a space. You can, however, set a different character, using **MstrControl()** with **M_SPACE_CHARACTER**. Depending on the type of character encoding used by the String Reader context (specified **MstrControl()** with **M_ENCODING**) any character of that specified type (either ASCII or Unicode) can be used as the space character. You can also set **M_SPACE_CHARACTER** to **M_NONE**, which specifies no space character.

The default string separator is ASCII 10, which is a new line. Therefore, when retrieving the entire text (**M_TEXT**), each string will be on a separate line, which makes it ideal for printing purposes. You can, however, control the string separator character, using **MstrControl()** with **M_STRING_SEPARATOR**. Depending on the type of character encoding used by the String Reader context (specified **MstrControl()** with **M_ENCODING**) any character of that specified type (either ASCII or Unicode) can be used as the string separator. You can also set **M_STRING_SEPARATOR** to **M_NONE**, which specifies no separator character.

You can use **MstrGetResult()** to get the size of the formatted string (**M_FORMATTED_STRING_SIZE**), the size of the non-formatted string (**M_STRING_SIZE**), and the size of the entire formatted text (**M_TEXT_SIZE**). When getting the size of the formatted string, the string's characters, the inserted space characters, and the null-terminated character are included. When getting the size of the non-formatted string, only the string's characters are included. When getting the entire formatted text, all the strings' characters, all the inserted space characters, and all the string separators are included; null-terminated characters are not included.

Fixing read problems using String Expert

Although the String Reader module is easy to use, there are many subtleties that can occasionally cause you to receive unexpected results. For example, a string might not be read or might not be read the way you expect it to be read. If this occurs, you can use the String Expert functionality in the String Reader interactive

utility, which makes a diagnosis of your String Reader settings and reports configuration problems. An alternative to using the String Reader utility is to use the String Expert functionality available using **MstrExpert()** from within your application.

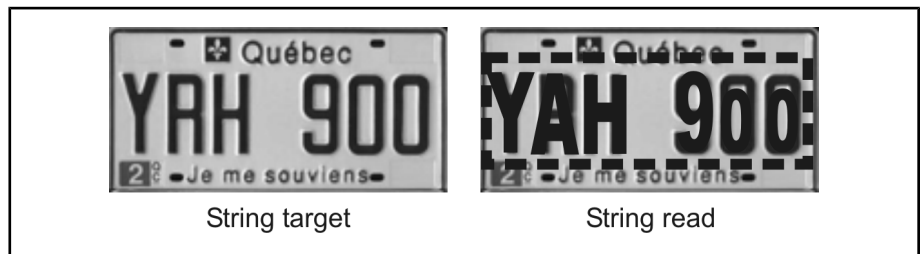
- ❖ Note that the String Expert functionality is not available for a fontless context.

To use the String Expert functionality in the String Reader utility, enter your String Reader settings and the string that you want to read and start String Expert. String Expert will then give a report of the errors and/or warnings that have been detected. Errors inform you why the string is not being read or why the string being read might not be what you expect. Warnings inform you that, although the string is being read, it is at the limit of one or more of your settings. You can choose to have String Expert only produce error reports or only warning reports but the default setting is to produce both error and warning reports. String Expert will also show you the string model associated with the specified error and/or warning.

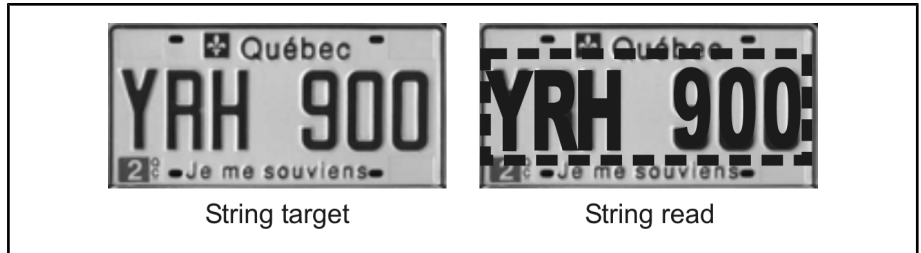
String Expert will give reports on a wide range of problems, such as an improperly set font or string model setting, as well as incorrect characters in your font or improper character constraints. For example, you might want to read a license plate that must consist of 3 letters and 3 digits, such as the license plates shown below. However, when you try to read the string from a correct source image, the string cannot be located. To establish the problem, enter the string that you expect to read (in this case, 'ZEY877') and start String Expert. When you get the error report, you might get a message saying that the string model reference scale is too high. In this case, you should lower the string scale and perform the read operation again. Now, it should finally read your string.



Another example might be that you want to read the string 'YRH900', but did not add the correct characters to the font. However, based on your string model acceptance levels, you have a valid read operation, although the string read is not what it should be. Depending on your settings and the actual characters in your font, an 'R' can be read as an 'A', or the numerical '0' character can be read as the alphabetical 'O' character.



By using the String Expert functionality and getting the error report for this situation, you will receive a message saying that the target string contains one or more characters that are not defined in your font. Upon adding these characters to the font, the read operation will then produce the results you expect.

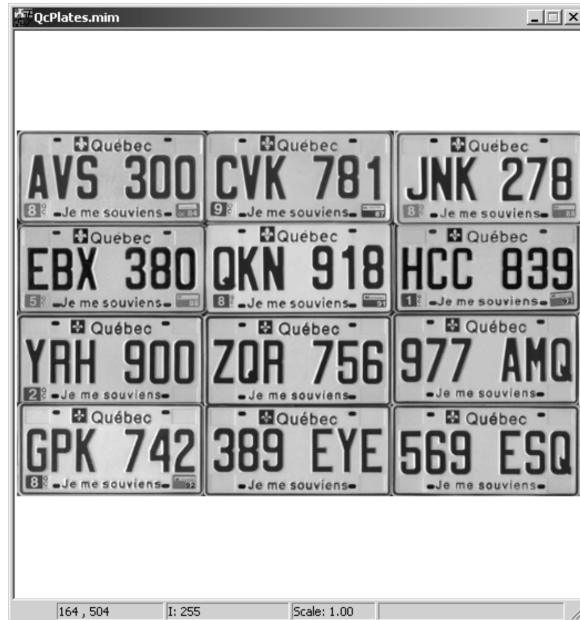


As mentioned previously, an alternative to using the String Expert functionality in the String Reader utility is using **MstrExpert()**. You might want to use this approach if you wanted to debug from within your application. To produce the error/warning report, pass **MstrExpert()** the string to read and the target image. You can also choose to only obtain an error report or only a warning report. Note that you can also use the String Expert functionality in the String Reader interactive utility by calling **MstrExpert()** with **M_INTERACTIVE**.

After calling **MstrExpert()**, you can get the errors and/or warnings that have been detected, using **MstrGetResult()**. To get the actual error or warning message, use **MstrGetResult()** with **M_REPORT_ERRORS** or **M_REPORT_WARNINGS**. To retrieve the total number of errors or warnings, use **MstrGetResult()** with **M_REPORT_NUMBER_OF_ERRORS** or **M_REPORT_NUMBER_OF_WARNINGS**. To retrieve the string associated with either a general error or warning, use **MstrInquire()** with **M_REPORT_STRING**.

An example

The String Reader example *MStr.cpp* illustrates how to define a font from an image with a mosaic of sample Quebec licence plates.



Sample Quebec license plates

Two string models are then defined and their controls are set to read two valid types of Quebec licence plates: three letters followed by three numbers, or three numbers followed by three letters. A valid licence plate reading is then performed in a target image containing a car on the road.



The example also displays the score of the string that was read, and the duration of the read operation.

```

/*****
/*
* File name: MStr.cpp
*
* Synopsis: This program uses the String Reader module to define a font
*           from an image containing a mosaic of Quebec license plates.
*           Two String Models are then defined and parameterized to read
*           only some valid Quebec license plates.
*           A license plate reading is then performed in a target image
*           of a car on the road.
*/

```

```

#include <mil.h>

/* MIL image file specifications. */
#define IMAGE_FILE_DEFINITION M_IMAGE_PATH MIL_TEXT("QcPlates.mim")
#define IMAGE_FILE_TO_READ    M_IMAGE_PATH MIL_TEXT("LicPlate.mim")

/* String containing all characters used for font definition. */
#define TEXT_DEFINITION        "AVS300CVK781JNK278 EBX380QKN918HCC839 "\
                               "YRH900ZQR756977AMQ GPK742389EYE569ESQ"

/* Font normalization size Y. */
#define NORMALIZATION_SIZE_Y  20L

/* Max size of plate string. */
#define STRING_MAX_SIZE       32L

/*****
/* Main. */
int MosMain(void)
{
    MIL_ID MilApplication,          /* Application identifier. */
    MilSystem,                     /* System identifier. */
    MilDisplay,                    /* Display identifier. */
    MilImage,                      /* Image buffer identifier. */
    MilOverlayImage,              /* Overlay image. */
    MilStrContext,                /* String context identifier. */
    MilStrResult;                 /* String result buffer identifier. */
    MIL_INT NumberOfStringRead;    /* Total number of strings to read. */
    MIL_DOUBLE Score;              /* String score. */
    MIL_TEXT_CHAR StringResult[STRING_MAX_SIZE+1]; /* String of characters read. */
    MIL_DOUBLE Time = 0.0;         /* Time variable. */

    /* Print module name. */
    MosPrintf(MIL_TEXT("\nSTRING READER MODULE:\n"));
    MosPrintf(MIL_TEXT("-----\n\n"));

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

    /* Restore the font definition image */
    MbufRestore(IMAGE_FILE_DEFINITION, MilSystem, &MilImage);

    /* Display the image and prepare for overlay annotations. */
    MdispSelect(MilDisplay, MilImage);
    MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
    MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

    /* Allocate a new empty String Reader context. */
    MstrAlloc( MilSystem, M_FEATURE_BASED, M_DEFAULT, &MilStrContext);

    /* Allocate a new empty String Reader result buffer. */
    MstrAllocResult(MilSystem, M_DEFAULT, &MilStrResult);

```

```

/* Add a new empty user defined font to the context. */
MstrControl(MilStrContext, M_CONTEXT, M_FONT_ADD, M_USER_DEFINED);

/* Add user defined characters from the license plate mosaic image. */
MstrEditFont(MilStrContext, M_FONT_INDEX(0), M_CHAR_ADD,
             M_USER_DEFINED + M_FOREGROUND_BLACK,
             MilImage, (void *)TEXT_DEFINITION, M_NULL);

/* Draw all the characters in the font in the overlay image. */
MgraColor(M_DEFAULT, M_COLOR_GREEN);
MstrDraw(M_DEFAULT, MilStrContext, MilOverlayImage, M_DRAW_CHAR,
         M_FONT_INDEX(0), M_NULL, M_ORIGINAL);

/* Normalize the characters of the font to an appropriate size. */
MstrEditFont(MilStrContext, M_FONT_INDEX(0), M_CHAR_NORMALIZE,
             M_SIZE_Y, NORMALIZATION_SIZE_Y, M_NULL, M_NULL);

/* Add 2 new empty strings models to the context for the 2 valid types of
   plates (3 letters followed by 3 numbers or 3 numbers followed by 3 letters)
   Note that the read time increases with the number of strings in the context.
*/
MstrControl(MilStrContext, M_CONTEXT, M_STRING_ADD, M_USER_DEFINED);
MstrControl(MilStrContext, M_CONTEXT, M_STRING_ADD, M_USER_DEFINED);

/* Set number of strings to read. */
MstrControl(MilStrContext, M_CONTEXT, M_STRING_NUMBER, 1);

/* Set number of expected characters for all string models to 6 characters. */
MstrControl(MilStrContext, M_STRING_INDEX(M_ALL), M_STRING_SIZE_MIN, 6);
MstrControl(MilStrContext, M_STRING_INDEX(M_ALL), M_STRING_SIZE_MAX, 6);

/* Set default constraint to uppercase letter for all string models */
MstrSetConstraint(MilStrContext, M_STRING_INDEX(0), M_DEFAULT,
                 M_LETTER + M_UPPERCASE, M_NULL );
MstrSetConstraint(MilStrContext, M_STRING_INDEX(1), M_DEFAULT,
                 M_LETTER + M_UPPERCASE, M_NULL );

/* Set constraint of 3 last characters to digit for the first string model */
MstrSetConstraint(MilStrContext, M_STRING_INDEX(0), 3, M_DIGIT, M_NULL );
MstrSetConstraint(MilStrContext, M_STRING_INDEX(0), 4, M_DIGIT, M_NULL );
MstrSetConstraint(MilStrContext, M_STRING_INDEX(0), 5, M_DIGIT, M_NULL );

/* Set constraint of 3 first characters to digit for the second string model */
MstrSetConstraint(MilStrContext, M_STRING_INDEX(1), 0, M_DIGIT, M_NULL );
MstrSetConstraint(MilStrContext, M_STRING_INDEX(1), 1, M_DIGIT, M_NULL );
MstrSetConstraint(MilStrContext, M_STRING_INDEX(1), 2, M_DIGIT, M_NULL );

/* Pause to show the font definition. */
MosPrintf(MIL_TEXT("This program has defined a font with this ")
          MIL_TEXT("Quebec plates mosaic image.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

```



```

/* Clear the display overlay. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

/* Load image to read into image buffer. */
MbufLoad(IMAGE_FILE_TO_READ, MilImage);

/* Preprocess the String Reader context. */
MstrPreprocess(MilStrContext, M_DEFAULT);

/* Dummy first call for bench measure purpose only (bench stabilization,
   cache effect, etc...). This first call is NOT required by the application. */
MstrRead(MilStrContext, MilImage, MilStrResult);

/* Reset the timer. */
MappTimer(M_TIMER_RESET+M_SYNCHRONOUS, M_NULL);

/* Perform the read operation on the specified target image. */
MstrRead(MilStrContext, MilImage, MilStrResult);

/* Read the time. */
MappTimer(M_TIMER_READ+M_SYNCHRONOUS, &Time);

/* Get number of strings read and show the result. */
MstrGetResult(MilStrResult, M_GENERAL, M_STRING_NUMBER + M_TYPE_MIL_INT,
              &NumberOfStringRead);
if( NumberOfStringRead >= 1)
{
    MosPrintf(MIL_TEXT("The license plate was read successfully (%.2lf ms)\n\n"),
              Time*1000 );

    /* Draw read result. */
    MgraColor(M_DEFAULT, M_COLOR_BLUE);
    MstrDraw(M_DEFAULT, MilStrResult, MilOverlayImage, M_DRAW_STRING, M_ALL,
              M_NULL, M_DEFAULT);

    MgraColor(M_DEFAULT, M_COLOR_GREEN);
    MstrDraw(M_DEFAULT, MilStrResult, MilOverlayImage, M_DRAW_STRING_BOX, M_ALL,
              M_NULL, M_DEFAULT);

    /* Print the read result. */
    MosPrintf(MIL_TEXT(" String                               Score\n") );
    MosPrintf(MIL_TEXT(" -----\n") );
    MstrGetResult(MilStrResult, 0, M_STRING+M_TYPE_TEXT_CHAR, StringResult);
    MstrGetResult(MilStrResult, 0, M_STRING_SCORE, &Score);
    MosPrintf(MIL_TEXT(" %s                               %.1lf\n"), StringResult, Score );
}
else
{
    MosPrintf(MIL_TEXT("Error: Plate was not read.\n"));
}

/* Pause to show results. */
MosPrintf(MIL_TEXT("\nPress <Enter> to end.\n\n"));

```

```
MosGetch();

/* Free all allocations. */
MstrFree(MilStrContext);
MstrFree(MilStrResult);
MbufFree(MilImage);

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

return 0;
}
```

Chapter

13

Codes

This chapter describes how to read, write, and verify various codes.

MIL code module

Many industries label products using symbols from symbolologies such as UPC-A bar codes and PDF417 cross-row codes. This is done for identification purposes during different stages of production and distribution. Each **symbolology**, known as a **code type** in MIL, follows a different set of rules to encode the data into light and dark patterns.

MIL can both read symbols from and write symbols to images. To read symbols, MIL searches for specified code types in an image and decodes them. The decoded strings can then be used to identify the object, or objects, 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 also allows you to verify the quality of the codes you want to read, or have written to check if the codes meet your quality specifications. To verify a code, MIL computes the quality-grade of the code in the specified source image.

MIL supports 1D, 2D, and composite code types:

- **One-dimensional (1D) code types.** These are linear bar codes used to represent data with bars and spaces. These code types are typically used to represent short strings (for example, the identification number for a grocery store item, or a stock room part number). The MIL Code module supports read operations for multiple 1D codes in the same source image.
- **Two-dimensional (2D) code types.** These are cross-row and matrix codes used to represent data with blocks stacked in rows. 2D code types can store more information than 1D code types. These code types are typically used to represent longer strings (for example, postage and paper boarding passes).
- **Composite code types.** These are a combination of specific 1D and 2D code types. The 1D code type component encodes the item's primary identification, while the 2D code type component encodes additional data (for example, batch number or expiration date).

For examples of supported code types, see the *Supported code types* section later in this chapter.

Some code types support several methods of encoding, known as **encoding schemes**. This means a code type might, for example, 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 need 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. For more information, see the *Supported code types* section later in this chapter.

Supported buffer types

The MIL Code module only supports 8-bit unsigned one-band buffers. Performing a code operation on buffers in other formats will produce an error.

Steps to reading, writing, or verifying a code in an image

The following steps provide a basic methodology for using the MIL Code module:

1. Allocate a code context, using **McodeAlloc()**. MIL uses a code context to identify and control the reading, writing, and verifying of codes.
2. Add one or more code models to the code context, using **McodeModel()**, depending on the operation you need to perform. A code model contains control settings to read, write, or verify a particular code.

A code context can contain multiple code models of 1D code types (excluding RSS, Planet, and Postnet); for 2D and composite code types, a code context can contain at most one code model when adding the model. Specify the code type of the code model when adding the code model, using **McodeModel()**.

3. If necessary, change the control settings of the code context or the code models, using **McodeControl()** (for example, change the encoding scheme for your code type using **McodeControl()** with **M_ENCODING**, or change the error correction scheme for your code type using **McodeControl()** with **M_ERROR_CORRECTION**).
4. If reading or verifying a code, allocate a result buffer to hold the code results, using **McodeAllocResult()**.
5. If reading or verifying a code, grab or load an image that contains the code into an allocated image buffer. If the image is large, and contains information which might be misinterpreted as a code, create a child buffer to isolate the code from the rest of the image. If your image contains other important information besides the code, the presearch feature can help MIL locate a 2D code; enable the feature using **McodeControl()** with **M_USE_PRESEARCH**.

- ❖ Note that you can read multiple 1D codes (excluding RSS, Planet, and Postnet code types) from the same image.

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 *Customizing write operation settings* section later in this chapter.

6. To perform a read operation, use **McodeRead()**. To perform a write operation, use **McodeWrite()**. To perform a verify operation, use **McodeVerify()**.
7. Retrieve results, using **McodeGetResult()** or **McodeGetResultSingle()**.
8. If necessary, draw the results using **McodeDraw()**.
9. If necessary, save your code context, using **McodeSave()**.
10. Free all allocated objects, using **McodeFree()**.

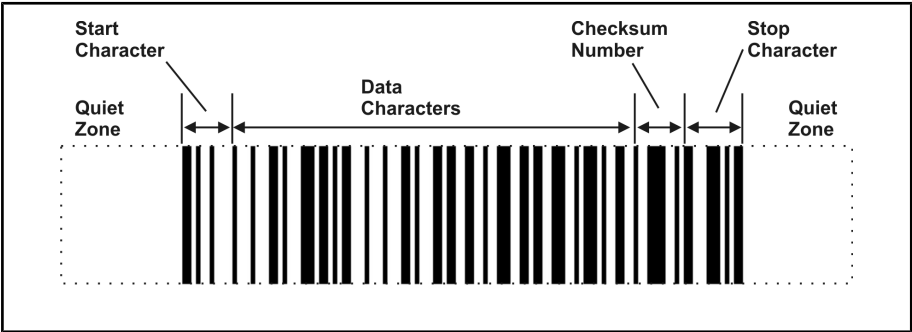
Note that you must take into consideration any particularities of the chosen code type. More detailed and code type-dependent information is described in subsequent sections of this chapter, as well as in the MIL Reference.

Once created, a MIL code context can be saved and restored as needed. Restoring this information using **McodeRestore()** rather than creating the MIL code from scratch saves time, especially if the restored code requires no further modifications.

Basic concepts

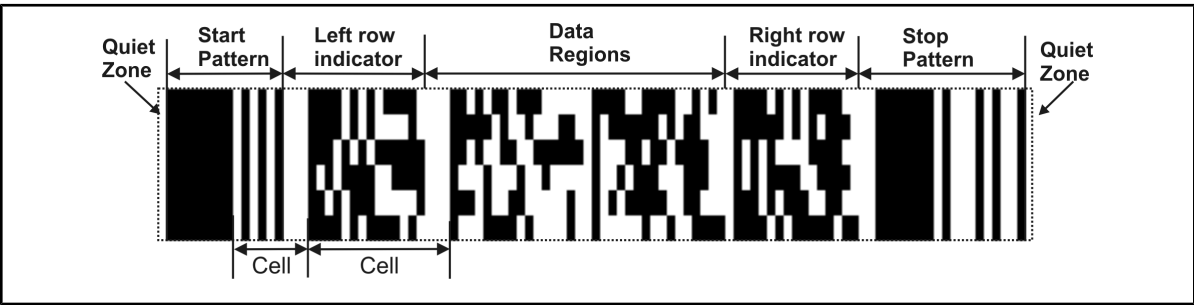
The following are terms commonly used in code operations:

- **One-dimensional (1D) code types.** 1D code types are linear bar codes containing vertical bars and spaces. The following is an example of a Code 128 code type.



- **Two-dimensional (2D) code types.** 2D code types contain blocks spanning multiple rows and spaces. There are two types of 2D codes supported by MIL: matrix codes and cross-row codes.

The following is an example of a PDF417 cross-row code type:



The following is an example of a data matrix code type:

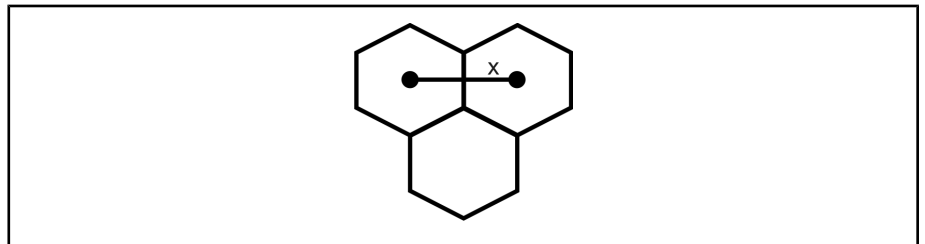


- **Aperture.** The aperture is the size of the region in which a smoothing operation is performed when computing the scan reflectance profile. The aperture should be large enough so that insignificant defects are ignored and small enough so that the contrast between the foreground and the background remains strong.
- **Bearer bars.** Bars that run along the top and bottom of a code, which are illustrated in the diagram below.



- **Cell.** The narrowest dark or white space into which data is encoded. Cells have different shapes, depending on the code type. Cells in 1D code types are bars or spaces. Some 2D codes have cells that are approximately-square blocks, while Maxicode cells are hexagonal. In 1D and 2D codes, bars can occupy several contiguous cells, as can spaces.
- ❖ When dealing with 1D code types (except Postnet and Planet) and 2D cross row codes (MicroPDF, PDF417, and Truncated PDF), industry uses the term module to refer to a cell.
- **Cell size.** Width, in pixels, of the cell, the code's narrowest unit. It is the size of the cell in X.

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



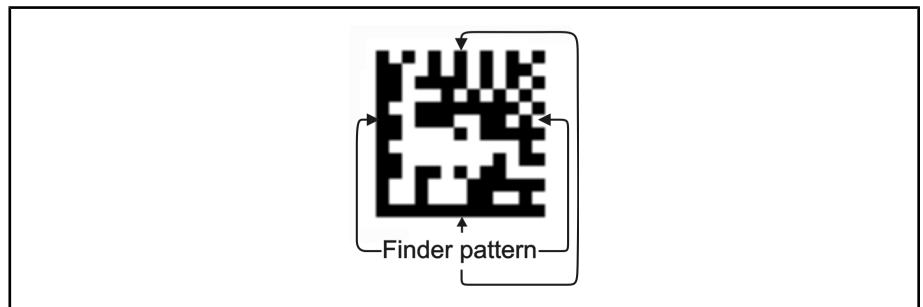
- ❖ The industry refers to the cell size as either x-width or module size.

- **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 are put through a checksum calculation whose results are compared against the checksum number.
- **Check digit.** A character included in a code for the purpose of performing a mathematical check on the value of the decoded code to ensure its accuracy.
- **Codeword.** 2D code types store data in a predetermined sequence of bars and spaces that constitute a codeword. For example, when dealing with PDF417, a codeword constitutes a consecutive sequence of 4 bars and 4 spaces totaling 17 cell. The 417 in PDF417 refers to this codeword structure. A code contains several types of codewords, including data codewords, control codewords, and row indicator codewords.
- **Code context.** The container for all code models. The code context also contains global settings that apply to the reading, writing, and the verification of the code.
- **Code model.** The container for all the control settings to read, write, or verify a particular code.
- **Column.** A series of cells along the Y-axis. Columns are only referred to when dealing with codes that have more than one row (such as 2D codes).
- **Composite code.** Composite codes combine a 1D code type and a 2D code type into one code. Composite codes are most often used to encode large strings of data.



- **Encoding scheme.** A set of rules governing how a string is encoded. Not all code types support all encoding schemes, so you must be aware of the characteristics of your code type.

- **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.
- **Erasure.** An erasure is a missing or unreadable codeword at a known position. All erasures are errors, but not all errors are erasures.
- **Finder pattern.** The finder pattern is on the perimeter of a Data Matrix code. It consists of two solid lines and two alternating light and dark lines. The finder pattern is used for symbol identification, orientation, and cell location. A finder pattern is illustrated in the diagram below.



- **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 1D code types, quiet zones are optional. For 2D matrix codes, the quiet zone must be present on all four sides of the code; for 2D cross-row codes, the quiet zone must be present on 2 sides. The quiet zone must contain no marks; even hand-written scribbles in the quiet zone will impede the read operation significantly. The size of the quiet zone is dependent on the code type.
- **Module.** See cell.
- **Scan reflectance profile.** A scan reflectance profile of a code is a record of values measured along a line across the width of the code (line profile).

- **Scan line.** A scan line is the line along which the code is read. The scan line can be in any direction, but must be straight. Multiple scanlines are needed to read the code; the first scanline is usually taken across the middle of the code. Scanlines are used with codes that have 1D components (such as, 1D code types and composite codes).
- **Scan path.** A scan path is the path along which the aperture moves across the code. Multiple scan paths are needed to verify the code; the first path is usually taken across the top of the code. Scan paths are used with codes whose information should be verified in multiple directions, such as, codes with 2D components.
- **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.

Technical code information

For detailed information about specific code types, refer to one of the following sources.

Code type		Source
1D Code types	General	<i>Automatic identification and data capture techniques — Bar code print quality test specification — Linear symbols</i>
	BC412	<i>Semi T1-95 (Reapproved 0303) Specification for back surface bar code marking of silicon wafers</i>
	Code 128	<i>Automatic identification and data capture techniques - Code 128 bar code symbology specification</i>
	Code 39	<i>Automatic identification and data capture techniques - Code 39 bar code symbology specification</i>
	EAN 8, EAN 13, UPC-A, UPC-E	<i>Automatic identification and data capture techniques - Bar code symbology specification - EAN-UPC</i>
	Interleaved 2 of 5	<i>Automatic identification and data capture techniques - Interleaved 2 of 5 bar code symbology specification</i>
	RSS	<i>Automatic identification and data capture techniques - Reduced Space Symbology (RSS) bar code symbology specification</i>
	Code 93	<i>AIM Uniform symbology specification - Code 93</i>
	Codabar	<i>AIM Uniform symbology specification - Codabar</i>
	Pharmacode	For technical information about Pharmacode, see <i>The Pharmacode Guide</i> by Laetus
	Planet code	United States Postal Service site, Rapid Information Bulletin Board System (http://www.ribbs.usps.gov)
	Postnet	United States Postal Service site, (http://www.usps.com) and find the POSTNET bar code Certification page
2D code types	General	<i>Automatic identification and data capture techniques — Bar code print quality test specification — Two-dimensional symbols</i>
	Matrix codes	<i>Specification for marking of wafers with a two-dimensional matrix code symbol Specification for back surface marking of double-side polished wafers with a two-dimensional matrix code symbol AIM Uniform symbology specification - MaxiCode AIM Uniform symbology specification - Data matrix AIM Uniform symbology specification - QR Code</i>
	Cross-row codes	<i>Automatic identification and data capture techniques - PDF417 bar code symbology specification Automatic identification and data capture techniques" MicroPDF417 bar code symbology specification</i>
Composite codes	General	<i>Automatic identification and data capture techniques - EAN.UCC Composite bar code symbology specification</i>

Supported code types

Whether you plan on reading, writing, or verifying codes, you must specify the code type when adding a code model to your code context using **McodeModel()**. If your read operation is not successful, ensure that you have added a code model using the appropriate code type.















- ❖ A code context can only contain multiple code models of 1D code types (excluding RSS, Planet, and Postnet code types). For RSS, Planet, Postnet and all 2D code types, a code context can only contain one code model.

If setting up a code context for reading codes that meet an ISO-compatible SEMI-specification, you can use a predefined code context, distributed with MIL, using **McodeRestore()**.

This section lists the supported code types, and their MIL predefined constants. It also lists the predefined SEMI code contexts.







1D code types

The following is a list of the supported 1D code types.

Code type	Sample	MIL constant
BC412		M_BC412
Codabar		M_CODABAR
Code 128		M_CODE128
Code 39		M_CODE39
Code 93		M_CODE93
EAN 8		M_EAN8
EAN 13		M_EAN13
Interleaved 2/5		M_INTERLEAVED25
Pharmacode		M_PHARMACODE
Planet		M_PLANET
Postnet		M_POSTNET
RSS Code		M_RSSCODE
UPC-A		M_UPC_A
UPC-E		M_UPC_E

2D code types


The following is a list of the supported 2D code types.

Family of code	Code type	Sample	MIL constant
Matrix codes	Data Matrix		M_DATAMATRIX
	Maxicode		M_MAXICODE
	QR code		M_QRCODE
Cross-row codes	PDF417		M_PDF417
	MicroPDF417		M_MICROPDF417
	Truncated PDF417		M_TRUNCATED_PDF417

Composite codes

A composite code combines a 1D and 2D code together in a single image. You specify the 1D and 2D components by specifying their corresponding encoding scheme. Control types that apply to either part of a composite code also apply to the composite code as a whole. For the purposes of this chapter, composite codes will not be discussed explicitly.

The following is an example of a composite code.

Code type	Sample	MIL constant
Composite code		M_COMPOSITECODE

Predefined SEMI code contexts

MIL is distributed with three ISO-compatible SEMI code contexts, each containing a code model and code model settings that match the specification. These can be used directly, or modified to suit your needs.

Code context file	Specification	Associated code type
SEMI_T1-95r0303.mfo	SEMI T1-95 (Reapproved 0303)	M_BC412
SEMI_T2-0298E.mfo	SEMI T2-0298E	M_DATAMATRIX
SEMI_T7-0303.mfo	SEMI T7-0303	M_DATAMATRIX

To use a predefined SEMI code context, restore it using **McodeRestore()** with the appropriate file name. These files are located in the *Matrox Imaging\contexts* folder. Once restored, the code context can be modified using the MIL Code functions.

Supported encoding schemes, sub-types, and error correction schemes by code type

In many cases, each of the supported code types can use one of several encoding schemes and error correction schemes. In addition, composite codes and RSS 1D codes are families of codes, with each member being a sub-type and typically having a different encoding scheme.

Supported encoding schemes and sub-types

To read, write, or verify most code types, you must ensure that the proper encoding scheme is selected for your code model using **McodeControl()** with **M_ENCODING**. This is not the case when reading or verifying RSS and composite code types; in these cases, **M_ENCODING** is ignored and the sub-type is used instead (**M_SUB_TYPE**). **M_SUB_TYPE** allows you to specify multiple sub-types so that you can limit the number of sub-types that MIL tries to read or verify. The encoding scheme is implicitly selected based on the selected sub-types. For example, when dealing with a composite code, if you specifying a sub-type of **M_RSS_EXPANDED** + **M_EAN13**, MIL reads your code more quickly because it will only look for these two family members as opposed to all the available sub-types.

❖ Note that, for best results, specify the encoding scheme when reading a Code 39 code type.

The following table lists the encoding schemes supported by each 1D code type (except RSS):

Encoding scheme (M_ENCODING)	1D code types (except RSS)			
	M_BC412 and M_CODABAR	M_CODE39 and M_CODE93	M_CODE128	M_EAN8, M_EAN13, M_UPC_A, M_UPC_E, M_INTERLEAVED25, M_PHARMACODE, M_PLANET, and M_POSTNET
M_ENC_ASCII	---	Yes	Yes	---
M_ENC_NUM	---	---	---	Yes
M_ENC_STANDARD	Yes	Yes	---	---

The following table lists the encoding schemes and sub-types supported by RSS 1D code types:

Encoding scheme (M_ENCODING)	Sub-type (M_SUB_TYPE)
M_ENC_RSS_EXPANDED	M_RSS_EXPANDED
M_ENC_RSS_EXPANDED_STACKED	M_RSS_EXPANDED_STACKED
M_ENC_RSS_LIMITED	M_RSS_LIMITED
M_ENC_RSS14	M_RSS14
M_ENC_RSS14_STACKED	M_RSS14_STACKED
M_ENC_RSS14_STACKED_OMNI	M_RSS14_STACKED_OMNI
M_ENC_RSS_LIMITED	M_RSS_LIMITED
M_ENC_RSS14_TRUNCATED	M_RSS14_TRUNCATED

The following table lists the encoding schemes supported by each 2D code type:

Encoding scheme (M_ENCODING)	2D code types			
	M_DATAMATRIX	M_MAXICODE	M_MICROPDF417, M_PDF417, and M_TRUNCATED_PDF417	M_QRCODE
M_ANY	Yes	---	Yes	Yes
M_ENC_ALPHA	Yes	---	---	---
M_ENC_ALPHANUM	Yes	---	---	---
M_ENC_ALPHANUM_PUNC	Yes	---	---	---
M_ENC_ASCII	Yes	---	---	---
M_ENC_ISO8	Yes	---	---	---
M_ENC_MODE2	---	Yes	---	---
M_ENC_MODE3	---	Yes	---	---
M_ENC_MODE4	---	Yes	---	---
M_ENC_MODE5	---	Yes	---	---
M_ENC_MODE6	---	Yes	---	---
M_ENC_NUM	Yes	---	---	---
M_ENC_QRCODE_MODEL1	---	---	---	Yes
M_ENC_QRCODE_MODEL2	---	---	---	Yes
M_ENC_STANDARD	---	---	Yes	---

The following table lists the encoding schemes, 1D part, 2D part, and sub-types supported by composite codes. The supported composite codes have a 2D part that is either MicroPDF (CC-A or CC-B) or PDF417 (CC-C). The sub-type relates to the 1D part of the composite code, while the encoding scheme relates to both the 1D and 2D parts.

Encoding scheme (M_ENCODING)	M_COMPOSITECODE		
	1D part established by M_SUB_TYPE	2D part	
		M_MICROPDF417	M_PDF417
M_ENC_EAN13	M_EAN13	Yes	---
M_ENC_EAN8	M_EAN8	Yes	---
M_ENC_UCCEAN128_MICROPDF417	M_UCCEAN128	Yes	---
M_ENC_UCCEAN128_PDF417	M_UCCEAN128	---	Yes
M_ENC_UPCA	M_UPC_A	Yes	---
M_ENC_UPCE	M_UPC_E	Yes	---
M_ENC_RSS_EXPANDED	M_RSS_EXPANDED	Yes	---
M_ENC_RSS_EXPANDED_STACKED	M_RSS_EXPANDED_STACKED	Yes	---
M_ENC_RSS_LIMITED	M_RSS_LIMITED	Yes	---
M_ENC_RSS14	M_RSS14	Yes	---
M_ENC_RSS14_STACKED	M_RSS14_STACKED	Yes	---
M_ENC_RSS14_STACKED_OMNI	M_RSS14_STACKED_OMNI	Yes	---
M_ENC_RSS14_TRUNCATED	M_RSS14_TRUNCATED	Yes	---

Supported error correction schemes

In most cases, MIL can automatically detect the error correction scheme for read and verify 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.

To specify the error correction scheme, use **McodeControl()** with **M_ERROR_CORRECTION**. If you specify a scheme that is not supported by your code type, MIL returns an error. If your code type uses a checksum, when a discrepancy exists between the calculated and stored checksum, the read operation will yield no results.

The following table lists the error correction schemes supported by each 1D encoding scheme:

Code type	Error correction scheme	
	M_ECC_CHECK_DIGIT	M_ECC_NONE
M_BC412	Yes	Yes
M_CODABAR	---	Yes
M_CODE39	Yes	Yes
M_CODE93	Yes	---
M_CODE128	Yes	---
M_EAN8	Yes	---
M_EAN13	Yes	---
M_INTERLEAVED25	Yes	Yes
M_PHARMACODE	---	Yes
M_PLANET	Yes	---
M_POSTNET	Yes	---
M_RSSCODE	Yes	---
M_UPC_A	Yes	---
M_UPC_E	Yes	---

The following table lists the error correction schemes supported by each 2D encoding scheme:

Code type	Error correction scheme				
	M_ANY	M_ECC_200, M_ECC_n	M_ECC_H, M_ECC_L, M_ECC_M, M_ECC_Q	M_ECC_REED_SOLOMON	M_ECC_REED_SOLOMON_n
M_DATAMATRIX	Yes ¹	Yes	---	---	---
M_MAXICODE	---	---	---	Yes	---
M_MICROPDF417	---	---	---	Yes	---
M_PDF417	Yes	---	---	---	Yes
M_QRCODE	Yes	---	Yes	---	---
M_TRUNCATED_PDF417	Yes	---	---	---	Yes

¹Note that M_ANY is only available for data matrix code types when reading the code.

Customizing read and verify operation settings

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

Multiple occurrences

Search region

The MIL Code module is designed to read one or many codes in an image; the module only supports searching for multiple occurrences of 1D code types (excluding RSS, Planet, and Postnet code types). If you want to read more than one of any 1D code type in the image, set the number of codes that you want to read for each code model using **McodeControl()** with **M_NUMBER**.

To set the total number of codes to read, use **McodeControl()** with **M_TOTAL_NUMBER**. The default value is **M_ALL**, which finds the expected number of occurrences specified for each model.

For example, if an application must read one code in an image, you should add a code model to the code context for each of the code's possible code types. Since only one code must be read for any image, you should set **M_TOTAL_NUMBER** to 1, and set **M_NUMBER** for each code model to 1.

The following table further demonstrates the relationship between **M_TOTAL_NUMBER** and **M_NUMBER**, by showing an example of different combinations of possible results:

Code context element	Number of occurrences set	Number of occurrences read							
		Ex. 1	Ex. 2	Ex. 3	Ex. 4	Ex. 5	Ex. 6	Ex. 7	Ex. 8
Context	5 (maximum)	5	5	5	5	5	5	5	5
Model 0	1	0	1	0	1	0	1	0	1
Model 1	3	0	0	1	1	2	2	3	3
Model 2	M_ALL	5	4	4	3	3	2	2	1

- ❖ Verify operations do not support multiple occurrences and require a code context that contains a single code model.

Search region

Child buffers

It is recommended to use a child buffer for read or verify operations, especially if your image contains more codes than your code context is configured to read; otherwise, MIL will select which codes to read or verify, up to the specified maximum number of codes to read (**McodeControl()** with **M_TOTAL_NUMBER**). Using child buffers is also recommended because it allows for a fast and robust read or verify operation if your image contains other information that might be misinterpreted as a code.

A quiet zone is an optional part of each 1D code type's specification and a required part of each 2D code type's specification. For best results, if your code has a quiet zone, your child buffer must be large enough to contain the quiet zone. Generally, the minimum quiet zone for a 1D bar code is the width of the smallest bar and space. For a 2D bar code, the width is generally the size of the cell. MIL uses the quiet zone to identify the beginning/end of the code to read. 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 to be read have adequate quiet zones to perform a robust read or verify operation and return accurate results.

Presearching

There are some situations where you cannot create a child buffer for your read or verify operation. For example, your image is very complex and contains other information besides the code. In this case, you could use the MIL Code module's presearch algorithm. This helps MIL locate 2D codes in the image prior to the read or verify operation. The presearch algorithm is only supported for 2D codes and is disabled by default for efficiency. To set the presearch option, use **McodeControl()** with **M_USE_PRESEARCH**.

Foreground color

Setting the foreground color correctly improves the results of the read or verify operations. To set the foreground color, use **McodeControl()** with **M_FOREGROUND_VALUE**. The default foreground color is black. When dealing with situations using 1D codes where the foreground color might change, set **M_FOREGROUND_VALUE** to **M_FOREGROUND_ANY**. Note that this impacts the performance of read and verify operations.

Search speed

Setting the search speed

You can specify the speed at which to perform a read or verify 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 **McodeControl()** with **M_SPEED**. If you are having problems finding the code, you might want to search at a speed lower than the default.

Timing out your search

In certain cases, a read or verify operation might take longer than necessary for your purposes. In this event, you have two options to reduce the read or verify time. You can either specify a maximum decoding time for the read or verify operation using **McodeControl()** with **M_TIMEOUT**, or you can call **McodeControl()** with **M_STOP_READ** to stop the current read or verify operation when required. For the latter option, the call to **McodeControl()** must be done from another thread.

Cell size

Cell size and number

In most cases, you should not have to specify the cell size when reading or verifying codes; the default setting is sufficient. However, for some 2D code types, such as Data Matrix and Maxicode, read and verify operations can be more robust if you specify the cell size (that is, the size of the cell in X). Specify the cell size as a range, using **McodeControl()** with **M_CELL_SIZE_MIN** and **M_CELL_SIZE_MAX**. 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 1D codes if the cell size is less than 2 pixels, and 2D codes if the cell size is less than 3 pixels, even if the size is specified.

- ❖ It is highly recommended that you specify the best cell size range possible. This is especially important when using an adaptive threshold, see the *Thresholding* subsection in the *Customizing read operation settings* section in *Chapter 13: Codes*. When performing a presearch, the best results are returned when a cell range is given if the cell size is greater than 10 or smaller than 6.

Number of cells

MIL can search for the specified 2D code type and automatically determine the number of cells within the code. To increase the speed and robustness of the operation, specify the number of cells in the X (number of columns) and Y (number of rows) directions.

To specify the number of cells, use **McodeControl()** with **M_CELL_NUMBER_X** and **M_CELL_NUMBER_Y**. Note that specifying the number of cells is mostly useful when reading PDF417, Truncated PDF417, Data Matrix, and QRCode code types; the Maxicode code type has a fixed number of cells. For the PDF417 code type, **M_CELL_NUMBER_X** must equal $17c + 35$, where c is the number of columns, and 35 represents the number of cells required for the start and stop patterns. For the Truncated PDF417 code type, **M_CELL_NUMBER_X** must equal $17c + 18$, where c is the number of columns, and 18 represents the number of cells required for the start patterns.

For the PDF417 and the Truncated PDF417 code types, **M_CELL_NUMBER_Y** is used when calculating the **M_CELL_NUMBER_X**. For the MicroPDF417 code type, this setting should be a value between 1 and 4.

Search angle**Angular search range**

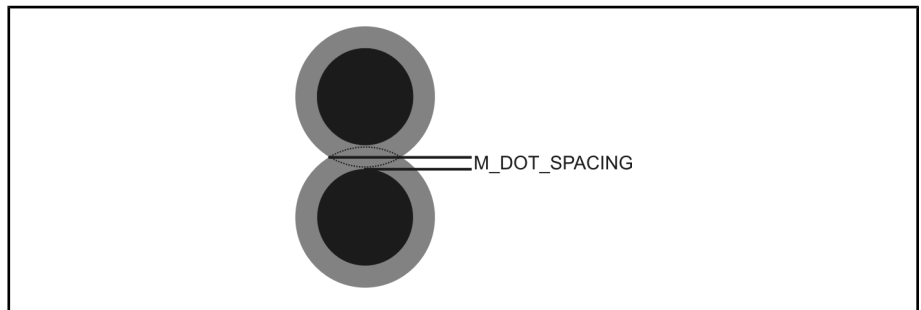
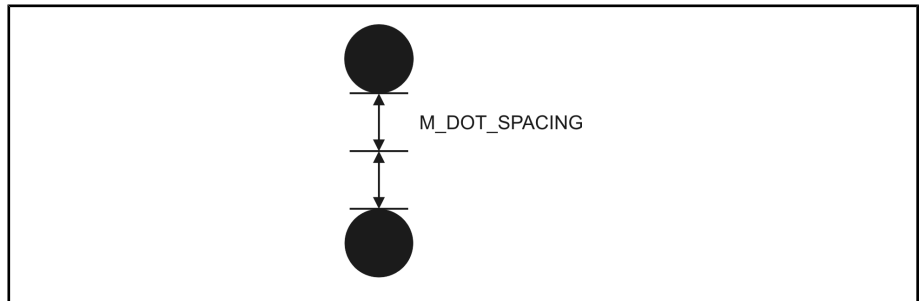
By default, MIL can read and verify codes if they fall within the angular range of $0 \pm 5^\circ$. If the code appears rotated in the image and you do not know the code's exact orientation, specify the nominal angle using **McodeControl()** with **M_SEARCH_ANGLE**. If you are unsure of the code's exact orientation, adjust the angular range using **McodeControl()** with **M_SEARCH_ANGLE_DELTA_NEG** and **M_SEARCH_ANGLE_DELTA_POS**. The angular range is the range of angles defined by **M_SEARCH_ANGLE - M_SEARCH_ANGLE_DELTA_NEG** and **M_SEARCH_ANGLE + M_SEARCH_ANGLE_DELTA_POS**. The operation speed is slower for angular ranges greater than $\pm 5^\circ$ when searching for 1D codes. The angular range does not affect the speed of the operation when searching for 2D code types. Note that you must specify the search angle if your code has bearer bars and you are not using **McodeControl()** with **M_BEARER_BAR**.

If the codes in your target image are very close to their nominal angles, you can disable searches at non-zero angles using **McodeControl()** with **M_SEARCH_ANGLE_MODE**. Disabling these calculations might speed up the search depending on the model and the target.

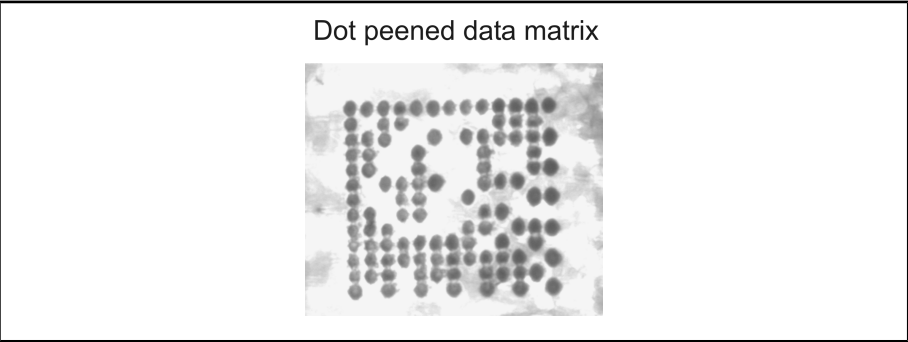
In some cases, you might need to flip the image of a Data Matrix code vertically or horizontally (using **MimFlip()**), to be read correctly. This is the case when, due to acquisition or printing, the image is essentially a mirror image of the correct code.

Dot spacing

Sometimes 2D matrix codes are composed of dots, such as when a code is punched into an object. Often in these cases, the dots have some spacing between them, which can make the code more difficult to read. For best results, set the dot spacing (using **McodeControl()** with **M_DOT_SPACING**) to half the pixel distance between dots. If a 2D code composed of dots is printed rather than punched, and the cells overlap due to printing artifacts or ink spread, set **M_DOT_SPACING** to half the depth of the overlap.



The following is an example of a code that might need you to specify the dot spacing.



String size

String size
Read and verify operations search for a code that encodes a string. The size of the string can be optionally specified, using **McodeControl()** with **M_STRING_SIZE_MIN** and **M_STRING_SIZE_MAX**.

String size

If you have set **M_ERROR_CORRECTION** to **M_ECC_CHECK_DIGIT**, the read operation assumes that there is a check digit at the end of the string. The specified string size should not include the check digit, since the check digit is not returned as part of the string.

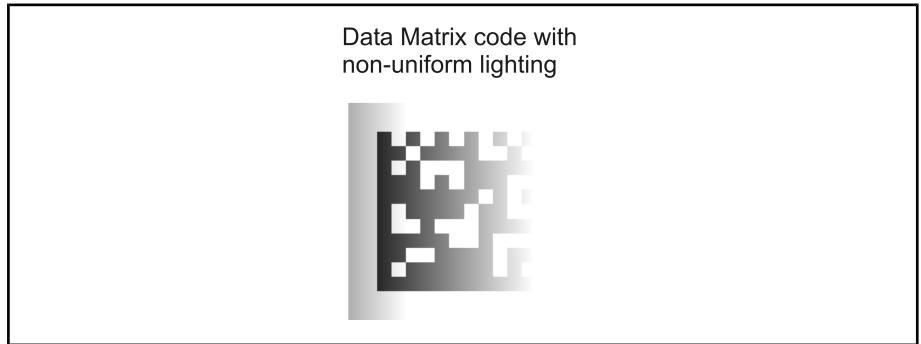
The following table lists code types that require a minimum and maximum string size to be specified:

Code type	String size	
	Minimum	Maximum
M_POSTNET	5, 9, or 11	5, 9 or 11
M_PLANET	11 or 13	11 or 13
M_CODABAR	3	--
M_EAN8	8	8
M_EAN13	13	13
M_UPC_A and M_UPC_E	12	12

Threshold value

Thresholding

During a read or verify 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, using **McodeControl()** with **M_THRESHOLD**. In the case of non-uniform lighting (see the image below for an example), when dealing with 2D matrix code types, MicroPDF417, and 1D code types (excluding Planet and Postnet), it is recommended that you set **M_THRESHOLD** to **M_ADAPTIVE**. This adaptive threshold mode computes, for each pixel, a threshold value based on its neighborhood.



When dealing with 1D code types (excluding Planet and Postnet) and using adaptive thresholding (**McodeControl()** with **M_THRESHOLD** set to **M_ADAPTIVE**), you must specify the minimum contrast (**McodeControl()** with **M_MINIMUM_CONTRAST**) between the foreground and background in the target image so that **McodeRead()** reads the code properly. Valid values are between 1 and 255. The default value is 50.

Generally, increasing the minimum contrast will make the read operation more robust to non-uniform lighting or noise so that the code is readable; however, in some instances, it could make a readable code unreadable. For example, if the contrast between a valid foreground pixel and its background is 25, and you set the minimum contrast to 50, the pixel will not be considered as a foreground pixel. Therefore, it is important to make sure that the minimum contrast value is not causing the read operation to ignore code features. In most cases, you can probably estimate an appropriate minimum contrast by looking at your image.

When dealing with 2D code types, the minimum contrast is automatically determined, so the adaptive threshold mode ignores the **M_MINIMUM_CONTRAST** setting.

When dealing with a composite code, the 1D and 2D parts are dealt with individually when using the default threshold. If the threshold is set to a value, both parts are read using the threshold that you set.

Customizing write operation settings

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

It is strongly recommended that you define your codes as completely as possible using the available control settings, since 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 and a MIL error will result.

Size of destination
image

Destination buffer size

The destination image buffer 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 **McodeInquire()** with **M_WRITE_SIZE_X** and **M_WRITE_SIZE_Y**.

Special characters

If you need to encode unprintable characters, such as a carriage return or tab, use a code type that supports them. If you use **McodeWrite()** with **M_ESCAPE_SEQUENCE**, you can use ASCII codes to encode all unprintable characters. In the string to encode, the unprintable 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 `"\\"`.

The encoded string for a Codabar code type must be numeric, but can contain the following characters: minus (-), dollar sign (\$), colon (:), slash (/), period (.), and plus (+). In addition, the string must start and end with any of the following characters: a, b, c, or d. The encoded string for a Maxicode code type with the **M_ENC_MODE2** or **M_ENC_MODE3** encoding scheme, must respect the structured carrier message format (the portion of the string that contains postal code, country code, and class of service information).

Foreground color

Setting the foreground color is essential for write operations. To set the foreground color, use **McodeControl()** with **M_FOREGROUND_VALUE**. The default foreground color is black.

Cell size and number**Cell size**

In most cases, you should not have to specify the cell size when writing codes; the default setting (**McodeControl()** with **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 **McodeControl()** with **M_CELL_SIZE_MIN** to force the cell size; the cell 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 *Destination buffer size* subsection in this section.

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

During a write operation, the number of cells will be automatically chosen to minimize the code written. To specify the number of cells in a 2D code, use **McodeControl()** with **M_CELL_NUMBER_X** and **M_CELL_NUMBER_Y**. 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. Specifying the number of cells is mostly useful when writing PDF417, Truncated PDF417, Data Matrix, and QR Codes code types; the Maxicode code type has a fixed number of cells. For the PDF417 code type, **M_CELL_NUMBER_X** must equal $17c + 35$, where c is the number of columns, and 35 represents the number of cells required for the start and stop patterns. For the Truncated PDF417 code type, **M_CELL_NUMBER_X** must equal $17c + 18$, where c is the number of columns, and 18 represents the number of cells required for the start patterns.

Bearer bars

Note that MIL does not generate bar codes with bearer bars. 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.

Verifying your code

Verification is used to check a code against quality specifications. Validation requires a better set of conditions (such as, lighting, code clarity, resolution, and uniformity of reflectance) than when reading codes. You can use **McodeVerify()** to verify the quality of 1D, 2D, or composite codes. Since global context settings are required to perform a verify operation, **McodeVerify()** requires that you pass it a code context identifier, and not a code model identifier, as a parameter.

- ❖ Note that **McodeVerify()** requires a code context with a single code model and is not supported for Pharmacode, Postnet, and Planet code types.

Attributes that can be verified

The following table lists the type of code attributes that can be verified, including the attributes of the scan reflectance profiles. To determine which of the following attributes can be verified for codes of a specific family of code types, refer to the *Results by code type* section later in this chapter.

Attribute	Notes
Axial nonuniformity	This is a measure of how spacing between sampling points differs between the X- and Y-axis (width to height).
Decodability	This attribute is used to measure the code against the permitted tolerances of the decode algorithm for the specific code type used. The better the code matches the expected code, the higher the decodability grade.
Decode	This attribute is used to verify whether the scan reflectance profiles of the code can be decoded.
Defects	This attribute is used to rank the worst deviation in the scan reflectance profiles of the code as a ratio of the defect found to the symbol contrast ($ERN_{max} / SymbolContrast$). A low grade depicts the presence of a significant defect, whereas a high grade depicts the presence of insignificant or no defects in the scan reflectance profile.
Edge determination	This attribute is used to verify whether the scan reflectance profile has the appropriate number of cells.
Minimum edge contrast	This attribute is used to verify whether the minimum contrast between two adjacent regions (for example, a bar and an adjacent space) is acceptable.
Minimum reflectance	This attribute is used to verify the reflectance of the darkest part of the code. Reflectance is a measure of the light reflected off the code. If the minimum reflectance is more than half of the maximum reflectance, then the code fails the quality test.
Modulation	This attribute is used to verify the intensity of the edge contrast of your code relative to the scan reflectance profile's contrast (symbol contrast).
Overall symbol grade	This is a measure of the worst grade for the results of the code.
Print growth	This is a measure of the extent in which the boundaries of the black and white markings of the code are within their cell's boundaries.
Start/stop pattern	This attribute is used to verify the quality of the set pattern that marks the end points of the code.
Symbol contrast	This attribute is used to verify the code contrast. The higher the contrast, the better it is for scanning purposes and the better the grade.
Unused Error Correction	This is a measure of the extent to which regional or spot damage in the code has eroded the reading safety margin that error correction provides.

Controls that must be set

There are some control types in `McodeControl()` that must be set before you verify a code. These are listed in the following table:

Control type	Notes
<code>M_FOREGROUND_VALUE</code>	Specifies the foreground color of the code. This control type is essential; the code will not be graded if the foreground value is not correctly set.
<code>M_ENCODING</code>	Specifies the type of encoding scheme for the code. This control has to be set for the code types where <code>M_ANY</code> is not supported.
<code>M_ERROR_CORRECTION</code>	Specifies the type of error correction for the code. This control has to be set for the code types where <code>M_ANY</code> is not supported.
<code>M_STRING_SIZE...</code>	Sets the maximum and minimum size of the string (number of characters) encoded in each code. These control types have to be set for the code types where <code>M_STRING_SIZE_MIN</code> and/or <code>M_STRING_SIZE_MAX</code> cannot be set to <code>M_ANY</code> .
<code>M_SEARCH_ANGLE...</code>	Specifies the nominal search angle and angular search range. These control types have to be specified if the code is not within the angular range of 0 ± 5 degrees.

Setting the aperture

The aperture can be set whenever dealing with 1D code types, 2D cross-row code types, and composite codes. The aperture is the size of the region in which a smoothing operation is performed. Ideally, the aperture size is set large enough so that the smoothing effect eliminates insignificant defects and small enough so that the contrast between the foreground and the background remains strong. If the aperture is not set correctly, the verification results (code grades) will not be accurate.

There are two ways to set the aperture for your MIL code: you can have MIL calculate the aperture size as a multiplicative factor of the cell size of your code, or you can set your aperture size to an absolute pixel value.

To have MIL calculate the aperture size, perform the following:

1. Set the aperture mode to relative, using **McodeControl()** with **M_APERTURE_MODE** set to **M_RELATIVE**.
2. Perform one of the following, depending on which best applies to your situation:
 - To have MIL calculate your aperture size according to the *ISO 15416* specification (or the *ISO 15420* specification for EAN/UPC), set **M_RELATIVE_APERTURE_FACTOR** to **M_AUTO**, and calculate **M_PIXEL_SIZE_IN_MM**, by proceeding to step 3.
 - ❖ Note that, if the resulting cell size in millimeters (that is **M_CELL_SIZE** x **M_PIXEL_SIZE_IN_MM**) is less than 0.1 millimeters, the aperture size is calculated using the following formula: $0.5 \times \text{M_CELL_SIZE}$.
 - To have MIL calculate your aperture size according to the relative aperture factor that you specify, set **M_RELATIVE_APERTURE_FACTOR** to a specific value. In this case, set **M_PIXEL_SIZE_IN_MM** to **M_UNKNOWN**. The aperture size is calculated using the following formula:
 $\text{M_RELATIVE_APERTURE_FACTOR} \times \text{M_CELL_SIZE}$.

An aperture factor that is too low can make the verify operation too sensitive to insignificant defects. Whereas, an aperture factor that is too high reduces the contrast between the foreground and the background.

 - To have MIL calculate your aperture size using an average value, set **M_RELATIVE_APERTURE_FACTOR** to **M_AUTO**. Set **M_PIXEL_SIZE_IN_MM** to **M_UNKNOWN**. The aperture size is calculated using the following formula: $0.5 \times \text{M_CELL_SIZE}$.
3. If having MIL calculate the aperture size according to *ISO 15416* or *ISO 15420*, set the pixel size of your code in millimeters, as follows:
 - a. Measure the smallest bar or space in your code with a ruler. Note that this result must be in millimeters.
 - b. Read the code from a typical image of the code, using **McodeRead()**.
 - c. Get the cell size in pixels, using **McodeGetResult()** with **M_CELL_SIZE**.

- d. Divide the measurement in millimeters by the cell size in pixels.
 - e. Set **M_PIXEL_SIZE_IN_MM** to this result.
- ❖ Note that if the setup for acquiring the code image is significantly changed (making the acquired code either significantly larger or smaller than the code used to perform the pixels in millimeters calculation), the calculation should be performed again and **M_PIXEL_SIZE_IN_MM** set to the new value.

To set the aperture size to an absolute pixel value (so that it will not change regardless of the cell size of your code image), use **McodeControl()** with **M_APERTURE_MODE** set to **M_ABSOLUTE**. Then, set the absolute aperture size to the known aperture size (in pixels), using **M_ABSOLUTE_APERTURE_SIZE**.

Retrieving results

To retrieve results for all code model occurrences, use **McodeGetResult()** with the result type for the result that you want returned. To retrieve a result for a specific code model occurrence, use **McodeGetResultSingle()**.

To retrieve the success of a read or verify operation, use **M_STATUS**. A positive status is returned only when all conditions, set for all the code models, are met. Note that when reading multiple code occurrences and more than one read operation fails, the error returned is the one you should probably fix first. For example, if you try to read two codes and one fails because the code was not found (**M_STATUS_NOT_FOUND**) and the other fails because the encoding type was unknown (**M_STATUS_ENC_UNKNOWN**), the latter is returned.

Every occurrence of every code model is associated with a unique index within the result buffer after a successful read or verify operation. When using **McodeGetResultSingle()**, you must specify the result index that you want to retrieve.

The following table presents an example of a code result buffer, showing the model index, angle, and string for 6 occurrences of 4 models in a context. If, for instance, you wanted to get additional results about only one of these occurrences, you should pass the result index to **McodeGetResultSingle()** along with the required result type. Results are ordered in the sequence that they are found.

Result index	0	1	2	3	4	5
M_CODE_MODEL_INDEX	3	3	1	0	2	0
M_ANGLE	9.0	7.0	2.0	4.0	0.0	0.0
M_STRING	First occurrence of code model 3	Second occurrence of code model 3	Only occurrence of code model 1	First occurrence of code model 0	Only occurrence of code model 2	Second occurrence of code model 0

The result index is not explicitly returned by any function in the MIL Code module, instead it must be deduced from the order in which the results are returned by **McodeGetResult()**; results are always indexed in ascending order starting from zero.

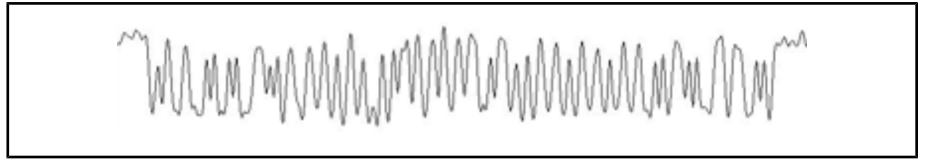
For each code occurrence read, you can retrieve either the decoded string or the length of the string, using **M_STRING** or **M_STRING_SIZE**, respectively. Combining **M_ESCAPE_SEQUENCE** with **M_STRING** will return the string with unprintable characters represented by their ASCII code (in hexadecimal notation, prefixed by \x). Combining **M_ESCAPE_SEQUENCE** with **M_STRING_SIZE** will return the length of the decoded string with the ASCII characters used to represent unprintable characters included in the count.

Drawing results

The **McodeDraw()** function provides several operations for drawing results in any specified image buffer. These operations can annotate an image non-destructively when drawn into the display's overlay buffer (see the the *Annotating the displayed image nondestructively* section in *Chapter 20: Displaying an image*).

You can use a previously allocated graphics context (see *Chapter 21: Generating graphics*) to control the drawing color, or use the default graphics context (**M_DEFAULT**).

Supported drawing operations include drawing the bounding box of the specified code occurrence or a cross-like symbol at the mid-point of the code occurrence. **McodeDraw()** can also draw the scan profile lines (scan paths) or the scan reflectance profiles of the code, as analyzed by the verify operation. The scan reflectance profile is generated from sampling the code along a scan path using a specific aperture. The following is an example of a scan reflectance profile of the code.



The scan profile is one of the scan paths used when performing a verification operation. Note that, unless otherwise specified, all scan profiles will be drawn. To draw a specific scan profile, set the **ResultIndex** of **McodeDraw()**. The following is an example of 10 of the scan profiles draw over the existing image of the code.



When drawing composite codes, you can draw either the 2D (**M_2D_COMPONENT**) or the 1D (**M_LINEAR_COMPONENT**) component of the code.

Results by code type

In the MIL Reference, the result types of **McodeGetResult()** and **McodeGetResultSingle()** are organized by their association with a specific operation type (that is, read and verify). The following table organizes the result types by their association with a code type. Note that verification results cannot be returned for the following 1D code types: Pharmacode, Postnet, and Planet code types.

Result type	1D codes	2D codes		Composite codes	Supported operation
		Matrix codes	Cross-row codes		
M_ANGLE	Yes	Yes	Yes	Yes	R
M_AXIAL_NONUNIFORMITY, M_AXIAL_NONUNIFORMITY_GRADE	No	Yes	No	No	V
M_BOTTOM_LEFT_X, M_BOTTOM_RIGHT_X, M_BOTTOM_LEFT_Y, M_BOTTOM_RIGHT_Y	Yes	Yes	Yes	Yes	R
M_CELL_NUMBER_X, M_CELL_NUMBER_Y	Yes	Yes	Yes	Yes	R
M_CELL_SIZE	Yes	Yes	Yes	Yes	R,
M_CODE_MODEL_ID, M_CODE_MODEL_INDEX	Yes	Yes	Yes	Yes	R, V
M_CODE_TYPE	Yes	Yes	Yes	Yes	R, V
M_CODEWORD_DECODABILITY, M_CODEWORD_DECODABILITY_GRADE	No	No	Yes	Yes	V
M_CODEWORD_DEFECTS, M_CODEWORD_DEFECTS_GRADE	No	No	Yes	Yes	V
M_CODEWORD_MODULATION, M_CODEWORD_MODULATION_GRADE	No	No	Yes	Yes	V
M_CODEWORD_YIELD, M_CODEWORD_YIELD_GRADE	No	No	Yes	Yes	V
M_DECODE_GRADE	No	Yes	No	No	V
M_DECODABILITY_GRADE	No	No	Yes	Yes	V
M_DEFECTS_GRADE	No	No	Yes	Yes	V
M_ENCODING	Yes	Yes	Yes	Yes	R, V
M_ERROR_CORRECTION	Yes	Yes	Yes	Yes	R, V
M_FOREGROUND_VALUE	Yes	Yes	Yes	Yes	R
M_MODULATION_GRADE	No	No	Yes	Yes	V
M_NUMBER	Yes	Yes	Yes	Yes	R, V
M_NUMBER_OF_CODEWORDS	No	Yes ¹	Yes	Yes ¹	R, V
M_NUMBER_OF_ERASURES	No	Yes ¹	Yes	Yes ¹	R, V
M_NUMBER_OF_ERRORS	No	Yes ¹	Yes	Yes ¹	R, V
M_NUMBER_OF_ERROR_CORRECTION_CODEWORDS	No	Yes ¹	Yes	Yes ¹	R, V
M_NUMBER_OF_ROWS	Yes	No	Yes	Yes	V
M_NUMBER_OF_SCANS, M_NUMBER_OF_SCANS_PER_ROW	Yes	No	Yes	Yes	V

Result type	1D codes	2D codes		Composite codes	Supported operation
		Matrix codes	Cross-row codes		
M_OVERALL_SYMBOL_GRADE	Yes	Yes	Yes	Yes	V
M_POSITION_X, M_POSITION_Y	Yes	Yes	Yes	Yes	R
M_RECOMMENDED_APERTURE_SIZE	Yes	Yes	Yes	Yes	R
M_PRINT_GROWTH, M_PRINT_GROWTH_GRADE	No	Yes	No	No	V
M_ROW_OVERALL_GRADE	Yes	No	Yes	Yes	V
M_SCAN_DECODABILITY, M_SCAN_DECODABILITY_GRADE	Yes	No	Yes	Yes	V
M_SCAN_DECODE_GRADE	Yes	No	Yes	Yes	V
M_SCAN_DEFECTS, M_SCAN_DEFECTS_GRADE	Yes	No	Yes	Yes	V
M_SCAN_EDGE_CONTRAST_MINIMUM, M_SCAN_EDGE_CONTRAST_MINIMUM_GRADE	Yes	No	Yes	Yes	V
M_SCAN_EDGE_DETERMINATION_GRADE	Yes	No	Yes	Yes	V
M_SCAN_ERN_MAXIMUM	Yes	No	Yes	Yes	V
M_SCAN_GUARD_PATTERN, M_SCAN_GUARD_PATTERN_GRADE	Yes ³	No	No	Yes ³	V
M_SCAN_INTERCHARACTER_GAP, M_SCAN_INTERCHARACTER_GAP_GRADE	Yes ⁴	No	No	No	V
M_SCAN_MODULATION, M_SCAN_MODULATION_GRADE	Yes	No	Yes	Yes	V
M_SCAN_PRINT_CONTRAST_SIGNAL	Yes	No	Yes	Yes	V
M_SCAN_PROFILE_END_X, M_SCAN_PROFILE_END_Y, M_SCAN_PROFILE_START_X, M_SCAN_PROFILE_START_Y	Yes	No	Yes	Yes	V
M_SCAN_QUIET_ZONE, M_SCAN_QUIET_ZONE_GRADE	Yes ²	No	No	Yes ⁵	V
M_SCAN_REFLECTANCE_MINIMUM_GRADE, M_SCAN_REFLECTANCE_MAXIMUM, M_SCAN_REFLECTANCE_MINIMUM	Yes	No	Yes	Yes	V
M_SCAN_REFLECTANCE_PROFILE_GRADE, M_SCAN_REFLECTANCE_PROFILE_LENGTH, M_SCAN_REFLECTANCE_PROFILE_VALUES	Yes	No	Yes	Yes	V
M_SCAN_SYMBOL_CONTRAST, M_SCAN_SYMBOL_CONTRAST_GRADE	Yes	No	Yes	Yes	V
M_SCAN_WIDE_TO_NARROW_RATIO, M_SCAN_WIDE_TO_NARROW_RATIO_GRADE	Yes ⁶	No	No	No	V
M_SCORE	Yes	Yes	Yes	Yes	R
M_SIZE_X, M_SIZE_Y	Yes	Yes	Yes	Yes	R
M_START_STOP_PATTERN_GRADE	No	No	Yes	Yes	V
M_STATUS	Yes	Yes	Yes	Yes	V
M_STRING, M_STRING_SIZE, M_TOTAL_STRING_SIZE	Yes	Yes	Yes	Yes	R, V

Result type	1D codes	2D codes		Composite codes	Supported operation
		Matrix codes	Cross-row codes		
M_SYMBOL_CONTRAST, M_SYMBOL_CONTRAST_GRADE	No	Yes	No	No	V
M_THRESHOLD	Yes	Yes	Yes	Yes	R
M_TIMEOUT_END	Yes	Yes	Yes	Yes	R, V
M_TOP_LEFT_X, M_TOP_LEFT_Y, M_TOP_RIGHT_X, M_TOP_RIGHT_Y	Yes	Yes	Yes	Yes	R
M_UNUSED_ERROR_CORRECTION, M_UNUSED_ERROR_CORRECTION_GRADE	No	Yes	Yes	Yes	V

¹If dealing with Data Matrix code types, this result type is only available when using a Reed Solomon-based algorithm as an error correction scheme.

²This result type is not available when dealing with RSS code types.

³This result type is only available when dealing with RSS code types and composite code types encoded with an RSS format.

⁴This result type is only available when dealing with Codabar and Code 39 code types.

⁵This result type is only available when dealing with composite code types encoded with a format of either EAN/UPC or UCC/EAN120.

⁶This result type is only available when dealing with Codabar, Code 39, and Interleaved 2/5 code types.

Chapter

14

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 subpixel accuracy and results can be returned in pixels or real-world units. For more information, see

Chapter 5: Camera calibration.

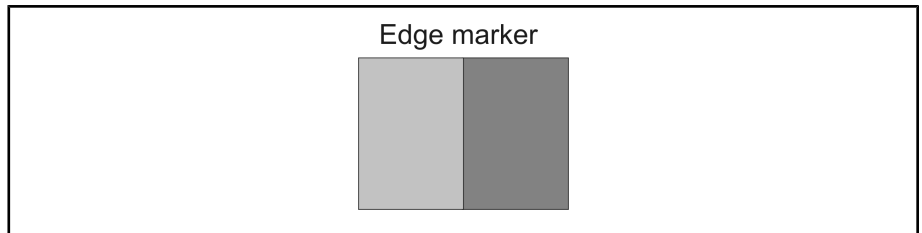
This chapter discusses finding and obtaining measurements of markers, how to define and set marker characteristics, and then the steps that are generally followed when taking measurements between markers.

Markers

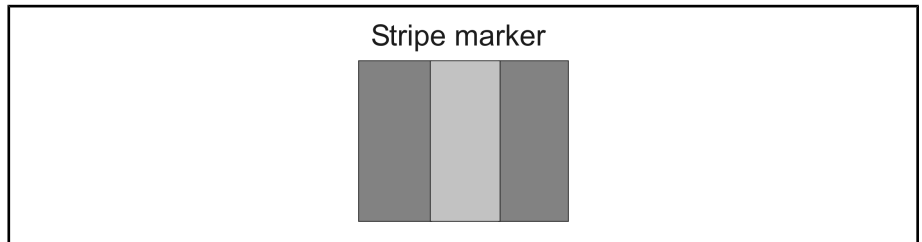
To take any type of measurement with the MIL Measurement module, you must first define your markers, using **MmeasSetMarker()**. The marker contains the image characteristics to search for in the target image.

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. In an image, edges are curves that delineate a boundary, which can be established from intensity transitions.



- **Stripe marker.** A marker consisting of a stripe (two edges) or multiple stripes. The edges of a stripe marker 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. You can specify the number of edges or stripes to locate, using **MmeasSetMarker()** with **M_NUMBER**. 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

The following steps provide a basic methodology for using the MIL Measurement module:

1. Allocate an edge or stripe marker. For a multiple marker, set the number of occurrences of the edge or stripe.
2. Place the measurement box and set the marker's characteristics.
3. Grab or load a target image. Optionally, preprocess it to improve its quality.
4. Find the marker in the target image and calculate measurements.
5. Read the results.

In general, the first three steps are performed once, while steps 4 and 5 are repeated as required. Note, you can avoid step 1 to 3 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 function 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 and other marker characteristics

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.

To improve the search, you should also specify the marker's characteristics, such as its approximate width, orientation, and polarity, using **MmeasSetMarker()**.

You can inquire about the current settings of a marker's characteristics, using **MmeasInquire()**.

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 to control the drawing color, or use the default graphics context (see the *Preparing for graphics* section in *Chapter 21: Generating graphics*). You can draw directly into the image buffer, or annotate the image non-destructively by drawing into its display overlay buffer. For more information, see the *Annotating the displayed image nondestructively* section in *Chapter 20: Displaying an image*.

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 preprocessing 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 edges' location. 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 (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 **MmeasGetResult()** with **M_NUMBER**. Global results, such as the maximum, minimum, mean, or standard deviation of any characteristic of the result group can be returned.

Annotating results

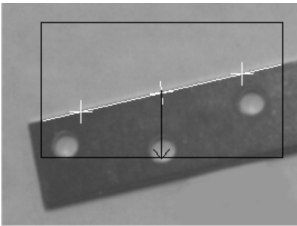
In addition to the **Mgra...** functions available for annotating your results, the **MmeasDraw()** function provides several additional operations for drawing features of the marker occurrences. For example, you can draw the edge profile of the occurrence, a cross at the found position, the width of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

Measurement examples

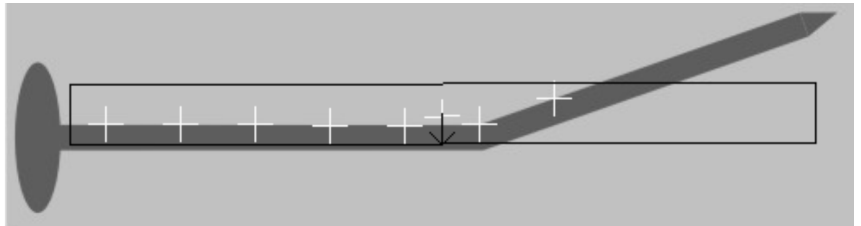
For examples on finding and obtaining measurements of markers, see the *Measurement examples* section later in this chapter.

Measurement box

A marker's measurement box indicates the region of the target image in which to search for the marker at a specified orientation. For more complex measurements, you can process this region in sections. The measurement module will find and fit a line to points marking the strongest pixel intensity transition for a subregion of each section. The default setting of the measurement box is the whole image with one section.



The result of a typical measurement operation

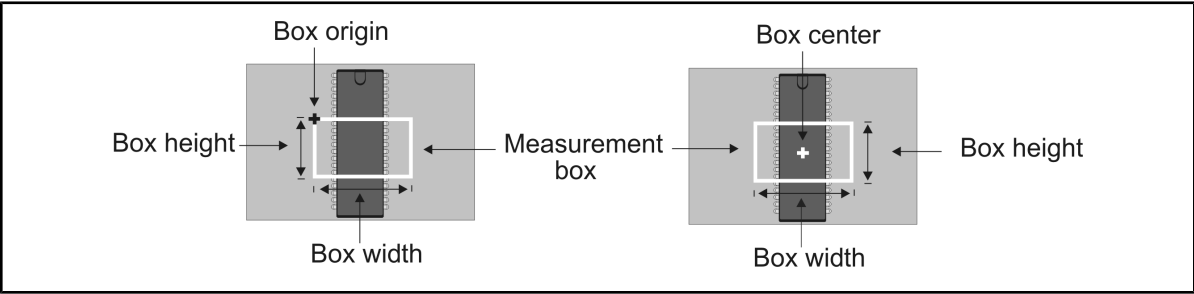


Subedge positions can be compared to detect irregularities

Placing the measurement box

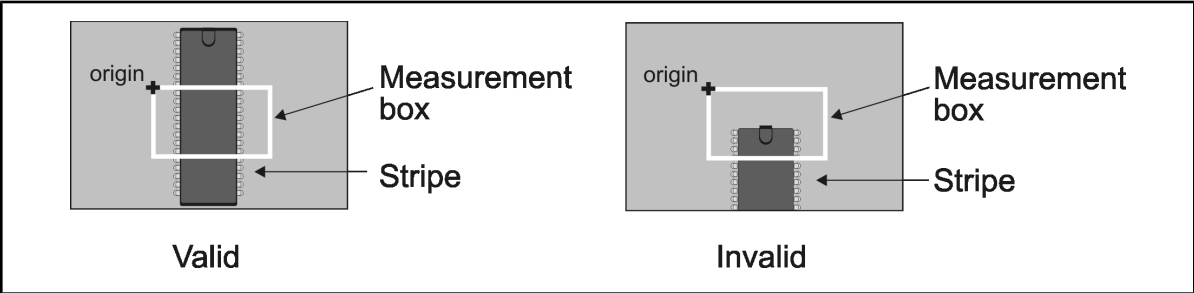
Placing the measurement box properly is essential to the success of the measurement operation, and depends on the orientation of the marker. Typically, the box should be placed so that the edges of the marker enter and leave opposite sides of the measurement box. The only exception is when subregions are used (discussed later).

Set the measurement box's size using `MmeasSetMarker()` with `M_BOX_SIZE`. You can set the box's position by either specifying its origin or center, with `M_BOX_ORIGIN` or `M_BOX_CENTER`, respectively.



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

The measurement box should be limited to as small an area as possible that still contains the marker. This will increase the chance of success of the operation, especially when using a detailed or complex target image.

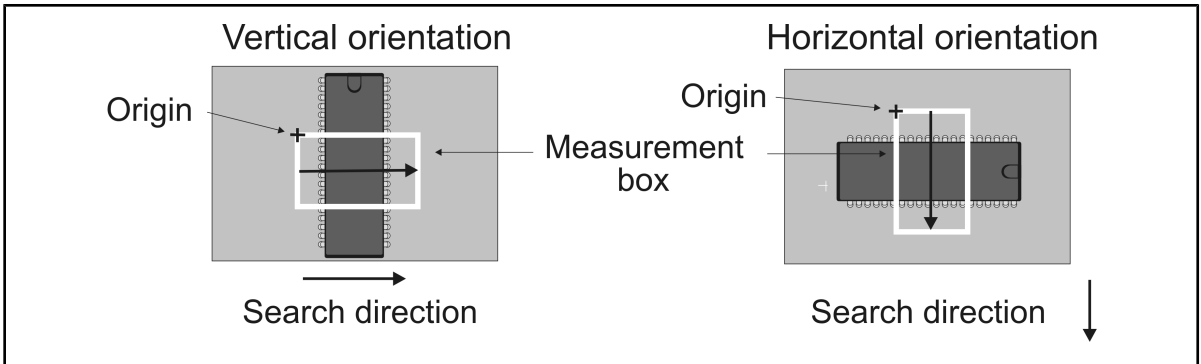


- ❖ The measurement box must fit entirely inside the target image unless subregions are used (discussed later).

Orientation

Marker orientation

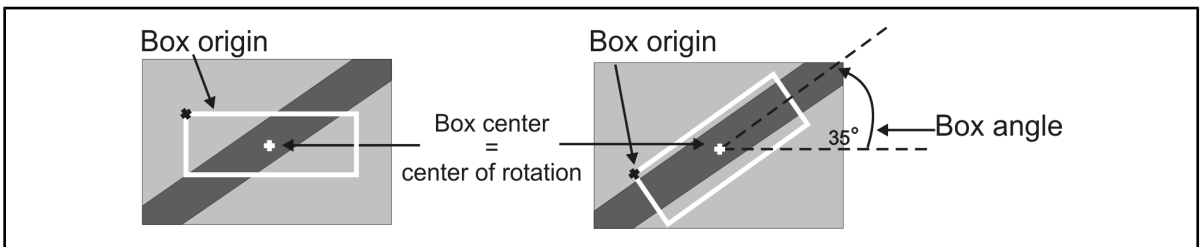
The orientation (**M_ORIENTATION**) specifies the expected direction of the marker relative to that of the measurement box and can be set to either **M_VERTICAL** or **M_HORIZONTAL**. The direction of the search is perpendicular to the orientation of the marker. If set to **M_ANY**, both orientations will be searched.

**Setting the measurement box angle**

The measurement operation 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. If the rotation is too great, the chance of not finding the marker or miscalculating characteristics, such as edge strength, and width, increases. In this case, the measurement box should be defined at 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°. 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()** with **M_BOX_ANGLE_REFERENCE**.



Multiple-angle search for a marker

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()` with `M_BOX_ANGLE_MODE`. Then, 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 approximate location and angle of the edge or stripe marker is first found by searching at every n degrees within the angular range, where n is specified by the 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.

During each step of the search, the Measurement module tries to find the marker with the highest score (`M_SCORE`); for an edge marker, this is the score of the edge, and for a stripe marker, this is the mean score of both edges.

- ❖ If searching within a range of angles causes the measurement box to be rotated beyond the edges of the target image, you must reposition or resize the measurement box unless subregions are used (discussed later).

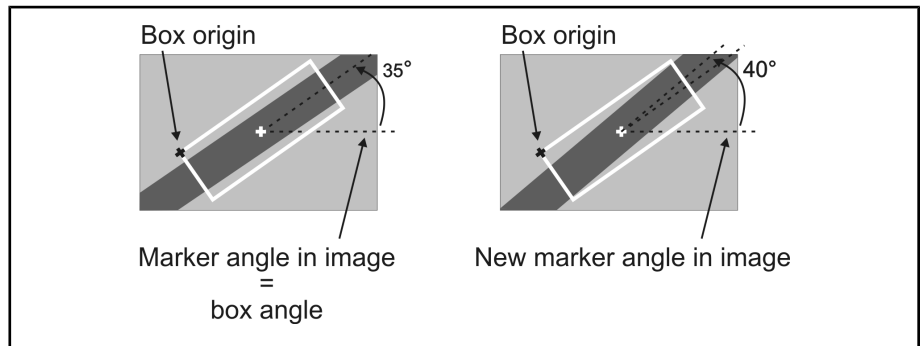
Determining the rotation tolerance of a marker

When performing a multiple-angle search for a marker, you should specify its rotation tolerance. The rotation tolerance is the amount by which the marker can be offset from the measurement box and still be found. 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 while maintaining the measurement box at a fixed position and angle. That is:

1. Make sure that the marker is at its expected (nominal) angle, and that the position and angle of the measurement box is set so that the marker is located within the measurement box at close to the same angle. The measurement box must remain

at a set position and angle. Therefore, multiple-angle search (**M_BOX_ANGLE_MODE**) must be disabled.

2. Use the **MimRotate()** function to rotate the image in very small increments (for example, 0.5°), in the positive direction, and perform a **MmeasFindMarker()** operation on each rotated image. 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.



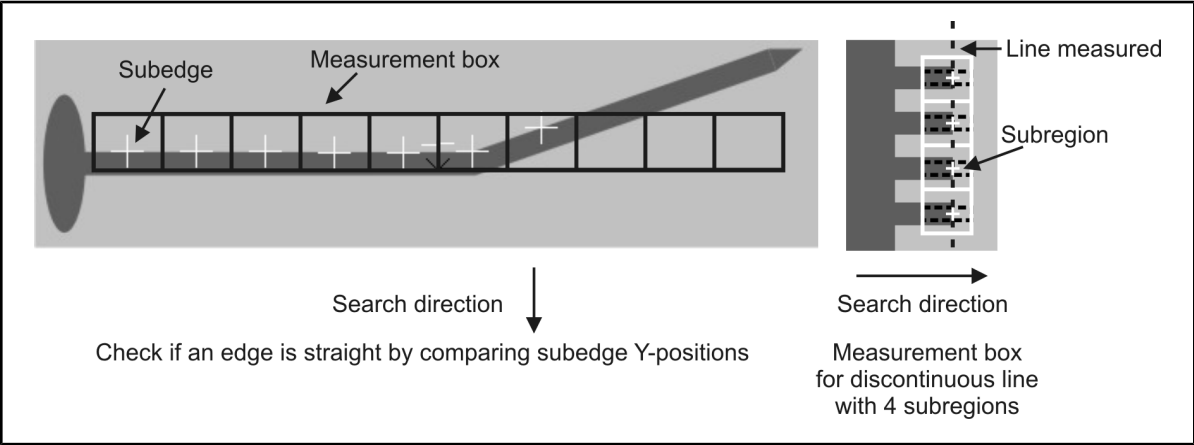
3. Check the results for the greatest angle that produces an acceptable score.
4. Repeat steps 2 and 3, rotating the image in the negative direction.
5. Take the minimum of the absolute value of these angles. Double this angle and set it as the rotation tolerance for the angular search.

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 processing time will be greatly increased when using this technique.

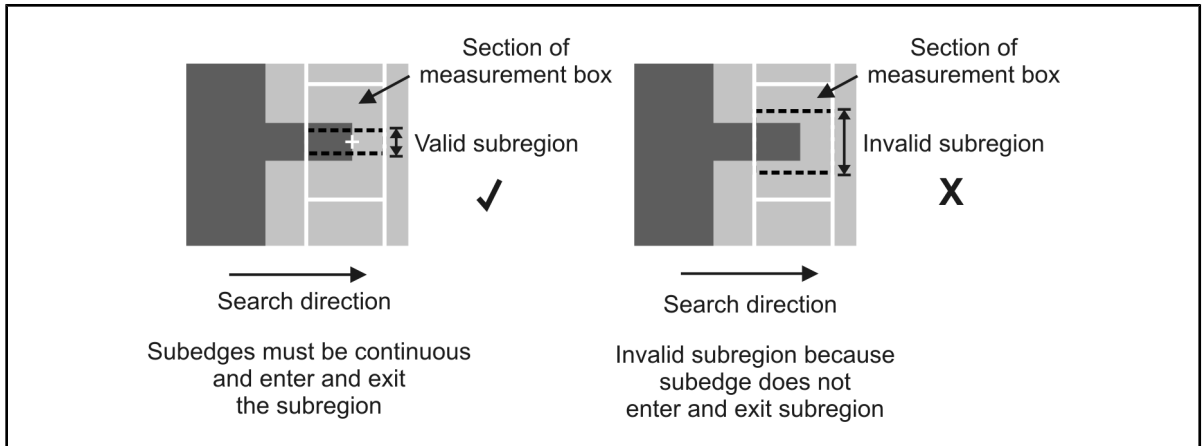
Subregions of the measurement box

You can divide the measurement box into sections and have the module process subregions of these sections. By default, subregions occupy the same area as their section, but can be reduced in size.

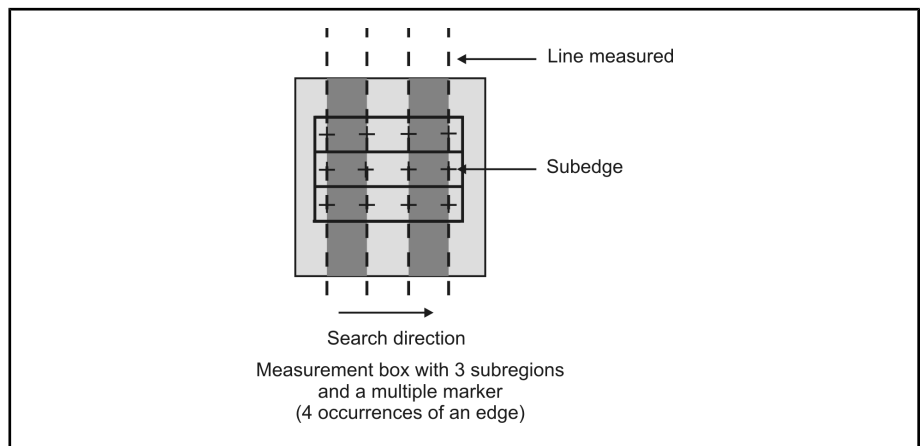
Subregions allow you to make more complex measurements. For example, you can use the information from the subregions to check if an edge is straight. In addition, you can find the line equation for a discontinuous edge that is disconnected at regular intervals.



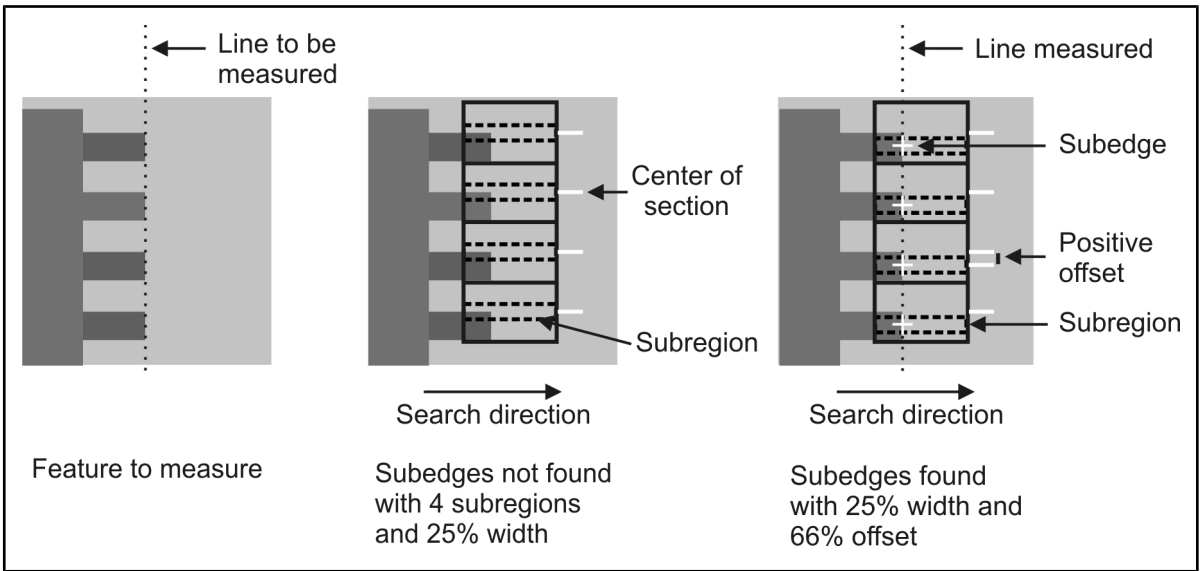
Each subregion acts like an independent measurement box; only subregions are searched for markers. To be found, the edge within the subregion (subedge) must be continuous and must enter and exit the entire subregion in the direction perpendicular to the search.



Subregions can be used with edge markers or stripe markers. In addition, they can be used to find single or multiple markers. For example, the following image shows a measurement box with three subregions and four occurrences of an edge marker.



Use **MmeasSetMarker()** to specify the number of sections and the corresponding area of the subregions. To divide the measurement box into sections, use **MmeasSetMarker()** with **M_SUB_REGIONS_NUMBER**. To narrow the subregion in each section, use **MmeasSetMarker()** with **M_SUB_REGIONS_SIZE**; specify the width of each subregion as a percentage of its section. By default, the subregion will be centered in each section. To move the subregion, use **MmeasSetMarker()** with **M_SUB_REGIONS_OFFSET** and specify the required displacement as a percentage of the maximum possible offset within a section. Depending on orientation, positive values will shift the subregion to the right or down and negative values will shift to the left or up.



- ❖ If subregions are used, only a minimum of two subregions need to be inside the target image for the search operation to succeed. This allows more flexibility in measurement box placement when searching through an angular range, because a small part of the measurement box is allowed to leave the target image.

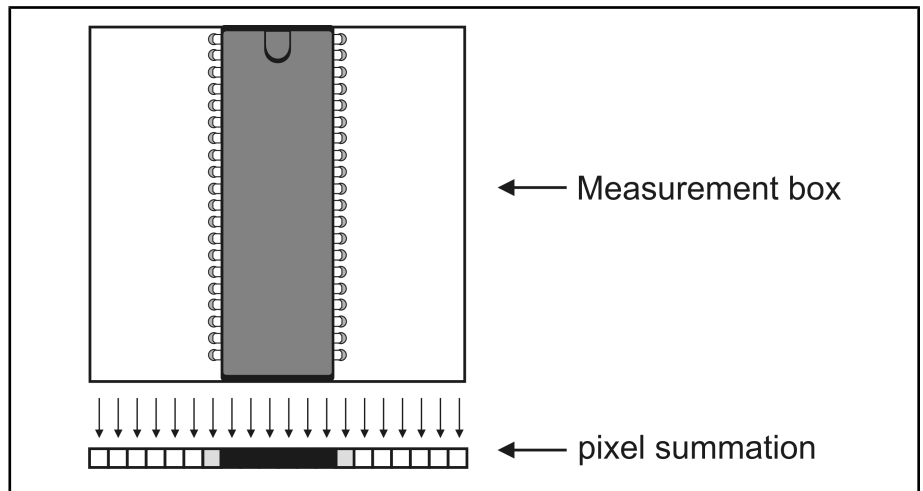
Search algorithm

This section describes the search algorithm used to identify edges in the Measurement module.

Identifying edges

The MIL Measurement module performs the following search algorithm to identify edges:

1. The MIL measurement search algorithm projects the pixels bounded by the specified two-dimensional measurement box into a one-dimensional pixel summation, storing the result in an internal projection buffer (that is, it takes the box's profile). The pixel summation is performed horizontally or vertically, depending on the measurement box's origin (**M_BOX_ORIGIN**) and the orientation of the marker (**M_ORIENTATION**). Each sum represents the intensity of all the pixels in that column.



2. The search algorithm applies a first derivative filter to the internal projection buffer (pixel summation). The filter finds the edge value of each projection value. The greater the difference in neighboring projection values, the larger the edge value. Edge values are represented as a normalized percentage of the maximum pixel value possible for the specific image buffer. Filters are discussed later in more detail.
3. The search algorithm then identifies possible edges by their edge strength (**M_EDGE_STRENGTH**), that is their maximum/minimum edge value (depending on the polarity of the edge represented by the sign of the edge value). It rejects as possible marker occurrences any edge with an edge strength below the edge threshold value (**M_EDGE_THRESHOLD**).
4. It then scores each possible edge based on geometric constraints that you specify. You can assign a specific weight (degree of importance) to each specified characteristic using **MmeasSetMarker()** with the **M_WEIGHT_FACTOR** combination constant. By default, the edge strength (**M_EDGE_STRENGTH**) receives the most importance. For more information on weights, see the *Edge markers: advanced characteristics* section later in this chapter. The edge(s) with the highest score is returned as an occurrence of the marker.

First derivative filter

The first derivative filter extracts the difference in neighboring projection values. The edge value calculated for each projection value of the measurement box is dependent on the type of the first derivative filter used. In the MIL Measurement module, there are two types of first derivative filters which can perform the edge extraction: Finite Impulse Response (FIR) filters and Infinite Impulse Response (IIR) filters. FIR filters are implemented using a non-recursive convolution operation with a predefined kernel of the specified size. To use a FIR filter, set **MmeasSetMarker()** **M_FILTER_TYPE** to **M_EULER** or **M_PREWITT**. IIR filters are implemented using a recursive convolution operation, and the kernel size would be theoretically infinite if this filter actually used a kernel. To use an IIR filter, set **MmeasSetMarker()** **M_FILTER_TYPE** to **M_SHEN**, and specify the degree of smoothness (strength of denoising) applied to the internal projection buffer of the measurement box, using **MmeasSetMarker()** with **M_FILTER_SMOOTHNESS**.

You can try these filters to see which best suits your application needs. The following is the general characteristics of each filter:

- **[-1,1] (M_EULER)**. This is a FIR filter with a convolution kernel of size 2. This filter is faster but more sensitive to noise, compared to the **M_PREWITT** filter.
- **[-1,0,1] (M_PREWITT)**. This is a FIR filter with a convolution kernel of size 3. This filter is slower but less sensitive to noise, compared to the **M_EULER** filter.
- **An IIR filter (M_SHEN)**. This is a Shen-Castan Infinite Support Exponential filter. This is an exponential weighting function, of the general form:

$$Ke^{-\beta|n|}$$

This IIR filter is slower than FIR filters (**M_EULER** or **M_PREWITT**). However, this filter is less sensitive to noise and provides more accurate results. You can also control the strength of denoising.

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()** function description for the default values). These values can be modified at any time, using **MmeasSetMarker()**.

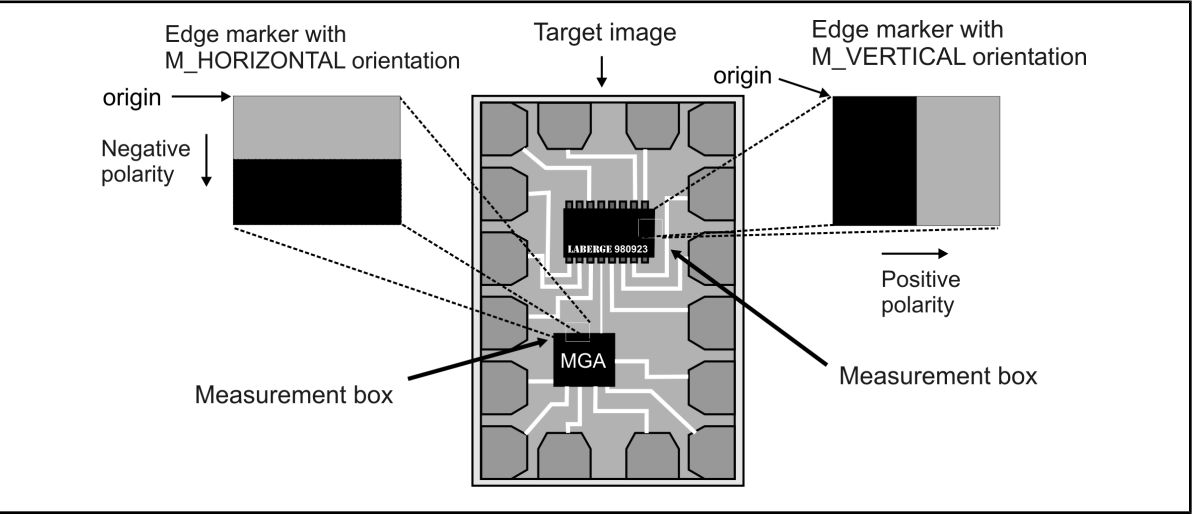
The important characteristics to set when searching for a marker are the measurement box, the polarity, the contrast, and for stripe markers, the width. Fundamental and advanced characteristics are discussed for edge, stripe, and multiple markers in the following sections. Note, all characteristics can be set to **M_ANY** if the value is unknown or not a criteria, unless otherwise specified in the MIL Reference.

Edge markers: fundamental characteristics

This section describes the fundamental characteristics of edge markers which are set using `MmeasSetMarker()`, or obtained as measurement results using `MmeasFindMarker()`.

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. For more information, see the *Measurement box* section earlier in this chapter.



Position and position variation

The marker's position is defined as the X- and Y-coordinates of the marker's center pixel (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 between two markers.

When several edges have similar characteristics within the same measurement box, then you can use the position characteristic to specify the approximate X- and/or Y-coordinates (**M_POSITION**, **M_POSITION_X**, **M_POSITION_Y**) at which to find the required marker's center.

You can also specify a tolerance for these coordinates (**M_POSITION_VARIATION**).

The position must be located within the measurement box (taking into account the measurement box's angle or angular range), otherwise an error is generated.

Contrast and contrast variation

You can indicate the typical difference in average grayscale values between the start and end of the intensity transition (edge width) from which an edge is established. (**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. When the edge strength is not very high, you should specify the contrast. 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.

You can also specify a tolerance for the contrast (**M_CONTRAST_VARIATION**).

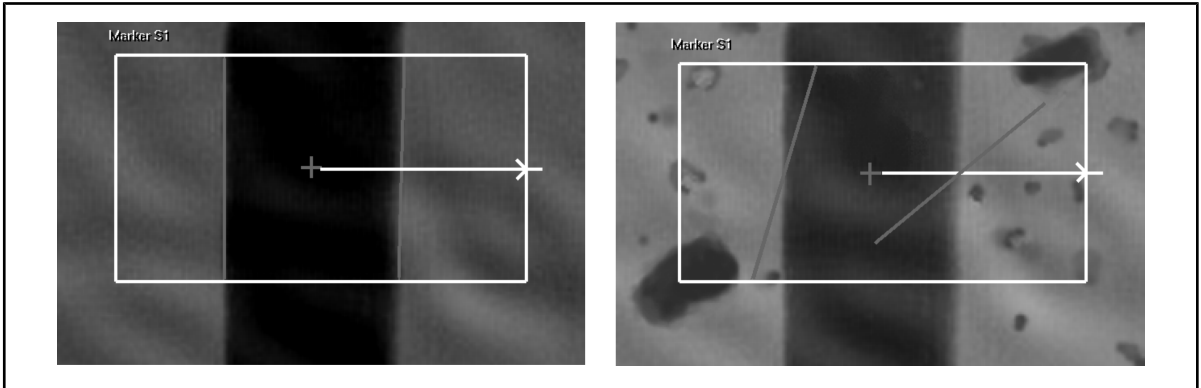
Length

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

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. If the measurement was performed using subregions, the line returned is the line that best passes through the centers of all the subedges found.

If only one subregion is used, MIL performs additional internal search operations in the upper and lower thirds of the measurement box to find the equation of the line. If an image has substantial specks that appear in either of these two areas, the calculated lines might be at incorrect angles, as illustrated in the following images.



Notice the edge results on the left compared to the results on the right. Often, raising the edge threshold (using **MmeasSetMarker()** with **M_EDGE_THRESHOLD**) will correct this type of error. Also, you should always verify the edge values using **MmeasSetMarker()** with **M_BOX_EDGE_VALUES**.

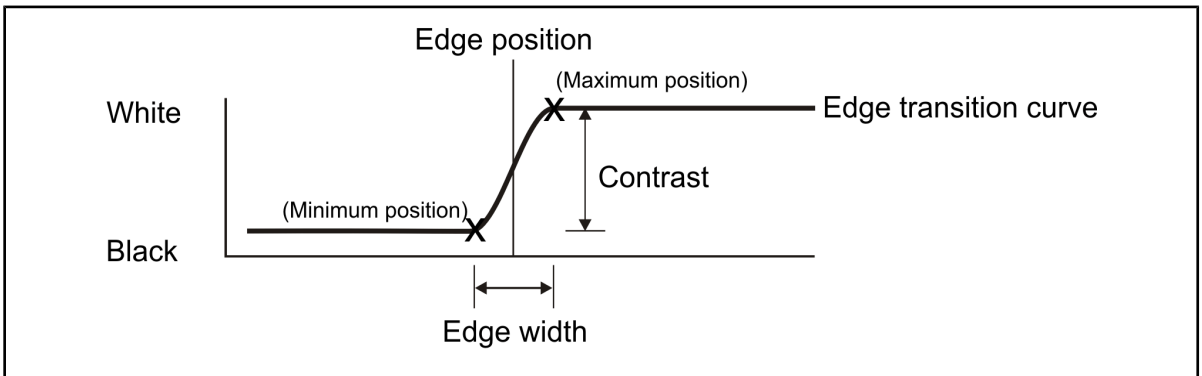
Note, the line equation of an edge marker cannot be set, but can be returned with **MmeasGetResult()**. The slope of the line equation (**M_LINE_EQUATION_SLOPE**) and the Y-intercept of the line equation (**M_LINE_EQUATION_INTERCEPT**) can also be returned separately.

Edge markers: advanced characteristics

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

Edge width

In an image, edges are curves that delineate a boundary, which can be established from intensity transitions. The smoother the image, the more gradual the transition. The edge width can be seen as a measure, in pixels, of this gradual transition in grayscale values. The diagram below illustrates a profile of gradual transition from dark to light.



The Measurement module calculates the edge's position to be in the middle of this edge width. The position variation is equivalent to half the edge width. Note, the more an edge is at an angle, the greater the stretching or distortion of its edge width.

The edge width of a 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 minimum and maximum positions of an edge in an edge marker.

The X-coordinate of these positions are the minimum/maximum position within the edge width from which an edge is extracted, respectively. The Y-coordinate is the Y-coordinate of the measurement box center. The minimum position is always on the side of the measurement box adjacent to the origin, and the maximum position is always on the side of the measurement box furthest from the 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, **MmeasFindMarker()**.

Edge strength

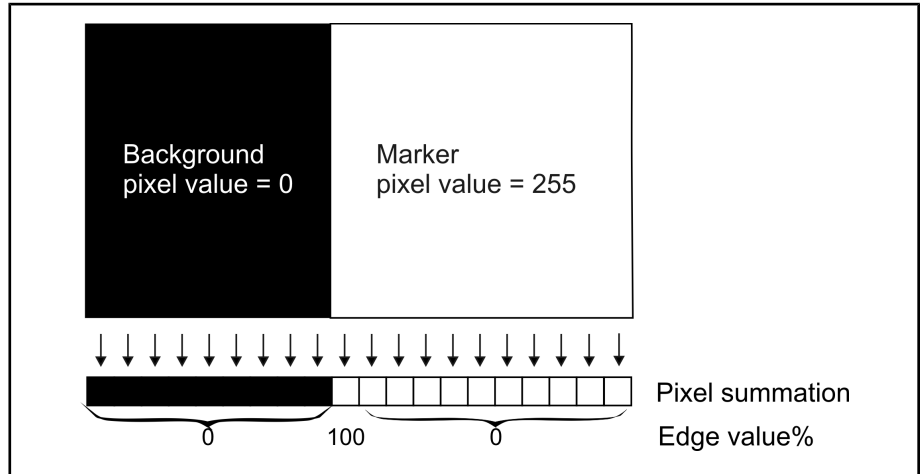
A marker's edge strength (**M_EDGE_STRENGTH**) is either the minimum or maximum edge value along the edge width, depending on the polarity of the edge. Edge values are the output of the first derivative filter (that is applied to the internal projection buffer) represented as a normalized percentage 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:

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

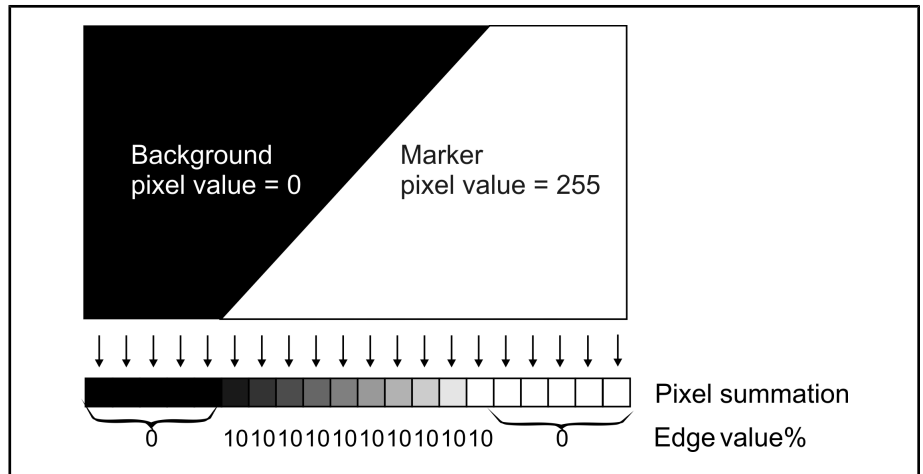
You can also specify a tolerance for the edge strength (**M_EDGE_STRENGTH_VARIATION**).

- ❖ For the following diagrams, edge values are calculated using an Euler FIR filter, [-1,1]. Using different filters will result in different edge values. For more information on identifying edges using first derivative filters, see the *Search algorithm* section earlier in this chapter.

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 (**M_CONTRAST**). By using the contrast characteristic, you can indicate the typical difference in average grayscale values between the start and end of the intensity transition (edge width) from which an edge is established, allowing the marker to be located. Contrast is useful in distinguishing between several different edges, particularly when the required edge does not have the largest edge strength, or when the edge is at an angle. 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.

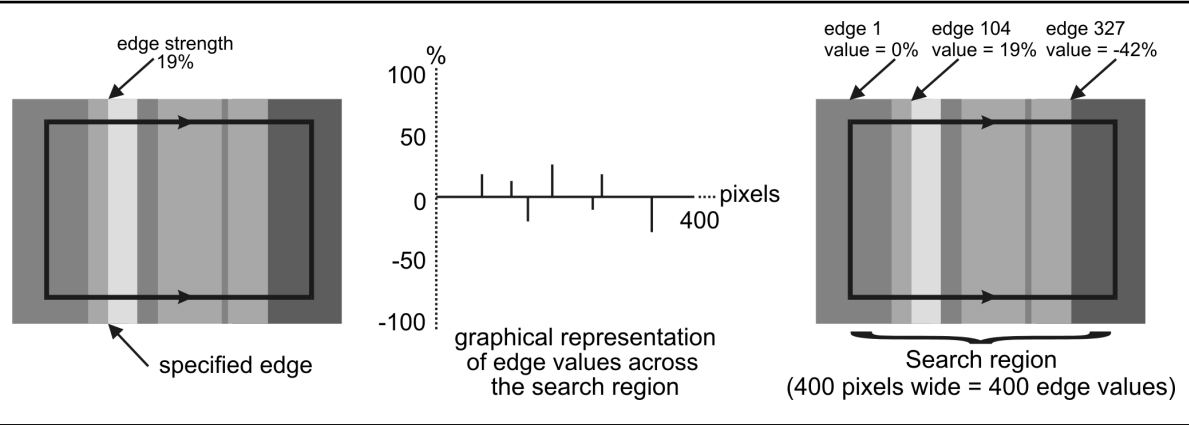
Edge threshold

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 **MmeasGetResult()** with **M_BOX_EDGE_VALUES**. This returns the calculated edge value for every projection value of the measurement box. This allows you to determine what is likely to be considered an edge and specify an appropriate edge threshold value.

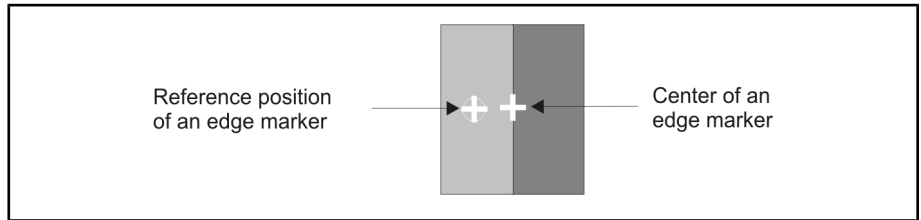
The illustration below shows a specific edge strength measurement, a graphical representation of edge values across the measurement box, and a sampling of these results.



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 measurements 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 center left of the marker.



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.

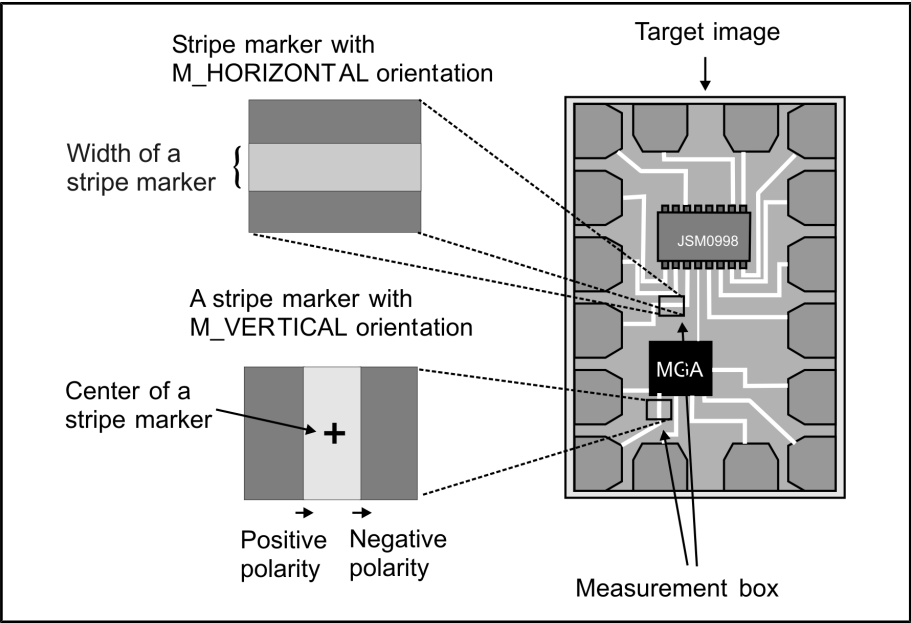
You can override this default by adding **M_WEIGHT_FACTOR** to certain of the marker characteristics (see **MmeasSetMarker()** in the MIL Reference manual for the list of applicable characteristics). However, to better control the search, it is recommended that when specifying weight factors, that you 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 expected 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

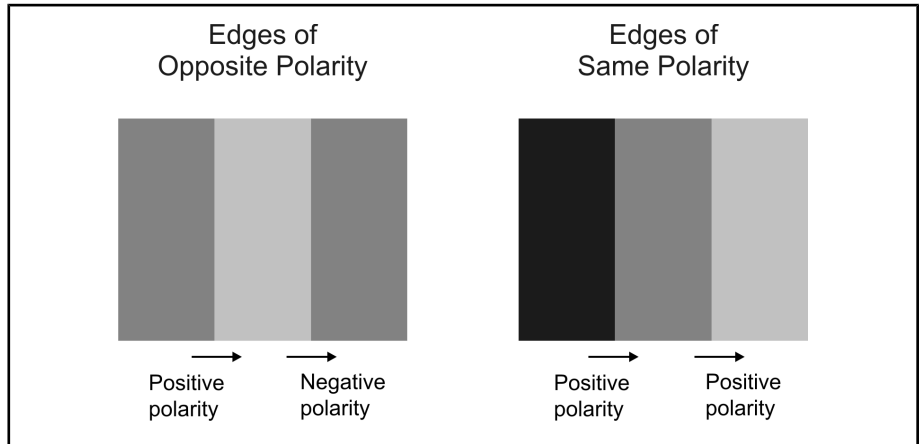
A stripe marker is simply a marker with two edges, therefore the discussion of edge characteristics applies to each of the stripe marker's edges. However, certain characteristics have special attributes applicable only to stripe markers.

As with edge markers, the characteristics of stripe markers are set using **MmeasSetMarker()**, or obtained as measurement results using **MmeasFindMarker()**.



Polarity

The polarity of an edge describes whether an edge is rising or falling. The edges of a stripe marker can have opposite (**M_OPPOSITE**) polarities or can have similar (**M_SAME**) polarities.



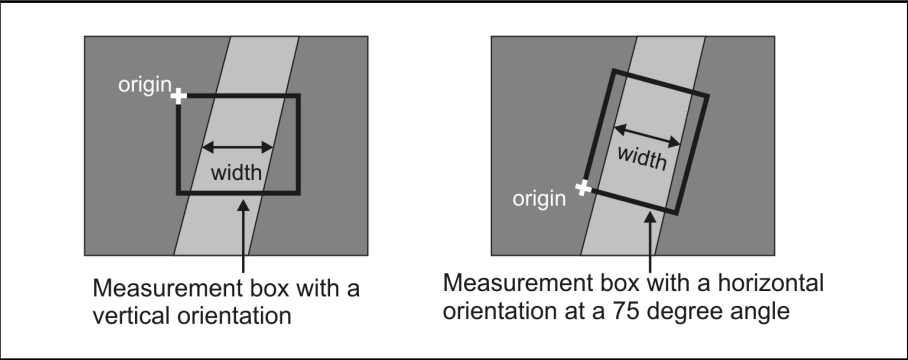
Contrast and contrast variation

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.

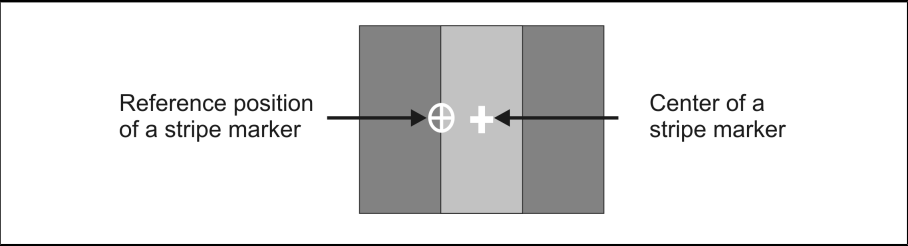
The width of a stripe marker is the average distance in pixels between its edges. Note, if the marker's measurement box (processing region) is at an angle, the width is measured according to the orientation of the box, as shown below.



Position

The position (**M_POSITION**) of a stripe marker is considered either the center between the two edges of the stripe or a position outside of the stripe marker, depending on the **M_POSITION_INSIDE_STRIPE** setting.

In addition, the stripe marker's center is the default reference position. The reference position is specified relative to the marker's center. You can move the reference position of a marker using **MmeasSetMarker()** with **M_MARKER_REFERENCE**.

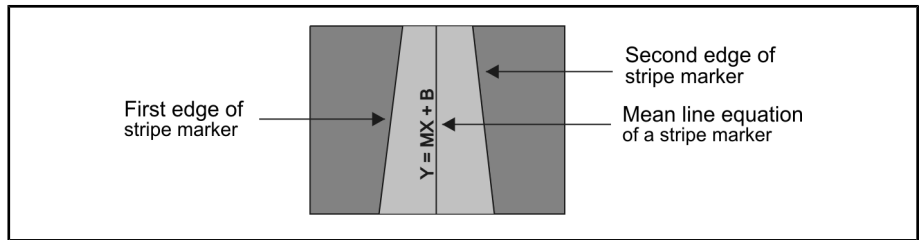


Length

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

In addition to calculating the line equation of each edge, you can calculate the equation of the mean line following a stripe marker: $Y = MX + B$, where M denotes the slope of the line and B denotes the Y-intercept. Essentially, this is the mean of the lines calculated for each edge of a stripe marker.



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

This section deals with advanced characteristics with stripe markers.

Minimum and maximum position

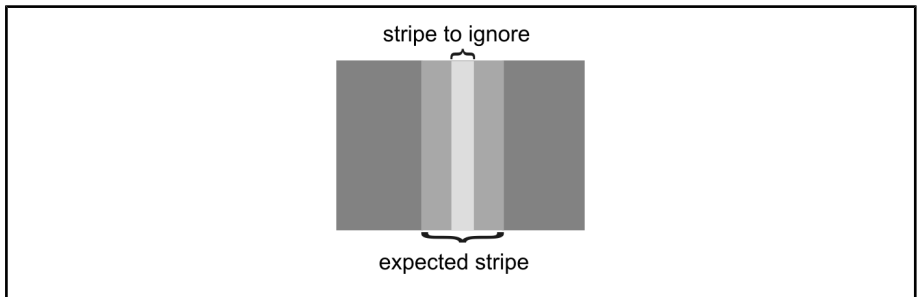
The minimum and maximum positions (**M_POSITION_MIN** and **M_POSITION_MAX**) indicate the minimum and maximum positions of edges in a stripe marker. For more information on the minimum and maximum position of an edge, see the *Edge markers: advanced characteristics* section earlier in this chapter.

The minimum and maximum positions of edges cannot be set but can be found with **MmeasFindMarker()**.

The average value of the two edges of a stripe marker is returned unless otherwise specified (using **MmeasGetResult()** with **M_EDGE_FIRST** or **M_EDGE_SECOND**).

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.

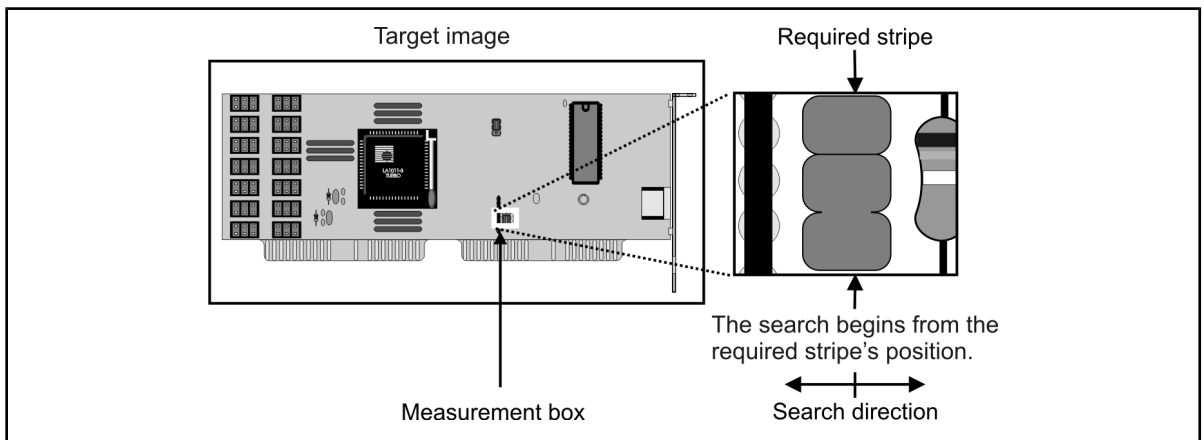


To find the larger stripe without having to determine and specify its width, specify 2 as the number of inside edges of the stripe marker (**M_EDGE_INSIDE**). To find the smaller stripe, specify 0 (the default, **M_ANY**, ignores the possibility of any inside edges). The identification of inside edges 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.

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

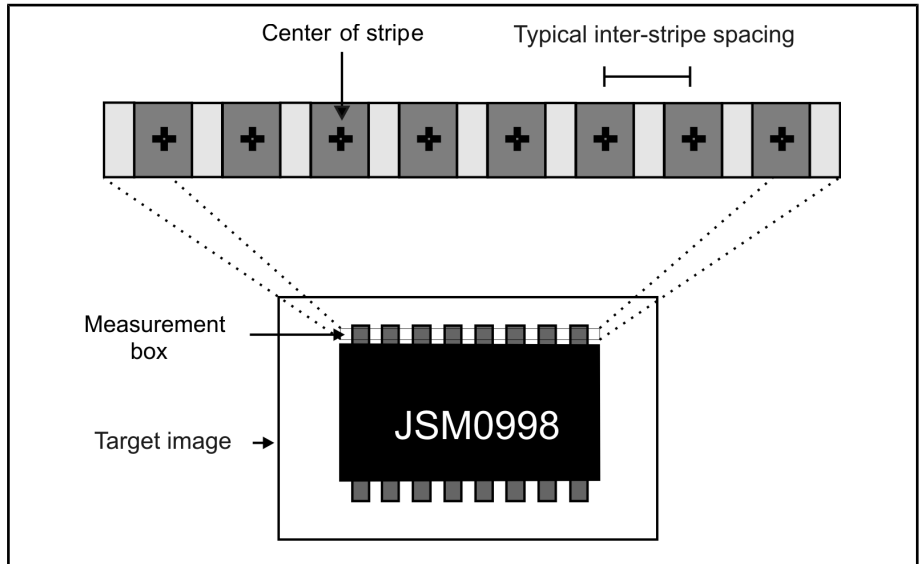
If necessary when defining a stripe marker, you can specify if the X- and Y-coordinates of **M_POSITION** must be located inside or outside of the stripe (**M_POSITION_INSIDE_STRIPE**). If the position is defined as being within the stripe, the search algorithm is modified so that the search proceeds 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's 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()**.

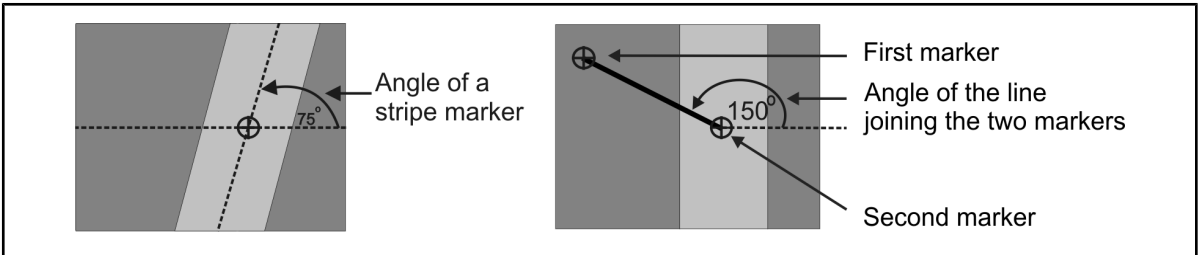
Calculating with multiple markers

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.

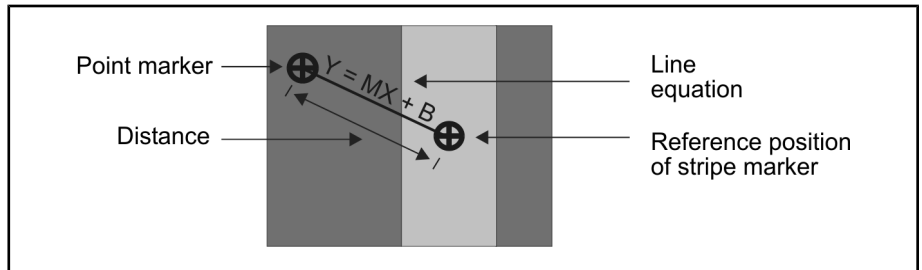


The angle is measured in a counter-clockwise direction relative to the positive X-axis, and can be a value from 0 to 360°.

Line equation and distance

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

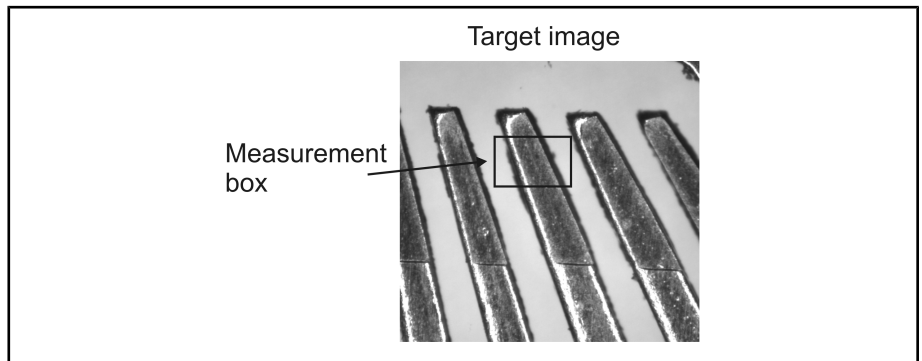
The distance between two markers is the distance between both of their reference positions, as illustrated in the diagram below.



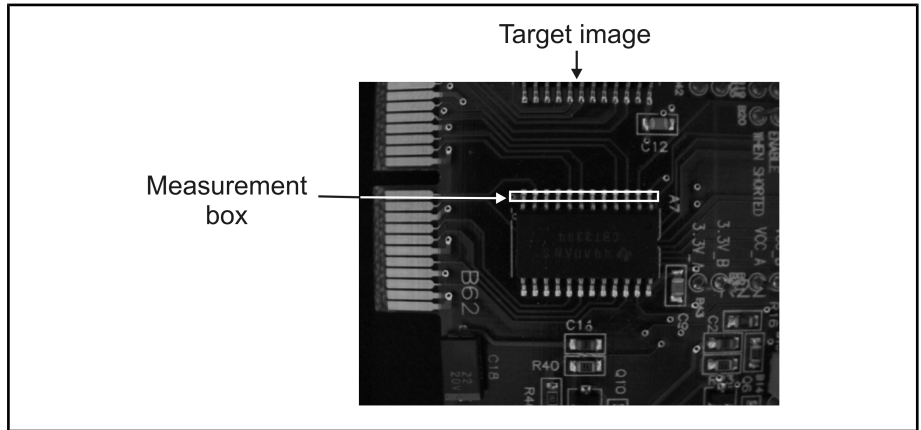
The horizontal or vertical distance between two markers can also be calculated by using `M_DISTANCE_X` or `M_DISTANCE_Y` as the result type with `MmeasGetResult()`.

Measurement examples

The Measurement example *MMeas.cpp* demonstrates how to find a stripe in an image and measure its position, width, and angle.



This example also demonstrates how to find the average position, average width, and average angle of a row of pins on a chip.



The example also shows you how to perform drawing operations and return measurement statistics.

```

/*****
/*
* File name: MMeas.cpp
*
* Synopsis: This program consist of 2 examples that use the Measurement module
*           to calculate the position, width and angle of objects in an image.
*           The first one measures the position, width and angle of a stripe
*           in an image, and marks its center and edges. The second one measures
*           the average position, width and angle of a row of pins on a chip.
*
*/
#include <mil.h>

/* Example selection. */
#define RUN_SINGLE_MEASUREMENT_EXAMPLE      M_YES
#define RUN_MULTIPLE_MEASUREMENT_EXAMPLE    M_YES

/* Example functions declarations. */
void SingleMeasurementExample(MIL_ID MilSystem, MIL_ID MilDisplay);
void MultipleMeasurementExample(MIL_ID MilSystem, MIL_ID MilDisplay);

/*****
Main.
*****/
int MosMain(void)

```



```

{
    MIL_ID MilApplication,      /* Application identifier. */
        MilSystem,            /* System Identifier.      */
        MilDisplay;           /* Display identifier.     */

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

    /* Print module name. */
    MosPrintf(MIL_TEXT("\nMEASUREMENT MODULE:\n"));
    MosPrintf(MIL_TEXT("-----\n\n"));

    #if (RUN_SINGLE_MEASUREMENT_EXAMPLE)
    SingleMeasurementExample(MilSystem, MilDisplay);
    #endif

    #if (RUN_MULTIPLE_MEASUREMENT_EXAMPLE)
    MultipleMeasurementExample(MilSystem, MilDisplay);
    #endif

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

    return 0;
}

/*****
Single measurement example.
*****/

/* Source MIL image file specification. */
#define MEAS_IMAGE_FILE      M_IMAGE_PATH MIL_TEXT("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

void SingleMeasurementExample(MIL_ID MilSystem, MIL_ID MilDisplay)
{
    MIL_ID      MilImage,                /* Image buffer identifier. */
        MilOverlayImage,                /* Overlay image.           */
        StripeMarker;                   /* Stripe marker identifier. */
    MIL_DOUBLE  StripeCenterX,           /* Stripe X center position. */
        StripeCenterY,                 /* Stripe Y center position. */

```

```

        StripeWidth,                /* Stripe width.          */
        StripeAngle;               /* Stripe angle.          */
MIL_DOUBLE CrossColor = M_COLOR_YELLOW, /* Cross drawing color.    */
BoxColor = M_COLOR_RED;           /* Box drawing color.      */

/* Restore and display the source image. */
MbufRestore(MEAS_IMAGE_FILE, MilSystem, &MilImage);
MdispSelect(MilDisplay, MilImage);

/* Prepare for overlay annotation. */
MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

/* Allocate a stripe marker. */
MmeasAllocMarker(MilSystem, M_STRIPE, M_DEFAULT, &StripeMarker);

/* Specify the stripe characteristics. */
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 and position. */
MmeasSetMarker(StripeMarker, M_BOX_ORIGIN, MEAS_BOX_POS_X, MEAS_BOX_POS_Y);
MmeasSetMarker(StripeMarker, M_BOX_SIZE, MEAS_BOX_WIDTH, MEAS_BOX_HEIGHT);

/* Draw the contour of the measurement box. */
MgraColor(M_DEFAULT, BoxColor);
MmeasDraw(M_DEFAULT, StripeMarker, MilOverlayImage, M_DRAW_BOX, M_DEFAULT, M_MARKER);

/* Pause to show the original image. */
MosPrintf(MIL_TEXT("Position, width and angle of the stripe ")
        MIL_TEXT("in the highlighted box\n"));
MosPrintf(MIL_TEXT("will be calculated and the center ");
MosPrintf(MIL_TEXT("and edges will be marked.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Clear the overlay image. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

/* 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_WIDTH, &StripeWidth, M_NULL);
MmeasGetResult(StripeMarker, M_ANGLE, &StripeAngle, M_NULL);

/* Draw the contour of the found measurement box. */
MgraColor(M_DEFAULT, BoxColor);
MmeasDraw(M_DEFAULT, StripeMarker, MilOverlayImage, M_DRAW_BOX, M_DEFAULT, M_RESULT);

```

```

/* Draw a cross on the center, left edge and right edge of the found stripe. */
MgraColor(M_DEFAULT, CrossColor);
MmeasDraw(M_DEFAULT, StripeMarker, MilOverlayImage, M_DRAW_POSITION, M_DEFAULT,
                                                  M_RESULT);

MmeasDraw(M_DEFAULT, StripeMarker, MilOverlayImage,
          M_DRAW_POSITION+M_EDGE_FIRST+M_EDGE_SECOND, M_DEFAULT, M_RESULT);

/* Print the result. */
MosPrintf(MIL_TEXT("The stripe in the image is at position %.2f,%.2f and\n"),
          StripeCenterX, StripeCenterY);
MosPrintf(MIL_TEXT("is %.2f pixels wide with an angle of %.2f degrees.\n"),
          StripeWidth, StripeAngle);
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Clear the overlay image. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

/* Free all allocations. */
MmeasFree(StripeMarker);
MbufFree(MilImage);
}

/*****
Multiple measurement example.
*****/

/* Source MIL image file specification. */
#define MULT_MEAS_IMAGE_FILE      M_IMAGE_PATH MIL_TEXT("chip.mim")

/* Processing region specification. */
#define MULT_MEAS_BOX_WIDTH      230
#define MULT_MEAS_BOX_HEIGHT     7
#define MULT_MEAS_BOX_POS_X      220
#define MULT_MEAS_BOX_POS_Y      171

/* Target stripe specifications. */
#define MULT_STRIPES_ORIENTATION  M_VERTICAL
#define MULT_STRIPES_POLARITY_LEFT M_POSITIVE
#define MULT_STRIPES_POLARITY_RIGHT M_NEGATIVE
#define MULT_STRIPES_NUMBER       12

void MultipleMeasurementExample(MIL_ID MilSystem, MIL_ID MilDisplay)
{
    MIL_ID      MilImage,           /* Image buffer identifier. */
               MilOverlayImage,     /* Overlay image.           */
               StripeMarker;        /* Stripe marker identifier. */
    MIL_DOUBLE  MeanAngle,          /* Stripe mean angle.       */
               MeanWidth,           /* Stripe mean width.       */
               MeanSpacing;         /* Stripe mean spacing.     */
    MIL_DOUBLE  CrossColor = M_COLOR_YELLOW, /* Cross drawing color.    */
               BoxColor   = M_COLOR_RED;    /* Box drawing color.      */

```

```

/* Restore and display the source image. */
MbufRestore(MULT_MEAS_IMAGE_FILE, MilSystem, &MilImage);
MdispSelect(MilDisplay, MilImage);

/* Prepare for overlay annotation. */
MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);
MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

/* Allocate a stripe marker. */
MmeasAllocMarker(MilSystem, M_STRIPE, M_DEFAULT, &StripeMarker);

/* Specify the stripe characteristics. */
MmeasSetMarker(StripeMarker, M_NUMBER, MULT_STRIPES_NUMBER, M_NULL);
MmeasSetMarker(StripeMarker, M_POLARITY, MULT_STRIPES_POLARITY_LEFT,
               MULT_STRIPES_POLARITY_RIGHT);
MmeasSetMarker(StripeMarker, M_ORIENTATION, MULT_STRIPES_ORIENTATION, M_NULL);

/* Specify the measurement box size and position. */
MmeasSetMarker(StripeMarker, M_BOX_ORIGIN, MULT_MEAS_BOX_POS_X, MULT_MEAS_BOX_POS_Y);
MmeasSetMarker(StripeMarker, M_BOX_SIZE, MULT_MEAS_BOX_WIDTH, MULT_MEAS_BOX_HEIGHT);

/* Draw the contour of the measurement box. */
MgraColor(M_DEFAULT, BoxColor);
MmeasDraw(M_DEFAULT, StripeMarker, MilOverlayImage, M_DRAW_BOX, M_DEFAULT, M_MARKER);

/* Pause to show the original image. */
MosPrintf(MIL_TEXT("The position and angle of a row of pins on a chip ")
          MIL_TEXT("will be calculated.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Find the stripe and measure its width and angle. */
MmeasFindMarker(M_DEFAULT, MilImage, StripeMarker, M_POSITION + M_ANGLE + M_WIDTH);

/* Draw the contour of the measurement box. */
MgraColor(M_DEFAULT, BoxColor);
MmeasDraw(M_DEFAULT, StripeMarker, MilOverlayImage, M_DRAW_BOX, M_DEFAULT, M_RESULT);

/* Draw a cross at the center of each stripe found. */
MgraColor(M_DEFAULT, CrossColor);
MmeasDraw(M_DEFAULT, StripeMarker, MilOverlayImage, M_DRAW_POSITION, M_ALL, M_RESULT);

/* Get the stripe's width, angle and spacing. */
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);

/* Print the results. */
MosPrintf(MIL_TEXT("The center and angle of each pin have been marked.\n\n"));
MosPrintf(MIL_TEXT("The statistics for the pins are:\n"));
MosPrintf(MIL_TEXT("Average angle   : %5.2f\n"), MeanAngle);

```

```
MosPrintf(MIL_TEXT("Average width   : %5.2f\n"), MeanWidth);
MosPrintf(MIL_TEXT("Average spacing : %5.2f\n\n"), MeanSpacing);
MosPrintf(MIL_TEXT("Press <Enter> to end.\n"));
MosGetch();

/* Clear the overlay image. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

/* Free all allocations. */
MmeasFree(StripeMarker);
MbufFree(MilImage);
}
```


Chapter

15

Metrology

This chapter explains how to use the MIL Metrology module to measure features and validate geometric tolerances in a target image.

MIL Metrology module

Many industries require the ability to measure and validate objects to, for example, assemble multiple parts. Historically, in such industries, manual quality control applications were created to verify the shape, size, and location of objects relative to one another. To accomplish this, a template was used. A template is a physical part, or mold, that typically contains holes and incisions at various positions. If the object being verified physically fit into the template, then that object had passed the quality inspection and could be used in assembly; otherwise, it had failed and would be discarded or reworked. The MIL Metrology module is a powerful set of functions that allows you to design such measurement and validation applications, using a virtual equivalent of a physical template.

The MIL metrology template represents a precisely measured and reusable model from which objects can be verified and measured. To build a metrology template, you must have features (which are like the holes and incisions) and geometric tolerances (which are like the positions that features must have, relative to other features).

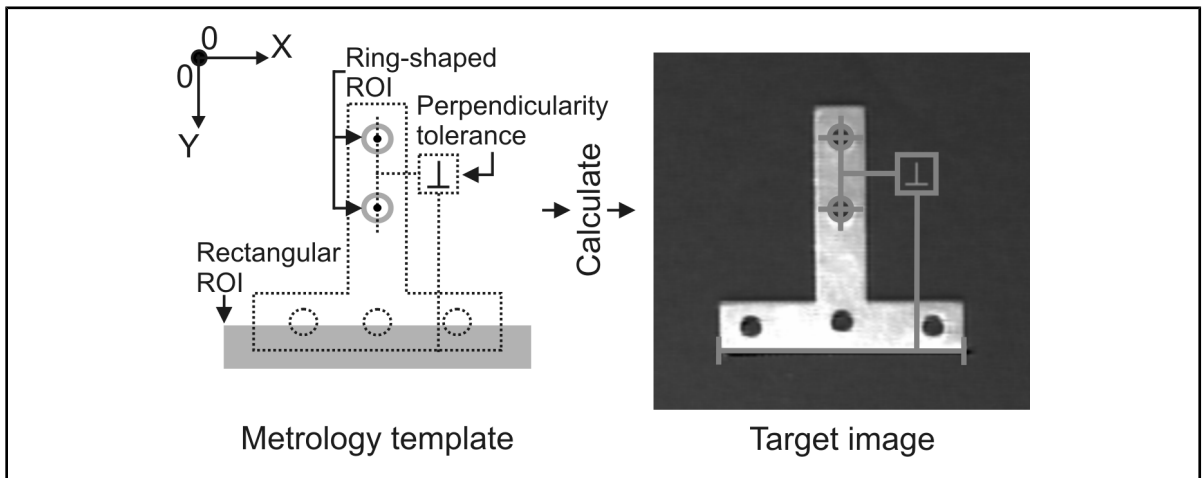
You can add either physically measured or constructed features to your metrology template. Physically measured features are geometric primitives (such as a circle) that are physically measured using the edge information (edgels) in the target image. Physically measured features, and their region of edgel extraction in the target image (ROI), must be placed at the appropriate location in the template.

Note that Metrology's edgel extraction is based on Edge Finder technology; for more information, see the *Extracting the edges* section in *Chapter 9: Edge Finder*.

Constructed features are geometric primitives that do not actually exist in the target image, they are mathematically defined (parametrically constructed features) or geometrically derived from a set of other features (either constructed or physically measured). For example, a constructed point feature can be derived at the center of a physically measured circle.

Features alone do not typically provide enough information to establish the metrology template. For example, if you add two segment features, you might also want to verify their total length, or if they are perpendicular. To do this, you can specify the geometric tolerances that the features of the object in the target image must respect to be considered valid.

The following example illustrates how you can use features and geometric tolerances in a metrology template to represent an appropriate object in a target image. To depict this object, the template has two ring-shaped ROIs where two circles are expected to be physically measured, and one rectangular ROI where one segment is expected to be physically measured. The template also contains the definition of a constructed segment, built from the center points of the two physically measured circles, and a perpendicularity tolerance between this constructed segment and the physically measured segment. The following example also shows the target image, which has been annotated to display the successfully measured features and the validated tolerance.



The MIL Metrology module allows you to specify the coordinates of features and feature ROIs relative to a reference frame, which can be set to either the global frame or a local frame.

The global frame is a coordinate system whose origin is at the top-left corner of the metrology template (by default, aligned with the top-left corner of the target image); the global frame is created automatically. A local frame is a coordinate system that can be located anywhere within the metrology template; you can create a local frame at a specified position or based on the positions of other features in the template. During calculation, the reference frame can be repositioned.

The MIL Metrology module provides complete support for calibration. If the target image is calibrated, calculations are performed in the calibrated real-world such that, without physically correcting your images, measurements and calculations are made even in the presence of distortion, and results calculated in real-world units.

Note that, when complex distortions exist, some results are meaningless when converted from real-world to pixel units (for example, angle). You should therefore be cautious when using distorted images.

- ❖ If a calibration object has been associated to the target image, you must specify dimensions for features, geometric tolerances, regions, and positions in real-world units.

The MIL Metrology module also allows you to restore a metrology context from a file or memory stream, or save a metrology context to a file or memory stream.

Steps to measure and validate expected features

The following steps provide a basic methodology for using the MIL Metrology module:

1. Allocate a metrology context to store the features and tolerances of your metrology template and to store global processing settings, using **MmetAlloc()**. When you allocate a metrology context, the global reference frame is automatically created.
2. Allocate a metrology result buffer to hold the results of your calculations, using **MmetAllocResult()**.
3. Define the features of your metrology template. For each feature:
 - a. Add the feature to the template, using **MmetAddFeature()**. Typically, you should add physically measured features before constructed features.
 - b. For a physically measured feature, you must set the feature's ROI in the template, using **MmetSetRegion()**.
 - c. If necessary, adjust feature settings using successive calls to **MmetControl()**.
4. Define the geometric tolerances of your metrology template. For each tolerance:
 - a. Add the tolerance to the template, using **MmetAddTolerance()**.
 - b. If necessary, adjust tolerance settings using successive calls to **MmetControl()**.
5. If necessary, specify your required global processing settings, using successive calls to **MmetControl()**.
6. If necessary, reposition features (typically, a reference frame) in the template, depending on the location of the object in the target image, using **MmetSetPosition()**.
7. Calculate features and validate geometric tolerances for the object in the target image, using **MmetCalculate()**.

8. Retrieve the required results from the result buffer, using **MmetGetResult()**.
9. If necessary, draw specific metrology result features and geometric tolerances in an image buffer, using **MmetDraw()**.
10. If necessary, save your metrology context, using **MmetSave()**.
11. Free your metrology context and result buffer, using **MmetFree()**.

Basic concepts

The basic concepts and vocabulary conventions for the MIL Metrology module are:

- **Active edgels.** The edgels used to build the feature. These edgels are within the ROI and adhere to all constraints.
- **Constructed feature.** A geometric primitive that has been mathematically defined or geometrically derived from a set of other features.
- **Edge.** A curve that delineates a boundary, which can be established from intensity transitions in an image.
- **Edgel.** Elementary points (or edge elements) within an edge. Each edgel represents an X- and Y-position, and the angular direction of the gradient.
- **Geometric tolerance.** The acceptable deviation from the definition of a feature, or the acceptable relationship between multiple features.
- **Global frame.** The metrology template's default reference frame.
- **Gradient angle.** The angle between the target image's horizontal axis and the edge's perpendicular direction (from black to white) at each edgel location.

- **Local frame.** A constructed reference frame.
- **(Physically) measured feature.** A geometric primitive that can be physically measured using the edgels in the feature's ROI.
- **Metrology context.** The container for all features and geometric tolerances of the metrology template, and global processing settings.
- **Metrology template.** The set of all features and geometric tolerances.
- **Output frame.** The coordinate system in which feature results are returned.
- **Parametrically constructed feature.** A geometric primitive that has been mathematically defined, rather than geometrically derived from a set of other features.
- **Reference frame.** The coordinate system in which features are defined and to which they are relative.
- **Region orientation.** The alignment of an ROI, with respect to a reference frame.
- **ROI.** Region of interest. In metrology, this is the delimited area in the target image from which to select edgels that will be used to build the feature (for physically measured features).
- **Target image.** The image in which to extract physically measured features, build constructed features, and verify geometric tolerances.
- **Template reference.** An image (or result buffer) that represents an example of a target image, used to help build or modify a metrology template.

Features

Once you have allocated your metrology context using **MmetAlloc()**, you must then add features to the metrology template, using **MmetAddFeature()**. You can add both physically measured and constructed features.

The following table lists all available features, their definition, and whether they can be physically measured or constructed.

Feature	Definition	Physically measured	Constructed
Segment	Part of a geometric line that is defined by a start point and end point.	Yes	Yes
Circle	A closed curve of points where each point is located at the same distance from one center point.	Yes	Yes
Arc	A continuous portion of a circle.	Yes	Yes
Edgel	Elementary points (or edge elements) within an edge. Each edgel represents an X- and Y-position, and the angular direction of the gradient.	Yes	Yes
Point	A geometric primitive that represents an X- and Y-position.	Yes	Yes
Local frame	A constructed reference frame.	—	Yes
Line	A one-dimensional geometric primitive that is straight and infinitely long.	—	Yes

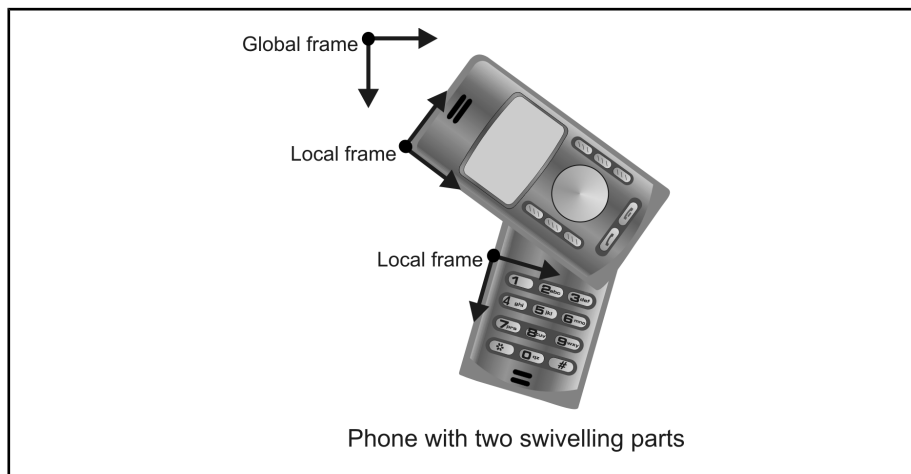
If you associate a calibration object to the target image, you must use real-world units to define or position features in the template. If there is no calibration object, use pixel units. For more information, see the *Calibration overview* section in *Chapter 5: Camera calibration*.

Reference frame

All positional information in a metrology template, for physically measured features (including their ROI) and constructed features, is relative to a coordinate system. The coordinate system that you use as your reference is called the reference frame.

Every feature and feature ROI has a reference frame with which it is associated. The reference frame can either be the global frame (created by default) or a local frame (user-defined feature).

You can reposition, rotate, or change reference frames, which can be useful when dealing with multiple parts in the same image.



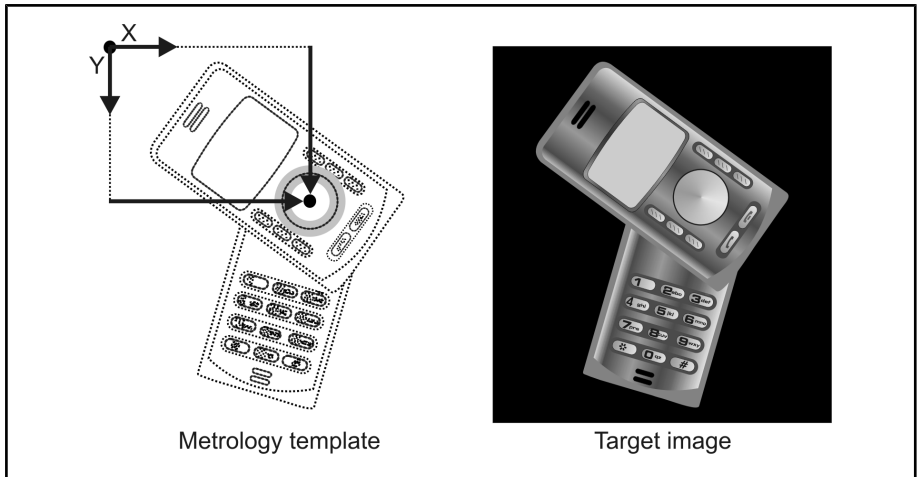
To reposition or rotate a reference frame, use **MmetControl()** with **M_POSITION_X**, **M_POSITION_Y**, and **M_ANGLE**. To change a feature's reference frame, use **MmetControl()** to specify the feature (**M_FEATURE_INDEX** or **M_FEATURE_LABEL**), and **M_REFERENCE_FRAME** to set the label or index of its new reference frame.

If you change a reference frame, move a reference frame, or alter its angle, all features and feature ROIs associated to that reference frame are modified accordingly.

Global frame

By default, a feature's reference frame is set to the global frame, where the origin (0,0) is aligned with the target image's top-left corner. The global frame is automatically created when you allocate a metrology context. There is only one global frame.

In the following example, the position (center) of a ring-shaped ROI is set according to the global frame.

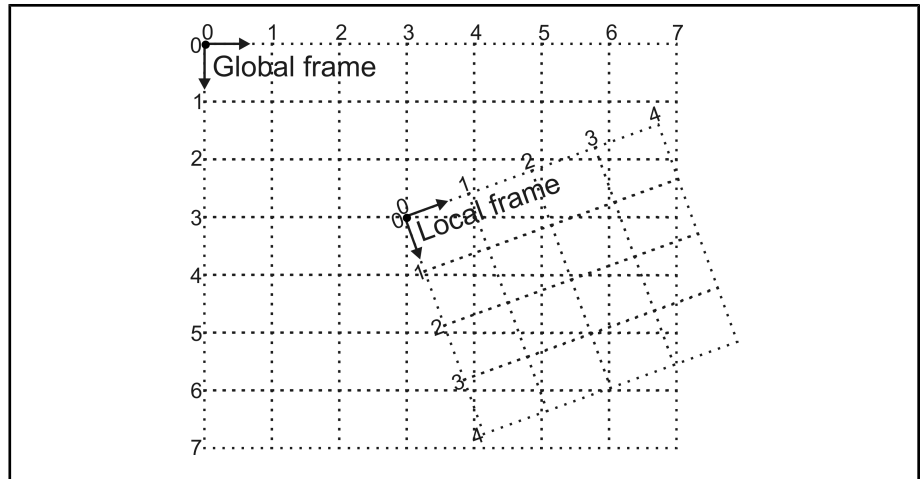


- ❖ If the target image is calibrated, the global frame's origin is aligned with the origin of the calibration.

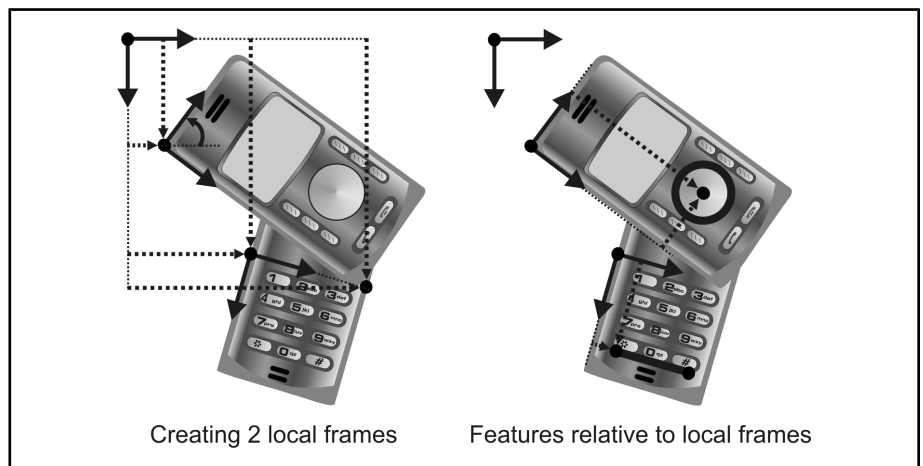
Local frame

In addition to the default reference frame (global frame), you can add your own reference frame feature anywhere within the global frame. Each reference frame you add is called a local frame.

To add a local frame, use **MmetAddFeature()** with **M_LOCAL_FRAME**. You can add a local frame by cloning it from another local frame (**M_CLONE_FEATURE**), constructing it from two points or one point and an angle (**M_CONSTRUCTION**), or by providing specific position and angle values (**M_PARAMETRIC**).

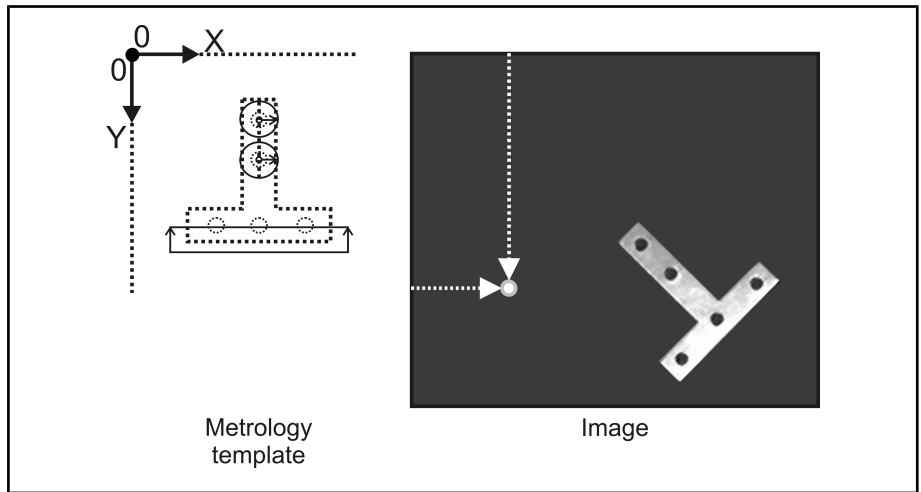


The following example illustrates the use of three reference frames: the global frame, a local frame constructed with position and angle values, and a local frame constructed with two points. The first illustration shows how the local frames are created. The second illustration shows how features can be added (a circle and a segment) according to each local frame. This is useful since the two parts of the phone swivel, though they are independently stable.



Set position

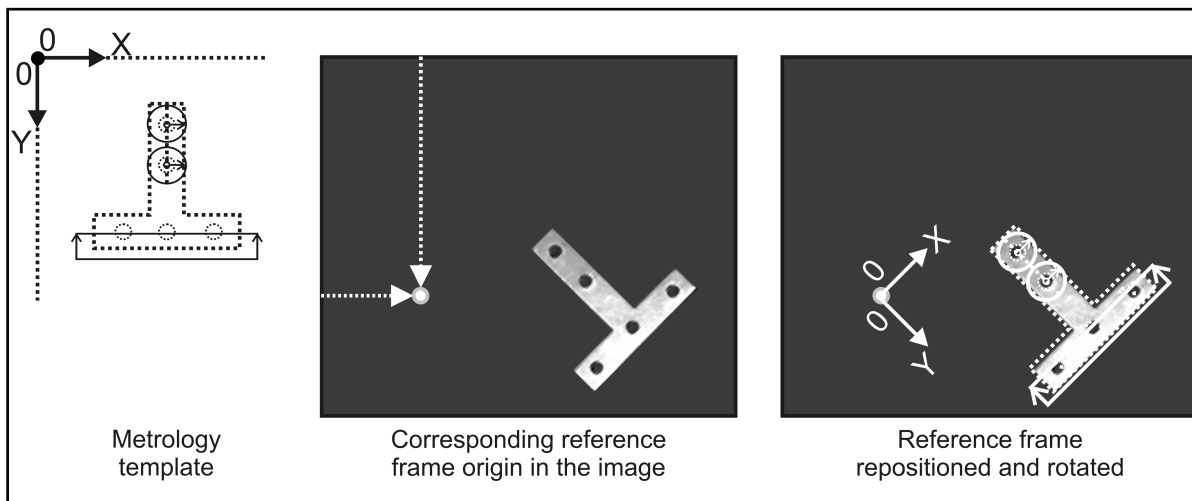
When you are setting the position of a reference frame within the metrology template, you are typically assuming that the corresponding position in the target image is at the same location. However, in real-world applications, the location of this position within the target image does not always correspond to the position in the template.



To help solve this problem, you can change the position of your reference frame just before performing metrology calculations.

To change the position of your reference frame, use **MmetSetPosition()** with explicit positional values (**M_POSITION**), translation and rotation values (**M_GEOMETRIC**), transformation coefficients (**M_FORWARD_COEFFICIENTS**), or the results of another MIL module (**M_RESULT**).

For example, you can use the MIL Model Finder module to find the object, and then use that occurrence's found position to place the reference frame with which to perform metrology calculations. For more information on an occurrence's found position, see the *Reference axis* subsection in the *Customizing search settings* section in *Chapter 8: Geometric Model Finder*.



Positions set with **MmetSetPosition()** are not saved with the metrology context. Note that you are not limited to repositioning the reference frame; you can move the expected location of any physically measured or parametrically constructed feature. To do so, use **MmetSetPosition()** and specify the label or index of the feature. Moving a feature other than the reference frame is typically useful only for very advanced applications.

(Physically) measured features

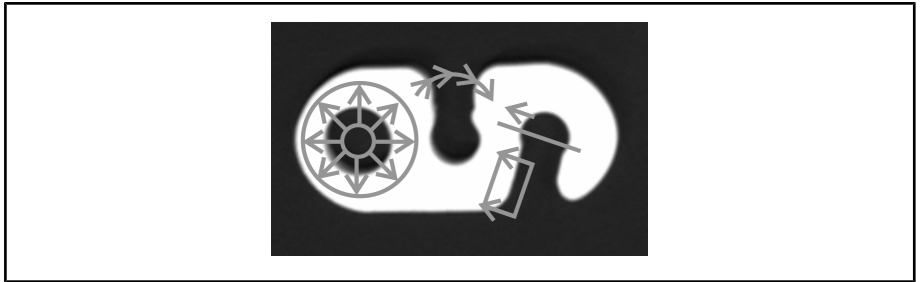
A physically measured feature is a geometric primitive (segment, arc, circle, point, and edgel) that can be physically measured within a specified ROI in the target image. To add a physically measured feature to the template, use **MmetAddFeature()** with **M_MEASURED**.

All physically measured features are composed of edgels (edge elements). A physically measured segment, arc, circle, and point feature contains edgels that form its definitive geometric shape; conversely, an edgel feature is a group of edgels without a definitive geometry. For more information, see the the *(Physically measured) edgel features* subsection in this section.

ROI

The ROI is the region in the target image that contains the edgels you want to use to add the physically measured feature. By default, the entire target image is used as the ROI. However, to reduce the likelihood of getting unwanted features, and to speed up the calculation, you will typically want to restrict the area from which edgels are used to calculate the feature. To delimit the ROI, use **MmetSetRegion()**. In general, good results come from well-defined ROIs.

Each ROI has an orientation, which refers to the alignment of an ROI, with respect to a reference frame. The following example shows 4 ROI's (ring, arc, segment, and rectangle) and their orientation, as depicted with an arrow. Note that you can draw an ROI (and its orientation) using **MmetDraw()** with **M_DRAW_REGION**.



For the features to be extracted, they must not only fall within the ROI, but their edgels' gradient angle must also fall along the region's orientation. Various control types can be used to set a valid relationship between the ROI's orientation, and the edgels' gradient angle. For more information, see the *Gradient angle* subsection in the *Degrees of freedom* section in *Chapter 15: Metrology*.

The ROI's position is set relative to the global frame by default. You can, however, use the coordinate system of any reference frame.

The following table lists all available ROIs, and the physically measured features that can be extracted from them.

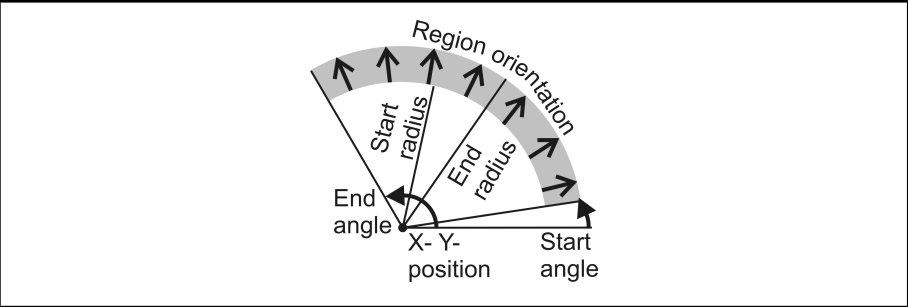
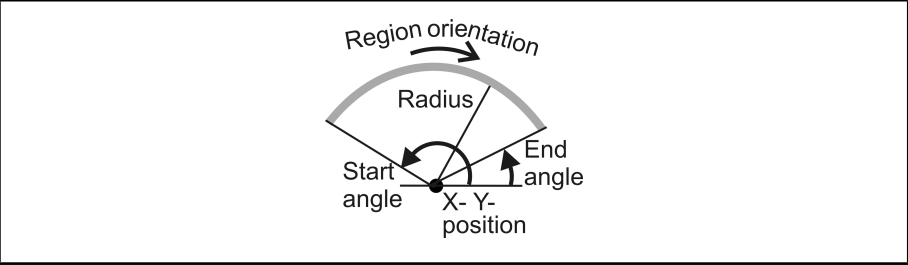
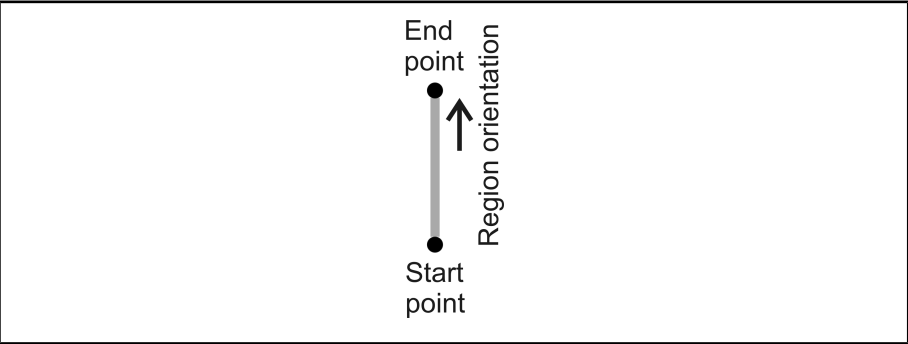
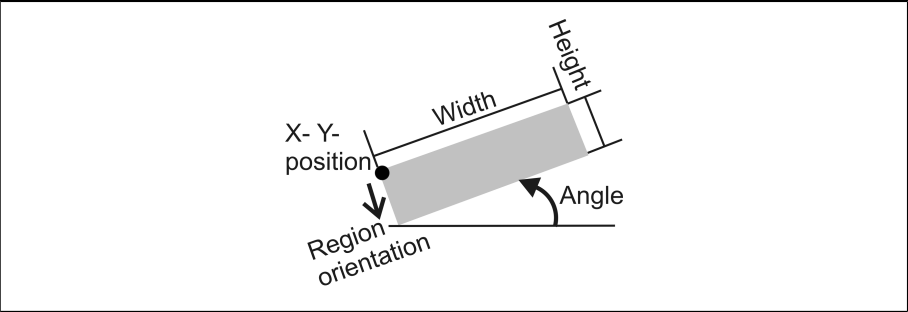
ROI	Feature				
	Point	Arc	Circle	Segment	Edgel
Infinite (default)	Yes	Yes	Yes	Yes	Yes
Rectangle	—	—	—	Yes	Yes
Segment	Yes	—	—	—	—
Arc	Yes	—	—	—	—
Ring	—	—	Yes	—	Yes
Ring-sector	—	Yes	—	Yes	Yes

For an infinite ROI (**M_INFINITY**), you must set the X- Y-position of the origin of the region. You must also explicitly set the angle of orientation (the default is the same as the reference frame). For a rectangular ROI (**M_RECTANGLE**), you must set the X- Y-position of the origin of the region. You must also set the region's width, height, and angle. The orientation is determined by the origin's position and angle. For a segment-shaped ROI (**M_SEGMENT**), you must set the X- Y-position of the segment's start point and end point. The orientation is from start point to end point. For an arc-shaped ROI (**M_ARC**), you must set the X- Y-position of the origin of the region. You must also set the region's radius, start angle and end angle. The orientation is from start angle to end angle.

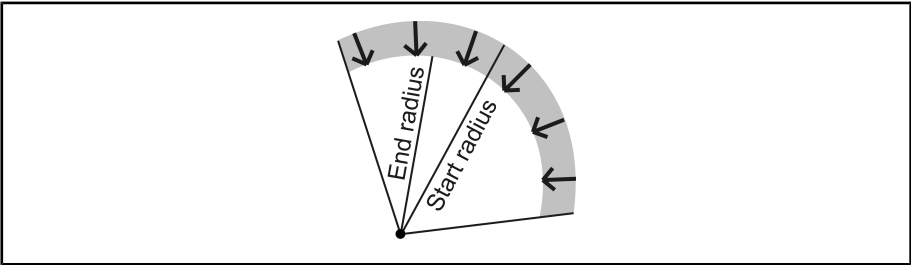
For a ring-shaped ROI (**M_RING**), you must set the X- Y-position of the origin (center) of the region. You must also set the start radius and the end radius. To delimit only a sector of a ring (**M_RING_SECTOR**), you must set the start angle and end angle of the ring. The orientation is from the start radius to the end radius. Also, the region itself is set counter-clockwise, from the start angle to the end angle (for ring-sector).

Note that **M_INFINITY** is similar to **M_RECTANGLE**, however an infinite region does not have a width and a height (boundaries).

The following examples illustrate rectangle, segment, arc, and ring-sector regions.



Note that, for ring-shaped ROIs, switching the start radius and end radius reverses the orientation.



Unlike other physically measured features, multiple measured points can be extracted from the ROI. For more information, see the *Multiple features* subsection in the *Features* section in *Chapter 15: Metrology*.

Best fit, inner fit, and outer fit operations

To add a physically measured feature, you must typically set the fit operation to use to build the feature from the edgels extracted from the ROI. All features cannot be used with all operations.

The following table lists the possible combinations. For more information on which edgels in the feature's ROI are considered by the fit operations, see the *Active edgels* subsection in the *Degrees of freedom* section in *Chapter 15: Metrology*.


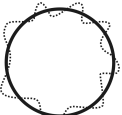
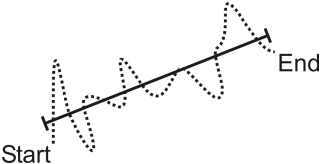
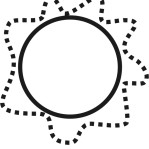
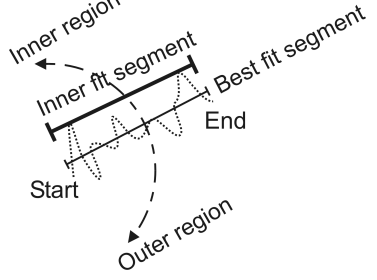

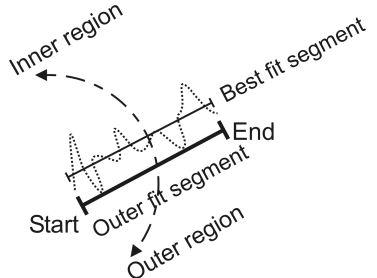
Build operation	Feature to build				
	Point	Arc	Circle	Segment	Edgel
Best fit (default)	Yes	Yes	Yes	Yes	—
Inner fit	—	—	Yes	Yes	—
Outer fit	—	—	Yes	Yes	—

A feature built with the best fit operation (**M_FIT**) will pass as close to as many of the edgels as possible. Note that the best fit segment separates the inner and outer fit regions, which are used to calculate the inner and outer fit segments. The starting point of the best fit segment is the extremity closest to the origin of the reference frame.

A feature built with the inner fit operation (**M_INNER_FIT**) will represent the innermost boundary of the set of the extracted edgels. For a circle feature, the inner fit is the largest possible circle that contains none of the edgels. For a segment feature, the inner fit depends on the best fit segment. The edgels used to build the inner fit segment are on the counter-clockwise side of the best fit segment, relative to its direction. The inner fit segment passes through the two farthest edgels of the inner fit region such that all the edgels of the inner fit region are located on the same side of the inner fit segment.

A feature built with the outer fit operation (**M_OUTER_FIT**) will represent the outermost boundary of the set of edgels. For a circle feature, the outer fit is the smallest possible circle that contains all of the edgels. For a segment feature, the outer fit depends on the best fit segment. The edgels used to build the outer fit segment are on the clockwise side of the best fit segment, relative to its direction. The outer fit segment passes through the two farthest edgels of the outer fit region such that all the edgels of the outer fit region are located on the same side of the outer fit segment.

The following table illustrates an arc, circle, and segment feature added with the different fit operations; the operations are performed on the edgels provided.

Build operation	Feature to build		
	Arc	Circle	Segment
Best fit			
Inner fit	Not available		
Outer fit	Not available		

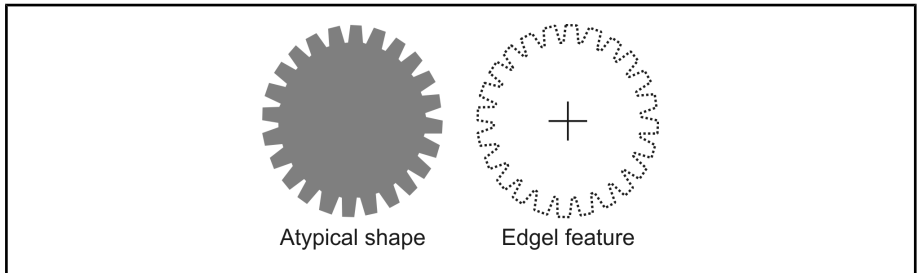
A feature built with a fit operation will not necessarily touch the edgels on which it is based. To determine the length of a segment feature that is built using a fit operation, the active edgels in the feature's ROI are projected onto the segment. For more information, see the *Active edgels* subsection in the *Degrees of freedom* section in *Chapter 15: Metrology*.

(Physically measured) edgel features

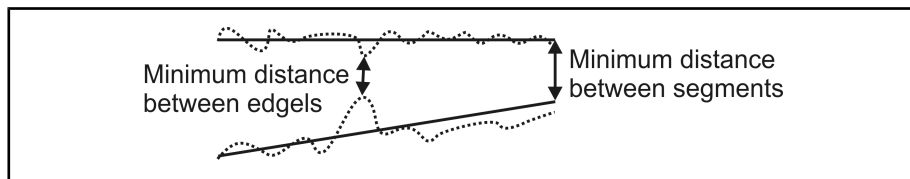
Edgel features do not have a definitive geometry; they are built using the raw information (edgels) extracted from the ROI. Each edgel represents an X- and Y-position, and an angular direction (of the gradient). An edgel feature typically contains numerous edgels.

To add an edgel feature, use **MmetAddFeature()** with **M_MEASURED** and **M_EDGEL**. By default, edgel features are built using all the edgels in the feature's ROI. However, you can choose to eliminate unwanted edgels, based on their gradient angle; this can prove useful since it allows you to customize the extracted edgels to fit your needs. For more information, see the *Gradient angle* subsection in the *Degrees of freedom* section in *Chapter 15: Metrology*.

An edgel feature is useful to add an atypical shape to the metrology template. That is, you can add features that do not represent a geometric primitive, such as an arc, circle, point, or segment.



The edgel information of a feature might also prove useful in several types of advanced applications. For example, you might want to set a true minimum distance tolerance between the edgels that compose two segments, rather than having the minimum distance based on the fitted segments that are built from the edgels.

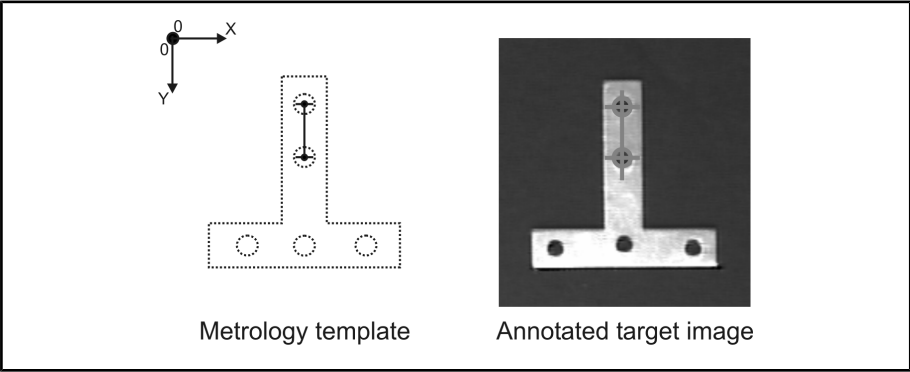


Constructed features

A constructed feature is a geometric primitive (point, arc, circle, segment, line, edgel, and local frame) that has been mathematically defined or geometrically derived from a set of other features (either constructed or physically measured). You do not set an ROI for constructed features.

To add a constructed feature, use **MmetAddFeature()** with **M_CONSTRUCTED**. For constructed features that are geometrically derived, you must also specify the label of the feature(s) to use for construction.

The following example shows a constructed segment added from the constructed center points of 2 physically measured circles.



To add a constructed feature, you must set the operation with which to build the feature. For a constructed feature that is geometrically derived, the operation selects how to use the specified feature(s) to build the constructed feature.

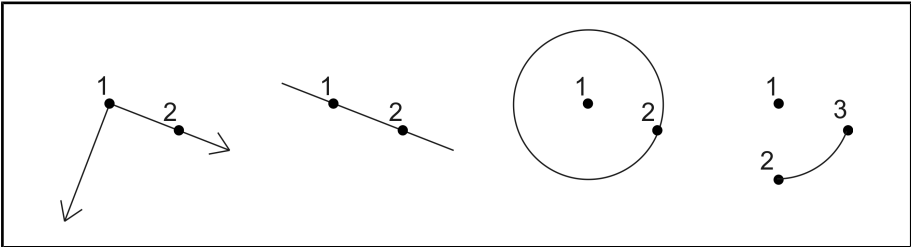
The following table lists all constructed features, and the operations to build them.

Build operation	Feature to build						
	Point	Arc	Circle	Segment	Line	Edge1	Local frame
Absolute angle	Yes	—	—	—	—	—	—
Relative angle	Yes	—	—	—	—	—	—
Absolute position	Yes	—	—	—	—	—	—
Relative position	Yes	—	—	—	—	—	—
Start position	Yes	—	—	—	—	—	—
End position	Yes	—	—	—	—	—	—
Middle	Yes	—	—	—	—	—	—
Center	Yes	—	—	—	—	—	—

Build operation	Feature to build						
	Point	Arc	Circle	Segment	Line	Edgel	Local frame
Closest	Yes	—	—	—	—	—	—
Max distance point	Yes	—	—	—	—	—	—
Extended intersection	Yes	—	—	—	—	—	—
Intersection	Yes	—	—	—	—	—	—
Parallel	—	—	—	—	Yes	—	—
Perpendicular	—	—	—	—	Yes	—	—
Angle	—	—	—	—	Yes	—	—
Relative sector	—	—	—	—	Yes	—	—
Bisector	—	—	—	—	Yes	—	—
Outer fit	—	—	Yes	Yes	—	—	—
Inner fit	—	—	Yes	Yes	—	—	—
Best fit	—	Yes	Yes	Yes	Yes	—	—
Basic construction	—	Yes	Yes	Yes	Yes	—	Yes
Clone	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Copy feature edgels	—	—	—	—	—	Yes	—
Parametric (default)	Yes	Yes	Yes	Yes	Yes	—	Yes

Basic construction

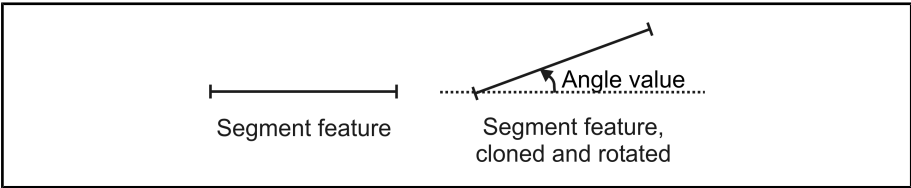
The basic construction operation (**M_CONSTRUCTION**) creates a constructed feature based on the minimum amount of information (that is, point features) needed for construction. The number of point features required, and how they are implemented, are dependent on the feature you are constructing. The following example shows how a local frame, line, circle, and arc are built using 2 or 3 points.



To build a constructed feature with the basic-construction operation, use **MmetAddFeature()** with the feature to add, **M_CONSTRUCTION**, and the label of the points required.

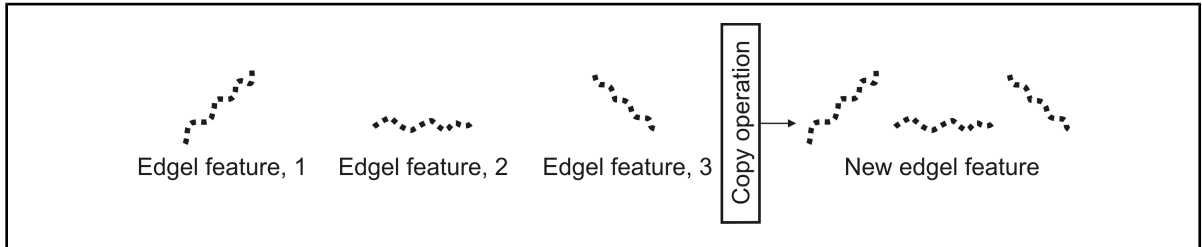
Cloning and copying

The clone operation (**M_CLONE_FEATURE**) creates a constructed feature that is an exact duplicate, or a transformed replica, of any feature (the source). To modify certain aspects of a feature when it has been cloned from a source feature, use **MmetControl()**. For example, you can translate (**M_CLONE_OFFSET_X**, **M_CLONE_OFFSET_Y**), rotate (**M_CLONE_ANGLE**), and/or scale (**M_CLONE_SCALE**) a cloned feature.



When modifying a cloned feature, values (position, scale, and angle) are relative to the source feature's reference frame. When cloning a reference frame, its associated features are not cloned. As with cloning, copying creates a duplicate of the source feature. However, unlike cloning, many features can be copied (**M_COPY_FEATURE_EDGELS**).

Each source feature must contain physically measured edgels. Copying creates a constructed edgel feature that contains all the specified edgels of each physically measured edgel feature that you provide.



By default, all the edgels of the features that you provide will be used to create the copied edgel feature. However, you can select to copy only the active edgels. Active edgels include the edgels used for the fit operation (used to build the physically measured feature(s) that you provide), and the edgels that meet all other fit and edgel constraints, such as the valid range for the edgels' gradient angle. To construct edgel features by copying only the active edgels, use **MmetControl()** with **M_EDGEL_TYPE** set to **M_ACTIVE_EDGELS**. For more information, see the *Active edgels* subsection in the *Degrees of freedom* section in *Chapter 15: Metrology*.

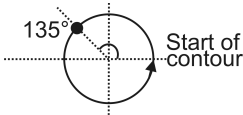
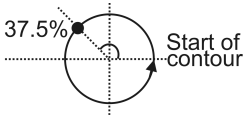
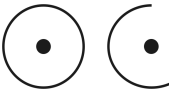
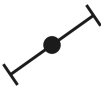
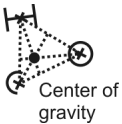
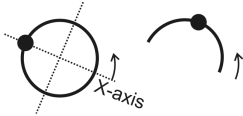
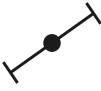

Fit





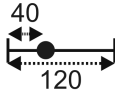

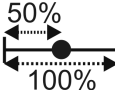
The fit operations create a constructed feature that is the best fit (**M_FIT**), inner fit (**M_INNER_FIT**), or outer fit (**M_OUTER_FIT**), of the feature information that you provide. This information can be points and/or edgels; the added feature will not necessarily touch these points/edgels. For example, you can add a constructed segment that is built using the best fit of a physically measured point feature (which contains multiple physically measured points). In this case, the constructed segment will pass as close to as many of the points as possible.

In general, fit operations for constructed features and physically measured features behave similarly. For more information, see the *Best fit, inner fit, and outer fit operations* subsection in the *Features* section in *Chapter 15: Metrology*.

Operations for building point features

A point feature can be constructed using one of several specialized operations. For example, you can use the center operation (**M_CENTER**) to construct a point at the center of a segment. The following table illustrates some of the operations available to build a point based on the specified feature(s).

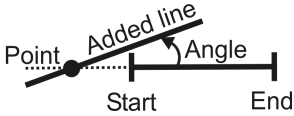
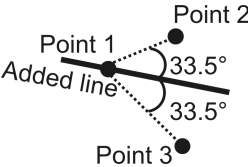
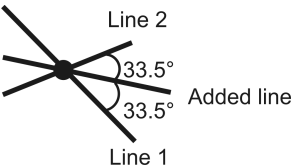
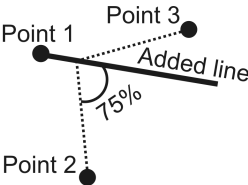
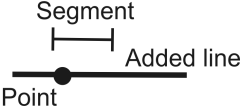
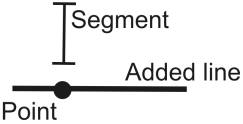
Build operation	Feature(s) used to build the point		
	Circle or Arc	Segment	Several features
Absolute angle		Not available	Not available
Relative angle		Not available	Not available
Center			
Middle			Not available
Closest	Not available	Not available	

Build operation	Feature(s) used to build the point		
	Circle or Arc	Segment	Several features
Max distance point	Not available	Not available	
Intersection	Not available	Not available	
Extended intersection	Not available	Not available	
Absolute position	 Length units	 Length units	Not available
Relative position	 Percentage	 Percentage	Not available

Operations for building line features

A line feature can be constructed using one of several specialized operations. For example, you can use the parallel operation (**M_PARALLEL**) to construct a line parallel to a specified segment.

The following table illustrates some of the operations available to build a line based on the specified feature(s). Some operations also require an angle value.

Build operation	Feature(s) used to build the line		
	A segment and a point	Three points	Two lines
Angle		Not available	Not available
Bisector	Not available		
Relative sector	Not available		Not available
Parallel		Not available	Not available
Perpendicular		Not available	Not available

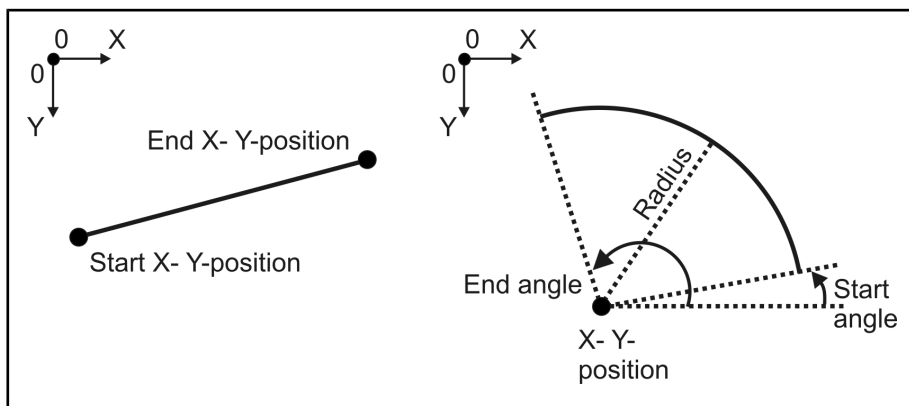
Parametrically constructed features

Most constructed features can be built mathematically (**M_PARAMETRIC**) by specifying explicit values, such as the feature's position and angle. You must use **MmetControl()** to specify the required values needed for the construction.

For example, to build a line parametrically, you must use **MmetControl()** with **M_LINE_A**, **M_LINE_B**, and **M_LINE_C** to specify the A , B , and C parameters of the line equation ($Ax + By + C = 0$).

Note that since a line is infinitely long, it has no position.

The following illustration shows how a segment and arc are constructed by providing explicit positional values. For an arc, you must also provide its radius, start angle, and end angle.



Multiple features

A multiple feature is a feature that holds several instances of that same feature type; every instance of the feature type is considered a subfeature of the multiple feature. For example, a point feature can be a collection of several physically measured points, each of which is a subfeature (subpoint) of the point feature. Point and edgel features are considered multiple features. In certain cases, you can access subpoints; however, you can never access subedgels.

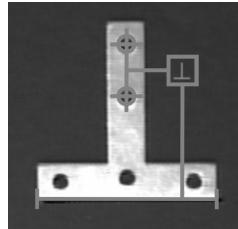
A multiple point feature is created when the ROI associated with a point feature contains more than one edgel. The subfeatures are numbered in ascending order along the orientation of the ROI.

You might need to specify the individual subfeatures (points) to use when adding a constructed feature based on a subfeature of a multiple physically measured point feature. To do so, use the **FeatureLabelArrayPtr** parameter to specify the label of the multiple point feature, and use the **SubfeatureIndexArrayPtr** parameter to specify the indices of the subpoints. The following table shows how a feature can be added using 8 points (specified from 5 features).

FeatureLabelArrayPtr	SubfeatureIndexArrayPtr
MultiplePointFeature1	19
MultiplePointFeature2	1
MultiplePointFeature2	13
MultiplePointFeature3	1
MultiplePointFeature3	14
MultiplePointFeature3	20
MultiplePointFeature4	8
MultiplePointFeature5	1

Geometric tolerances

After adding features to the metrology template, you will typically add geometric tolerances, using **MmetAddTolerance()**. A geometric tolerance defines the acceptable deviation from the definition of a feature, or the acceptable geometric relationship between multiple features.



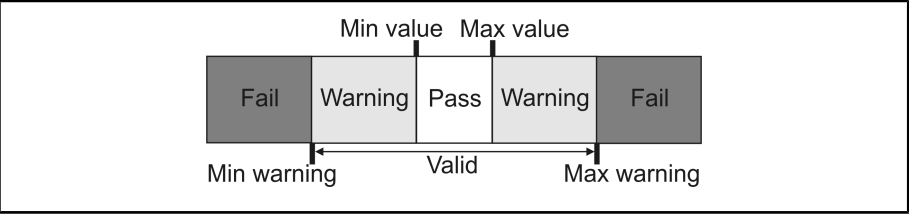
Perpendicularity
tolerance between two
constructed segments

When adding a geometric tolerance, you must set its minimum and maximum limit values. When defining the tolerance of one feature, these limits represent the acceptable deviation from the definition of the feature. For example, a segment's length tolerance can be between 90 and 100 pixels. For multiple features, these limits represent the valid range of acceptable values between the features. For example, the perpendicular tolerance between two segments can be $\pm 0.05^\circ$ (that is, 89.95° to 90.05°). To change the minimum and maximum limit values, use **MmetControl()** with **M_VALUE_MIN** and **M_VALUE_MAX**. You can also use **MmetControl()** with **M_VALUE_WARNING_MIN** and **M_VALUE_WARNING_MAX** to set warning values to alert you when the tolerance is close to its minimum and maximum limits.

When you call **MmetCalculate()**, the tolerance is calculated (for example, 95 pixels) and a status is assigned (for example, **M_PASS**). To retrieve the tolerance value and status, use **MmetGetResult()** with **M_TOLERANCE_VALUE** and **M_STATUS**.

The status of the geometric tolerance returns **M_PASS** or **M_WARNING** if the tolerance meets the specified requirements, and **M_FAIL** if it does not. **M_FAIL** is returned when a tolerance value falls outside the range defined by the minimum and maximum tolerance values and/or the minimum and maximum warning values. **M_WARNING** is returned when a tolerance value is less than or equal to

the minimum tolerance value, but greater than the minimum warning value. A warning also occurs when a tolerance value is greater or equal to the maximum tolerance value, and less than the maximum warning value. **M_PASS** is returned when the value is between the minimum and maximum tolerance values. By default, the minimum and maximum tolerance and warning values are 0. The following is an example of (non-zero) minimum and maximum tolerance and warning values:

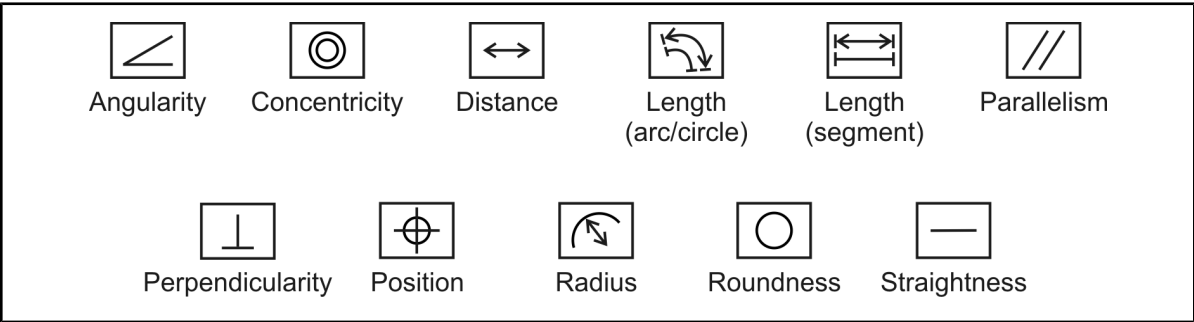


Note that a warning will never be returned if the warning range is within the pass range.

If a calibration object is associated to the target image, set values in real-world units. Otherwise use pixel units (for example, tolerance values and warning values, and positional values). For more information, see the *Calibration overview* section in *Chapter 5: Camera calibration*.

Adding a geometric tolerance

With **MmetAddTolerance()**, you can add one of the following geometric tolerances:



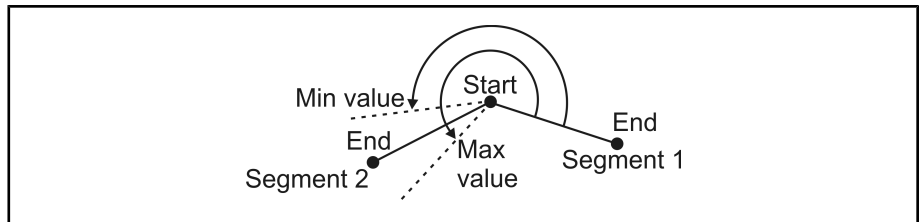
Angularity

An angularity tolerance (**M_ANGULARITY**) validates that the angle between two features falls within the specified angular range (for example, that two lines intersect at an angle between 25° and 35°). An angularity tolerance can be applied between the following features:

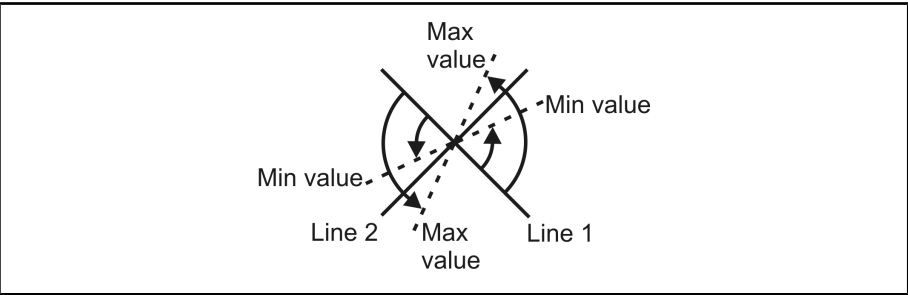
- Two segment features.
- Two line features.
- A linear feature (segment or line) and an edgel feature.

Note that an angularity tolerance between a segment and a line produces too many ambiguities and therefore cannot be calculated.

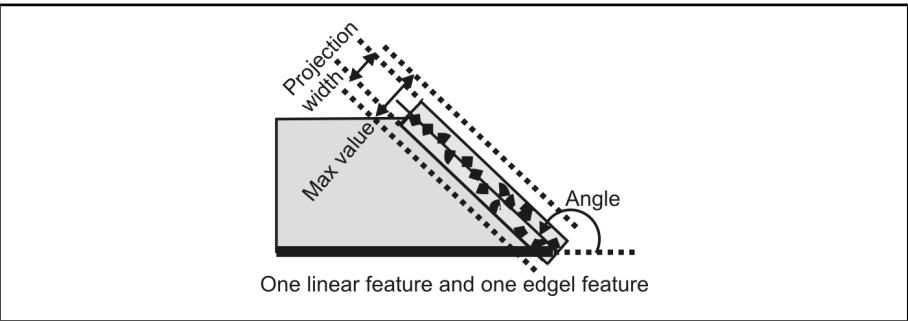
For two segment features, the angle is measured counter-clockwise from the first segment to the second segment. For calculation purposes, the start of each segment is placed at the same point to determine the angle. Therefore, the segments' starting points, as well as the order you list the segments in the **FeatureLabelArrayPtr** parameter, is important. The angle is measured from the first segment to the second segment. The minimum and maximum tolerance parameters set the valid angular range.



For two line features, the angle is measured counter-clockwise from the first segment to the second segment. Therefore, the order you list the segments in the **FeatureLabelArrayPtr** parameter is important. The minimum and maximum tolerance parameters set the valid angular range. Note that there are two ways of measuring this angle; both result in the same value.

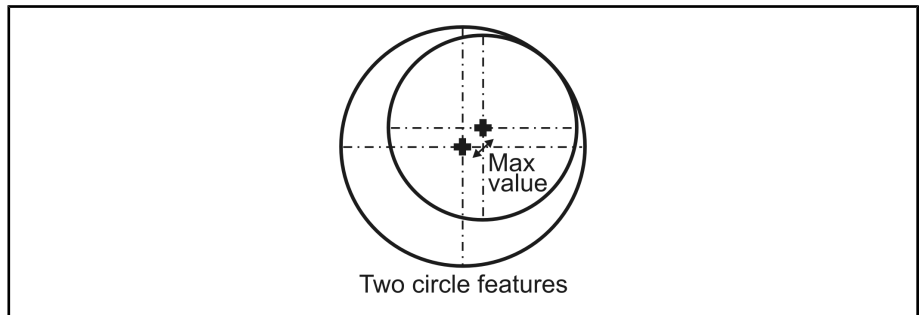


For one linear feature (segment or line) and one edgel feature, **M_ANGULARITY** validates the width of the projection of the edgel feature, along the nominal angle specified using **MmetControl()** with **M_ANGLE**. The angle is in the counter-clockwise direction relative to the linear feature. The minimum and maximum tolerance parameters set the valid projection width. Typically, you should set the minimum width value to 0. The width of the projection is in pixel or world units. The second feature specified in the **FeatureLabelArrayPtr** parameter must be edgel.



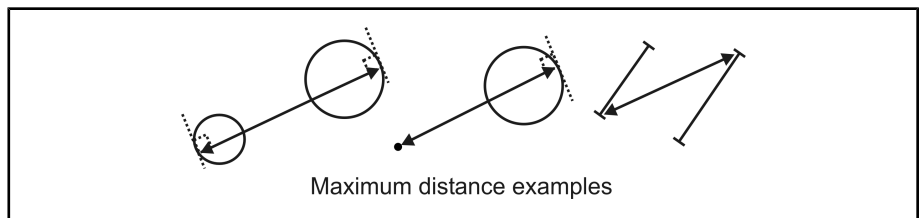
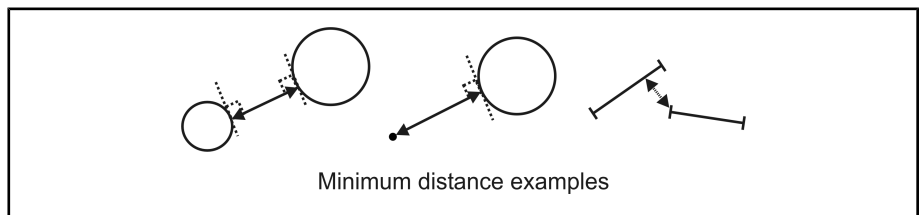
Concentricity

A concentricity tolerance (**M_CONCENTRICITY**) validates that two features have a common center. A concentricity tolerance can be applied to circle and arc features. The minimum and maximum tolerance parameters set the valid minimum and maximum distance allowed between the center of each feature. Typically, you should set the minimum distance value to 0.



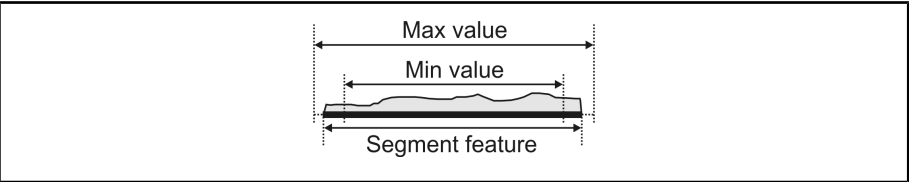
Distance

A distance tolerance (**M_DISTANCE_MIN** and **M_DISTANCE_MAX**) validates that the distance between two features meets either the specified minimum or maximum distance values. A minimum distance tolerance can be applied to any two features; maximum distance tolerances can be applied to any two features, except line. The minimum and maximum tolerance parameters set the valid minimum and maximum distances between features for either the minimum or maximum distance tolerance.



Length

A length tolerance (**M_LENGTH**) validates that the linear dimension of one feature meets the specified value. A length tolerance can be applied to one of the following features: segment, circle, or arc. For circle and arc, length refers to the circumference. The minimum and maximum tolerance parameters set the valid minimum and maximum lengths of the feature.

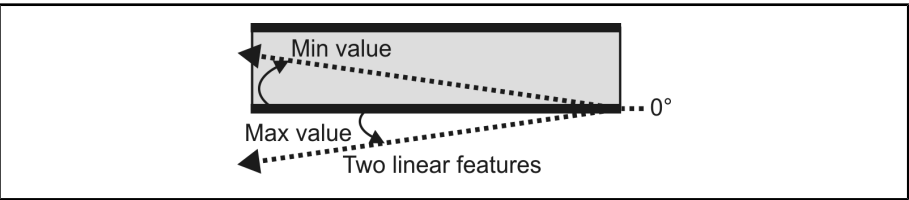


Parallelism

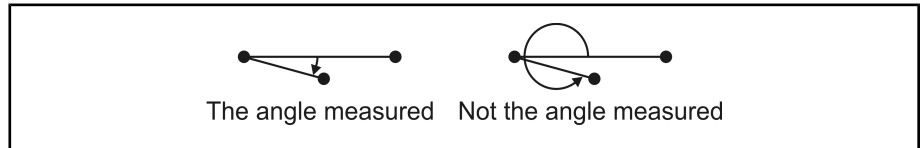
A parallelism tolerance (**M_PARALLELISM**) validates the degree to which two features are parallel. A parallelism tolerance can be applied between the following:

- Two linear features (segments and lines).
- A linear feature (segment or line) and an edgel feature.

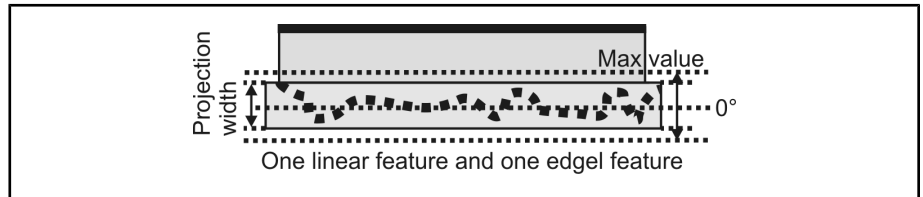
For any combination of 2 linear features, **M_PARALLELISM** validates that the angle of the two features, relative to the global reference frame, is the same. The minimum and maximum tolerance parameters set the valid angular range, from the nominal angle (0°). The order of the features specified in the **FeatureLabelArrayPtr** parameter, as well as the start/end of the segments, is inconsequential.



For parallelism tolerances between two linear features, angles are remapped to 0°; that is, the angle that is measured, and the angular range that you must provide, is the angle that is closest to 0°.



For a linear feature and an edgel feature, **M_PARALLELISM** validates the width of the projection of the edgel feature on a theoretical line perpendicular to the segment or line feature. The minimum and maximum tolerance values set the valid projection width. The smaller the width, the more the edgel feature is parallel to the given segment or line feature. Typically, the minimum width value is 0. The width of the projection is in pixel or world units. The second feature specified in the **FeatureLabelArrayPtr** parameter must be edgel.

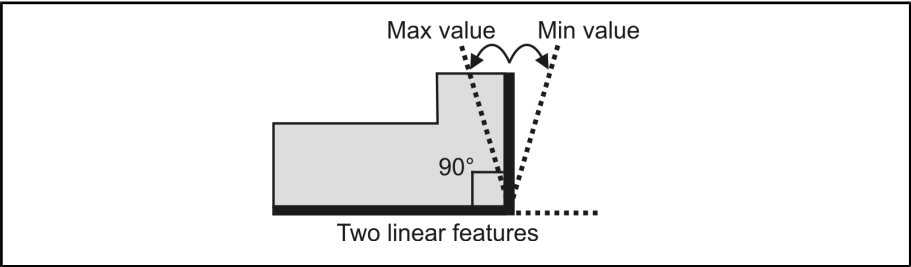


Perpendicularity

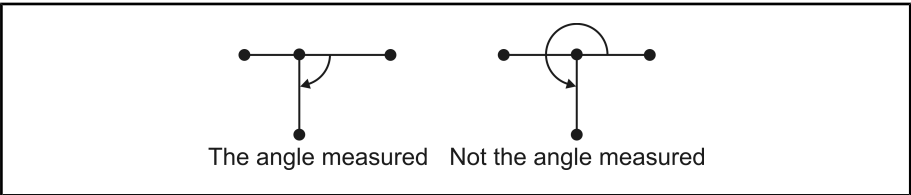
A perpendicularity tolerance (**M_PERPENDICULARITY**) validates the degree to which two features are perpendicular. A perpendicularity tolerance can be applied between the following features:

- Two linear features (segments and lines).
- A linear feature (segment or line) and an edgel feature.

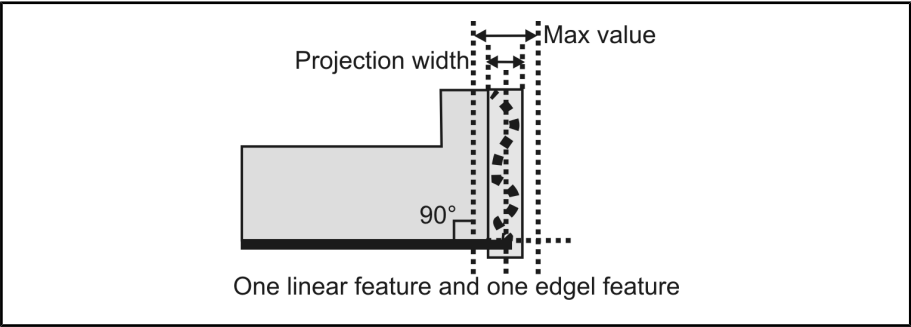
For any combination of 2 linear features, **M_PERPENDICULARITY** validates that the angle between the two features is 90° . The minimum and maximum tolerance parameters set the valid angular range, from the nominal angle (90°). The order of the features specified in the **FeatureLabelArrayPtr** parameter, as well as the start/end of the segments, is inconsequential.



For perpendicularity tolerances between two linear features, angles are remapped to 90°; that is, the angle measured and the angular range you provide is the angle closest to 90°.

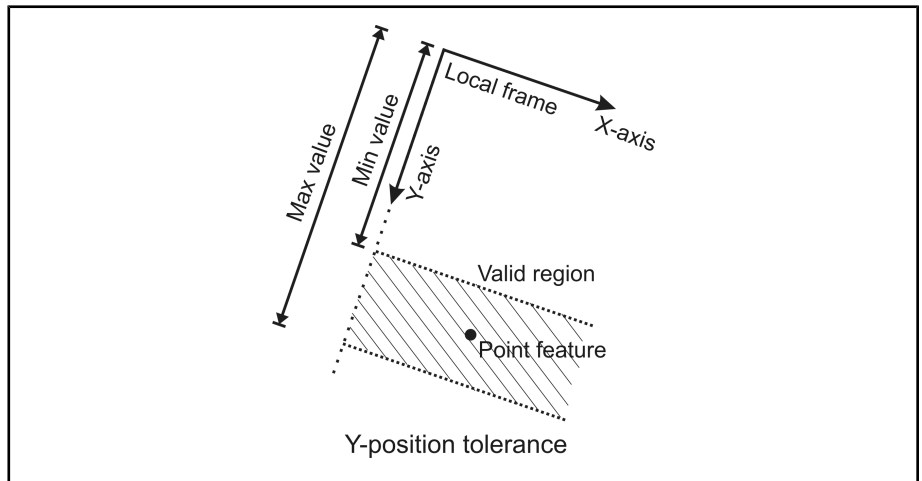
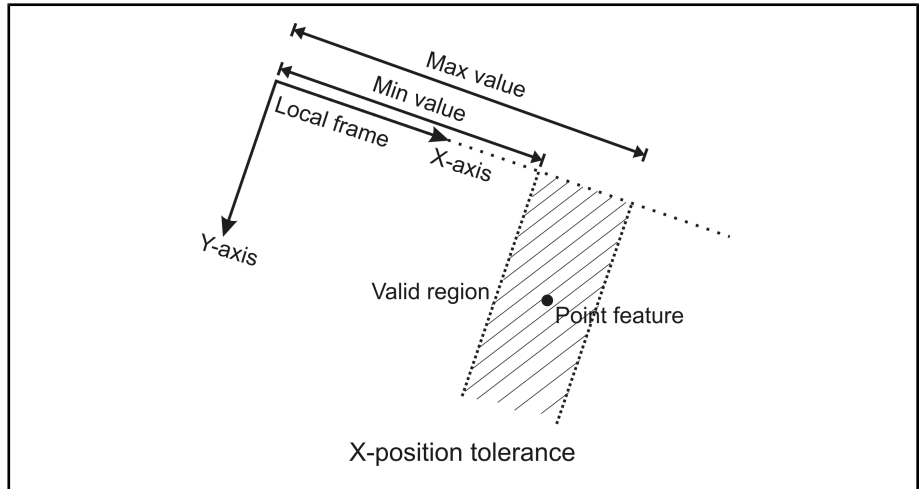


For a linear feature and an edgel feature, **M_PERPENDICULARITY** validates the width of the projection of the edgel feature on a theoretical line parallel to the segment or line feature. The minimum and maximum tolerance values set the valid projection width. The smaller the width, the more the edgel feature is perpendicular to the given segment or line feature. Typically, the minimum width value is 0. The width of the projection is in pixel or world units. The second feature specified in the **FeatureLabelArrayPtr** parameter must be edgel.



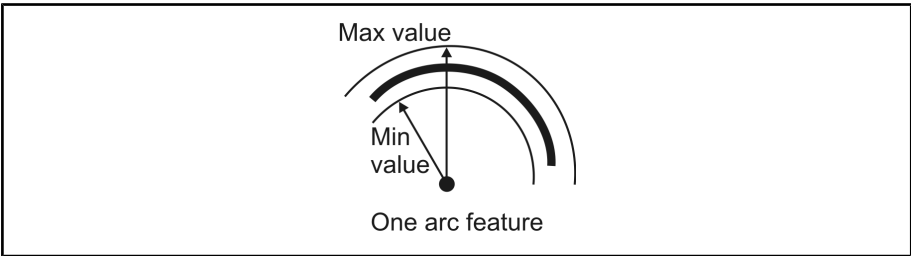
Position

A position tolerance (**M_POSITION_X** and **M_POSITION_Y**) validates that a feature must be located at the specified position, along the X- or Y-direction of the specified reference frame. A position tolerance can be applied to a local frame feature, and one of the following features: local frame, point, circle. The minimum and maximum tolerance parameters set the valid minimum and maximum positions for either the X- or Y-position tolerance.



Radius

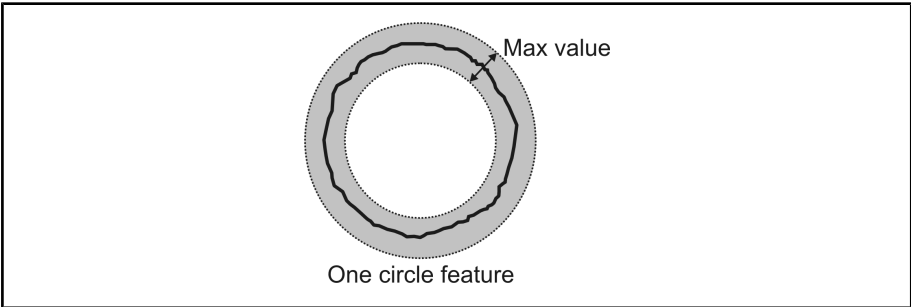
A radius tolerance (**M_RADIUS**) validates that the linear length between the feature's center and perimeter falls within the specified values. A radius tolerance can be applied to one of the following features: arc or circle. The minimum and maximum tolerance parameters set the valid minimum and maximum radius of the feature.



Roundness

A roundness tolerance (**M_ROUNDNESS**) validates that a feature is round. This is done by calculating the distance between the inner and outer circles formed, for example, by the given circle feature, as shown below. Note that the inner and outer circles are concentric.

A roundness tolerance can be applied to circle and arc features. The minimum and maximum tolerance parameters set the valid minimum and maximum width between the two concentric circles or arcs. Typically, you should set the minimum width value to 0.

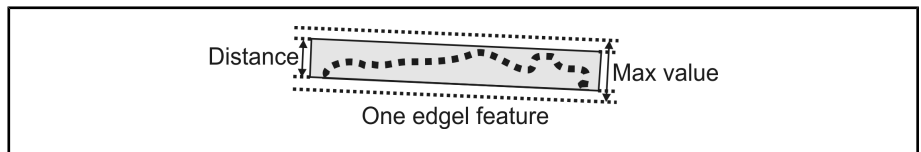


The same concept applies to an arc feature, since an arc is a portion of a circle.

Straightness

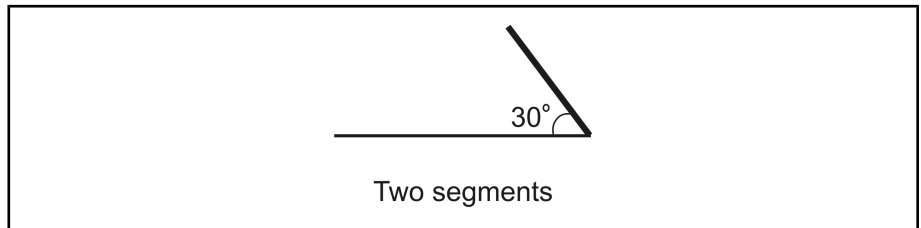
A straightness tolerance (**M_STRAIGHTNESS**) validates that a feature is straight. This is done by calculating the distance between the two parallel lines that are formed by the inner and outer boundaries of the feature. The angle of the two parallel lines is chosen such that the distance between them is the smallest.

A straightness tolerance can be applied to segment and edgel features. The minimum and maximum tolerance parameters set the valid minimum and maximum width between the two parallel lines. Typically, you should set the minimum width value to 0.

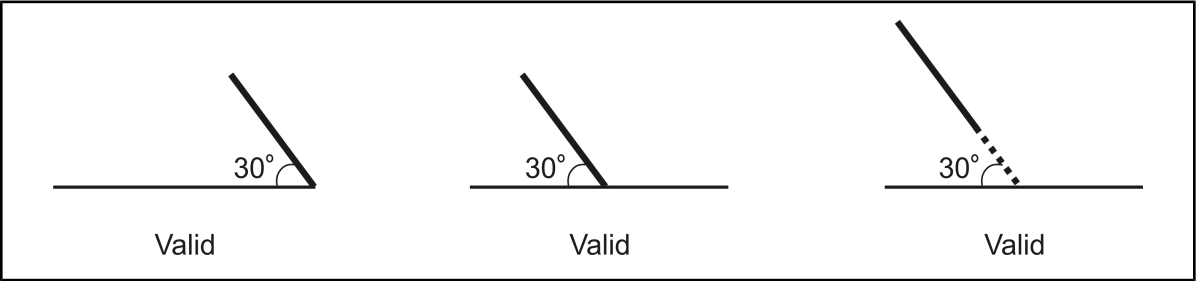


Using multiple geometric tolerances

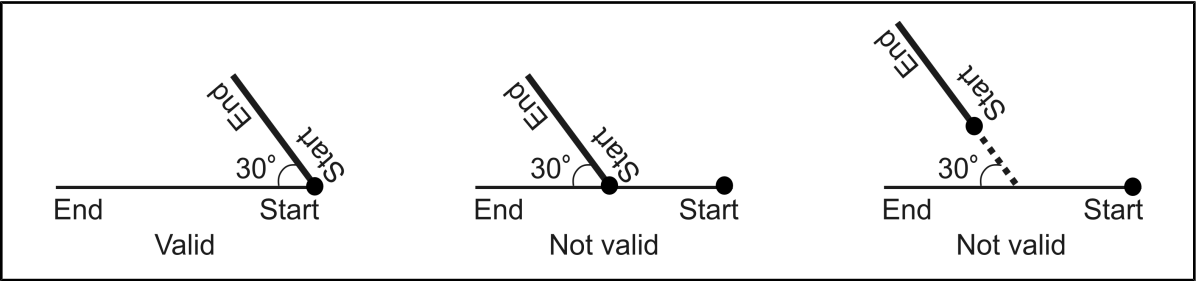
To accurately represent a relationship between features, the features might have to be validated by multiple geometric tolerances. For example, to verify the following result, you might set a 30° angularity tolerance between two segments.



However, using only the angularity tolerance, each of the following results has a valid tolerance status.

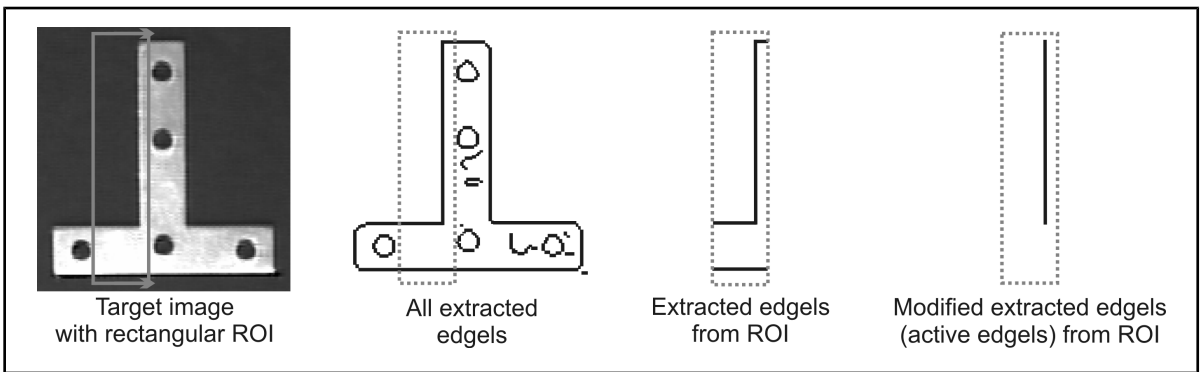


In this case, you need an additional tolerance to check the location of the segments. To do so, you would add two constructed point features at the start of each segment, using **MmetAddFeature()** with **M_POINT** and **M_POSITION_START**. You would then add a distance tolerance to these two points, ensuring they are at the same location. To satisfy the required feature relationship, both the angularity status of the segments and the distance status of the points must be valid.



Degrees of freedom

The Metrology module has been configured, by default, to conduct a fast and robust calculation of physically measured features in the target image. This is done, in part, by typically using all edges in the feature's ROI. However, for unusually complex target images, or to fit your application's needs, you might want to restrict which edges are used. Eliminating unwanted edges can result in a more precise feature extraction, a more accurate fit, and a more robust calculation. Edgels within the ROI that are not eliminated are considered active edgels.



Before adjusting the active edgels, you should ensure that the quality of your image is good. To do so, you can customize the processing settings for the edge extraction algorithm. For example, you can remove image noise by performing a smoothing operation. To adjust the processing settings, use **MmetControl()**, and alter the:

- Edge extraction filter (**M_EXTRACTION_SCALE**, **M_FILTER_MODE**, **M_FILTER_SMOOTHNESS**, **M_FILTER_TYPE**, **M_KERNEL_DEPTH** and **M_KERNEL_WIDTH**).
- Calculation precision (**M_FLOAT_MODE**).
- Magnitude type (**M_MAGNITUDE_TYPE**).
- Edge extraction threshold (**M_THRESHOLD_MODE**, **M_THRESHOLD_TYPE**, **M_THRESHOLD_VALUE_HIGH**, and **M_THRESHOLD_VALUE_LOW**).

These settings can be adjusted for each physically measured feature, or for the entire metrology context, which will affect all physically measured features. For more information on customizing the edge extraction algorithm, see the *Customizing the edge extraction settings* section in *Chapter 9: Edge Finder*.

Active edgels

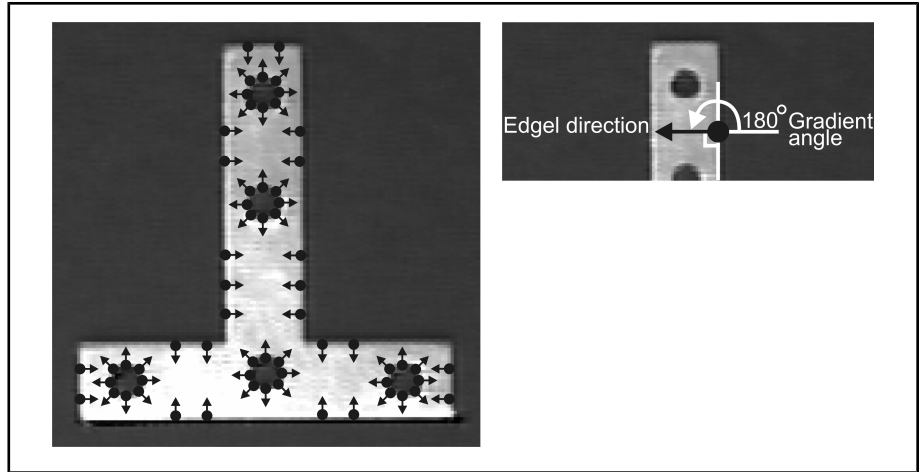
To achieve the appropriate set of active edgels, you can adjust the following constraints:

- The gradient angle restrictions (**MmetControl()** with **M_EDGEL_ANGLE_RANGE** and **M_EDGEL_RELATIVE_ANGLE**).
- The fit operation (**MmetAddFeature()** with **M_FIT**, **M_INNER_FIT**, or **M_OUTER_FIT**).
- The maximum fit iterations (**MmetControl()** with **M_FIT_ITERATIONS_MAX**).
- The minimum fit coverage (**MmetControl()** with **M_FIT_COVERAGE_MIN**).
- The maximum fit variation (**MmetControl()** with **M_FIT_VARIATION_MAX**).
- The maximum fit distance (**MmetControl()** with **M_FIT_DISTANCE_MAX**).

Note that active edgels are used by the fit operation to build the feature, however the fit operation itself can affect which edgels are considered active. That is, fit operations can be performed on the same pool of "active" edgels, but only those edgels used for that specific fit are the true active edgels. For example, an inner fit and outer fit can produce two different features, and thus can ultimately have two different sets of active edgels. For more information on fit operations, see the *Best fit, inner fit, and outer fit operations* subsection in the *Features* section in *Chapter 15: Metrology*.

Gradient angle

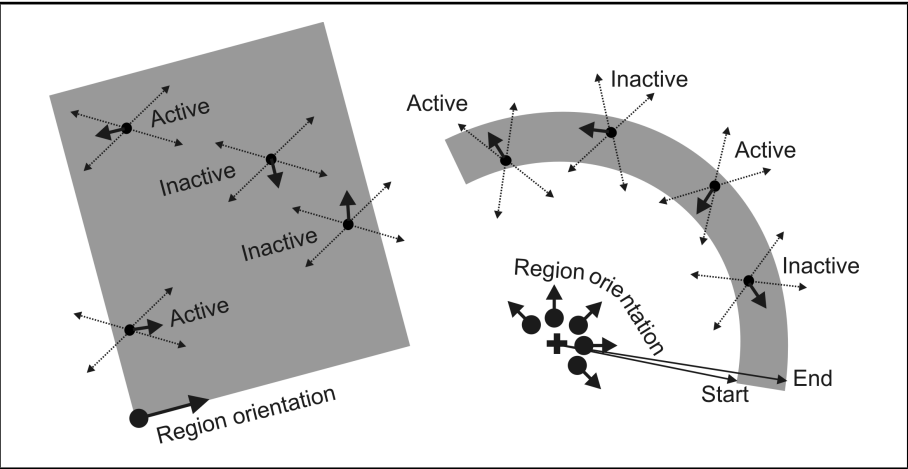
The gradient angle (edgel direction) is the counter-clockwise angle between the target image's horizontal axis and the edge's perpendicular direction (from black to white) at each edgel location. The following example illustrates various edgels and their direction.



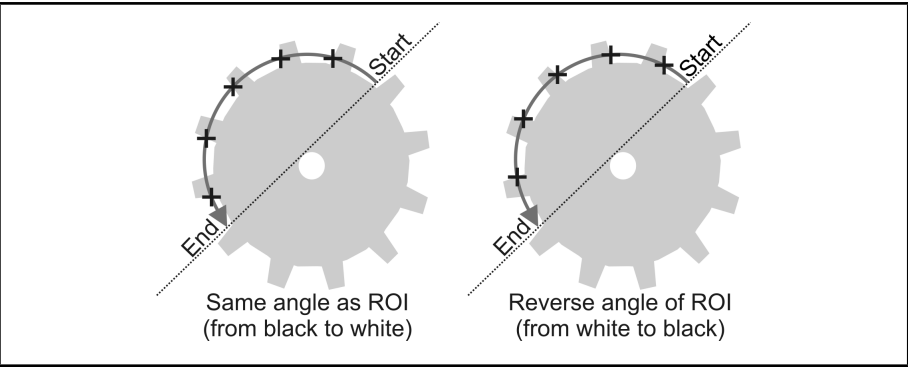
You can filter out unwanted edgels by restricting the degrees of freedom applied to the edgel's gradient angle. To do so, use **MmetControl()** with **M_EDGEL_ANGLE_RANGE**. Only those edgels that have a gradient angle that falls within the angular range can be used to build the feature. You can set the angular range to any value between 0.0° and 360.0° (the default is 180.0°).

By default, the angular range is set relative to the ROI's orientation. To change the relative angle, use **MmetControl()** with **M_EDGEL_RELATIVE_ANGLE**. If the relative angle is set to **M_SAME** (the default), only those edgels that have the same gradient angle (from black to white) as the ROI's orientation, and whose gradient angle falls within the angular range, will be used. If the relative angle is set to **M_REVERSE**, only those edgels that have the opposite gradient angle (from white to black) of the ROI's orientation, and whose gradient angle falls within the angular range, will be used (+180°). You can also set the relative angle to same or reverse (**M_SAME_OR_REVERSE**). Note that for circular ROI's (arc, ring, ring-sector), the ROI's orientation is radial. For more information on ROIs, see the *ROI* subsection in the *Features* section in *Chapter 15: Metrology*.

The following example shows how the gradient angle range (approximately 60°) and the relative angle (same and reverse) settings can be used with a rectangular ROI and ring-sector ROI to select active edgels.



The following example shows how the relative angle settings can be used with an arc-shaped ROI to select different active edge features in an object. Note that to get these results, a slight angular range must also be applied.



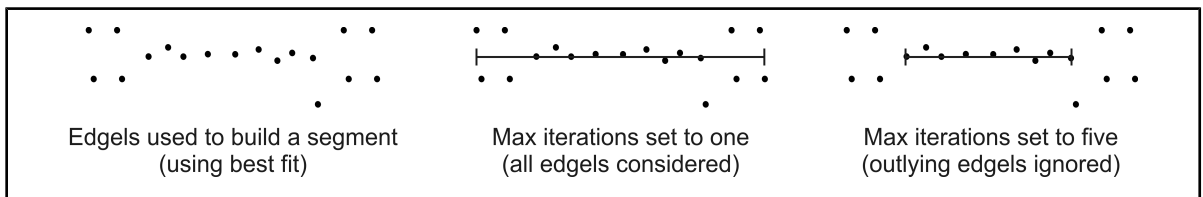
Note that gradient angle restrictions apply when building either physically measured features, or constructed features that use physically measured edgels as a source.

Minimum and maximum fit settings

The fit calculation can be manipulated using the minimum and maximum fit settings (**M_FIT_COVERAGE_MIN**, **M_FIT_ITERATIONS_MAX**, **M_FIT_VARIATION_MAX**, and **M_FIT_DISTANCE_MAX**).

Metrology calculates the fit based on an iterative process, with each iteration resulting in a more accurate fit than its predecessor. This type of process results in a robust fit, where unwanted edgels (for example, noise) are automatically eliminated. The iteration process takes into account the minimum fit coverage (**M_FIT_COVERAGE_MIN**), which represents the quantity of the fitted feature that must be covered by active edgels. You must set this value as a percentage. For example, you can ensure that 80% of the fitted feature is covered by active edgels; the remaining 20% can be gaps. The greater the value, the more active edgels are needed to cover the feature. The default is 0%.

The maximum fit iterations setting (**M_FIT_ITERATIONS_MAX**) allows you to set the maximum number of iterations used to compute a fitted feature. When this value is reached, the iteration process ends. A setting of one will consider all edgels/points in the fit. Settings higher than one will progressively eliminate outlying edgels/points in the fit. The more iterations, the better the fit, but the longer the calculation. By default, the number of iterations is determined automatically.



The iteration process can also end if the maximum fit variation setting (**M_FIT_VARIATION_MAX**) is reached. The maximum fit variation setting represents the maximum allowable difference in the feature's coefficients' values from one iteration to the next. For example, a best-fit line ($Ax + By + C = 0$) has three coefficients: A , B , and C . During each iteration, the best-fit line becomes more accurate and therefore, the values of the line's coefficients change. If, for instance, the maximum fit variation setting is set to 0.05, then as soon as the difference of each coefficients' value between iterations is less than 0.05, the iteration process ends.

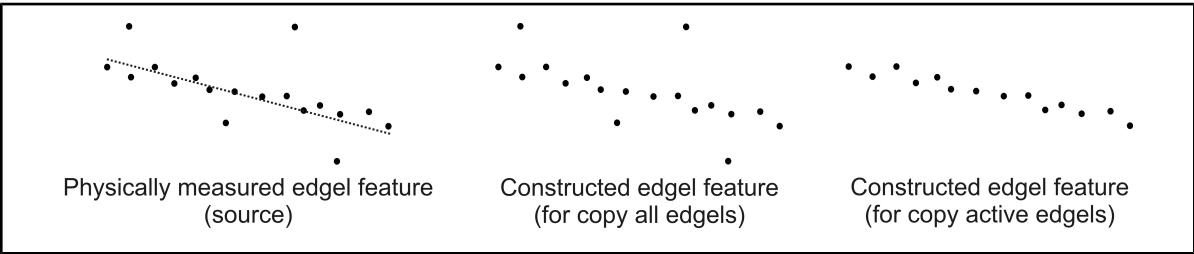
When the iteration process ends, either by reaching the maximum fit iteration or maximum fit variation setting, the maximum fit distance setting (**M_FIT_DISTANCE_MAX**) is applied. This setting allows you to set the greatest possible gap between an active edgel and the projected fit. The higher the value, the farther away an active edgel can be. You must set this value in pixel units. The default is no maximum distance. After the maximum fit distance has been calculated, the remaining edgels are then considered active edgels, provided they pass all other edgel constraints that have been set (for example, the gradient angle restrictions).

Note that the minimum and maximum fit settings apply when building either physically measured features or constructed features with a fit operation.

Copy feature edgels

When adding a constructed edgel feature with the copy feature edgels operation in **MmetAddFeature()**, the new edgel feature can be built by using either the active edgels (default) or all the edgels in the source feature's ROI, regardless of any edgel constraints. For example, if you select the active edgels and use **M_COPY_FEATURE_EDGELS** with a source feature that was built using the best-fit operation for a segment, edgels that were not used to calculate the segment's fit are not included in the new feature.

To select either the active edgels or all edgels, use **MmetControl()** with **M_EDGEL_TYPE** set to **M_ACTIVE_EDGELS** or **M_ALL_EDGELS**. Note that these settings only affect the copy feature edgels build operation. For other build operations, each edgel constraint control must be adjusted according to your active edgel requirements.



Calculating and retrieving results

Once you have added features and geometric tolerances to the metrology template (**MmetAddFeature()** and **MmetAddTolerance()**) and performed a calculation (**MmetCalculate()**), you can retrieve the results from your metrology result buffer, using **MmetGetResult()** with either the label of the feature or the label of the geometric tolerance.

To calculate correct results, it is important to note the calculation units (world or pixel), and the location of the reference frame (in the target image), before calling **MmetCalculate()**. If a calibration object has been associated to the target image, all tolerance and feature values (as well as ROIs and positions) must have been set in the real-world. For more information, see the *Calibration overview* section in *Chapter 5: Camera calibration*. If the position of a reference frame within the metrology template does not correspond to the same position in the target image, you must reposition the reference frame. For more information, see the *Set position* subsection in the *Features* section in *Chapter 15: Metrology*.

Occasionally, the calculation can take an unexpectedly long time. To prevent this, you can use **MmetControl()** with **M_TIMEOUT** to set a maximum calculation time. By default, there is no time limit.

Retrieving the results

Metrology offers several types of results that provide considerable feature and tolerance information. Typically, you retrieve results from your result buffer for one feature or one tolerance at a time. Examples of results that you can retrieve include:

- The calculation status of features and tolerances (**M_STATUS**).
- The feature's measurements (for example, **M_LENGTH**).
- The calculated tolerance value in the target image (**M_TOLERANCE_VALUE**). For example, the actual angle of intersection between two segments.

To verify the availability of a metrology result (if it has been calculated), use **MmetGetResult()** and combine **M_AVAILABLE** to the specified result type.

The status of features and geometric tolerances

Two of the most critical results that you will typically retrieve is the calculation status of your features and geometric tolerances. To do so, use **MmetGetResult()** with **M_STATUS**.

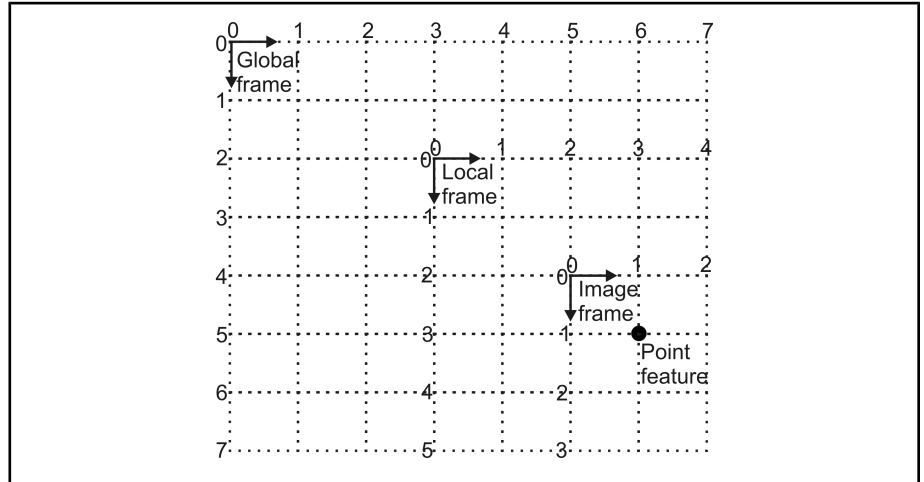
The status of the feature returns **M_PASS** if the feature was successfully validated and **M_FAIL** if it was not. There are many circumstances that can cause the feature to be invalid. For example, a feature is invalid if it does not exist in the specified ROI of the target image; in this case, you might need to move the reference frame (**MmetSetPosition()**). A feature can also be invalid if the information provided is insufficient; this will occur, for example, if you try to construct a point of intersection between two physically measured segments that never actually cross one another.

The status of the geometric tolerance returns **M_PASS** or **M_WARNING** if the tolerance was successfully validated and **M_FAIL** if it was not. Since the Metrology module provides highly sensitive measurement calculations, tolerances will fail unless they are strictly adhered.

Output frame

By default, feature results are returned according to the global frame, even if the feature is defined relative to a local frame. To change the frame used to return feature results (output frame), use **MmetControl()** with **M_OUTPUT_FRAME**. You can set the output frame to the feature's reference frame (**M_REFERENCE_FRAME**), any local frame (valid label of a local frame feature), the target image (**M_IMAGE_FRAME**), or the global frame (**M_GLOBAL_FRAME**).

For example, the following image shows how the X- Y-position of a point feature can be (6,5), (3,3), or (1,1) depending on whether the output frame is set to the global frame, the local frame, or the image frame.



For more information on global and local frames, see the *Reference frame* subsection in the *Features* section in *Chapter 15: Metrology*.

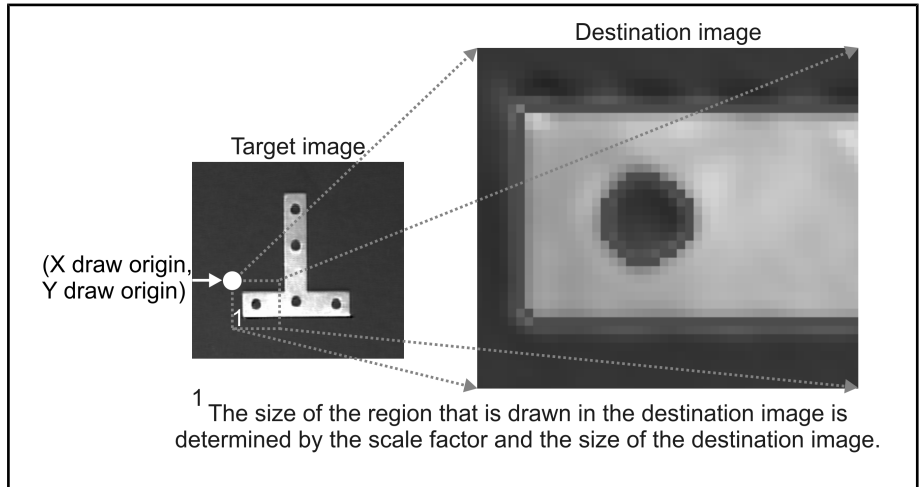
Annotating the results

The `MmetDraw()` provides several operations for drawing results in any specified image buffer. For example, you can draw the successfully validated feature, the extracted edgels, and the geometric tolerance. For the symbology used to draw geometric tolerances, see the *Geometric tolerances* section earlier in this chapter.

You can also choose to draw in the display's overlay buffer. By drawing into the display's overlay buffer, you can annotate an image non-destructively (see the *Annotating the displayed image nondestructively* section in *Chapter 20: Displaying an image*).

Note that if you are using a calibrated target image, the calibration is taken into account when annotating your results; that is, drawings might be distorted, according to the calibration. For example, a straight line in the world might be drawn as a curve in the image.

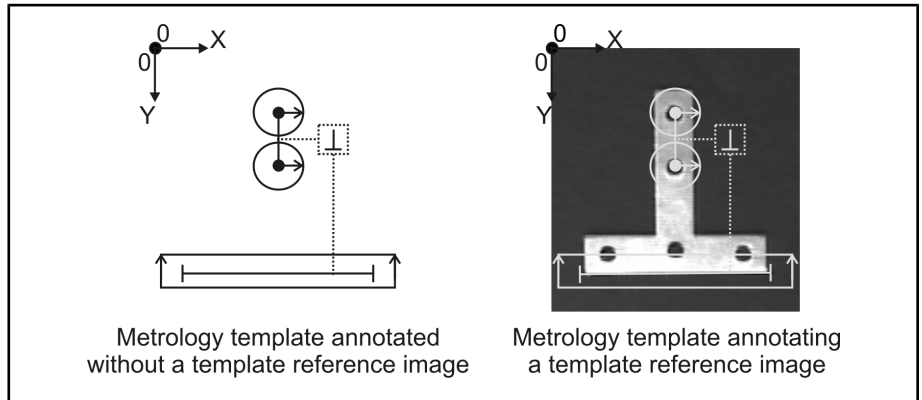
The **MmetDraw()** function also allows you to draw features of a zoomed region (in the target image) by specifying the appropriate values for **MmetControl()** with **M_DRAW_RELATIVE_ORIGIN_X**, **M_DRAW_RELATIVE_ORIGIN_Y**, **M_DRAW_SCALE_X**, and **M_DRAW_SCALE_Y** control types. The relative origin values specify, in pixels, the coordinates of the top-left corner of the region in the target image, while the scale values specify the X- and Y-scaling factors used to fill the destination image buffer.



When zooming, **MmetDraw()** will draw in the destination image buffer even if the buffer is not large enough to contain all of the zoomed image from the specified starting point. If necessary, the image will be clipped.

Template reference

A template reference is an image or result buffer that you can use as an example of a target image. Since it provides a visual sample, a template reference can help you build or modify a metrology template.



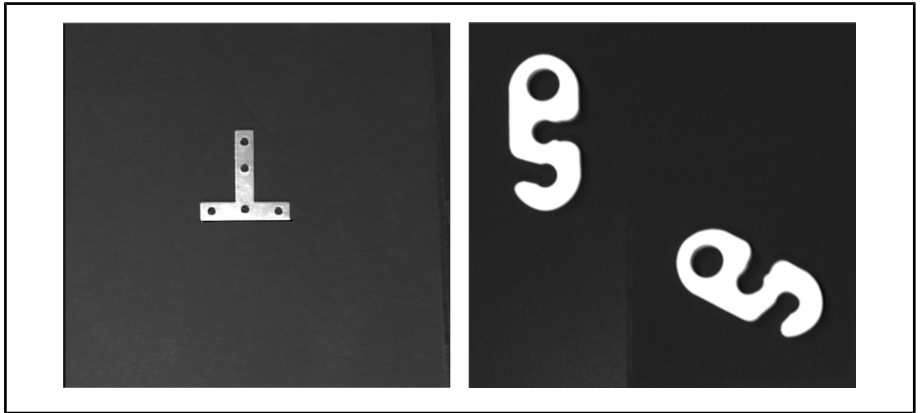
To set a template reference, use **MmetControl()** with **M_TEMPLATE_REFERENCE_ID**. In this case, you can draw the results of features and tolerances extracted from that template reference. To do so, you will need to:

1. Inquire **M_TEMPLATE_REFERENCE_SIZE_X** and **M_TEMPLATE_REFERENCE_SIZE_Y** to allocate an appropriate destination buffer.
2. Pass the metrology context identifier to **MmetDraw()**.
3. Draw the template reference (**M_DRAW_TEMPLATE_REFERENCE**).
4. Draw the results.

Note that the template reference does not in any way affect metrology calculations.

Metrology example

The metrology example *Mmet.cpp* defines the features and geometric tolerances of a metrology template. The template is used to calculate the features and verify the geometric tolerances on the following parts.



The example also reports the status of the geometric tolerances and retrieves precise feature measurements.

```

/*****
/*
 * File name: Mmet.cpp
 *
 * Synopsis: This program uses the MIL Metrology module to measure gemetric
 *           features and to validate tolerance relationships between features.
 */

#include <mil.h>

/* Example selection. */
#define RUN_SIMPLE_IMAGE_EXAMPLE      M_YES
#define RUN_COMPLETE_IMAGE_EXAMPLE    M_YES

/* Example functions declarations. */
void SimpleImageExample(MIL_ID MilSystem, MIL_ID MilDisplay);
void CompleteImageExample(MIL_ID MilSystem, MIL_ID MilDisplay);

/*****
Main.
*****/
int MosMain(void)

```

```

{
    MIL_ID MilApplication,      /* Application identifier. */
        MilSystem,            /* System Identifier.      */
        MilDisplay;           /* Display identifier.     */

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

    /* Print module name. */
    MosPrintf(MIL_TEXT("\nMETROLOGY MODULE:\n"));
    MosPrintf(MIL_TEXT("-----\n\n"));

    #if (RUN_SIMPLE_IMAGE_EXAMPLE)
    SimpleImageExample(MilSystem, MilDisplay);
    #endif

    #if (RUN_COMPLETE_IMAGE_EXAMPLE)
    CompleteImageExample(MilSystem, MilDisplay);
    #endif

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

    return 0;
}

/*****
Simple example.
*****/
/* Source MIL image file specification. */
#define METROL_SIMPLE_IMAGE_FILE M_IMAGE_PATH MIL_TEXT("SingleModel.mim")

/* Region parameters */
#define TOP_RING_POSITION_X      240
#define TOP_RING_POSITION_Y      155
#define TOP_RING_START_RADIUS    2
#define TOP_RING_END_RADIUS      15

#define MIDDLE_RING_POSITION_X   240
#define MIDDLE_RING_POSITION_Y   190
#define MIDDLE_RING_START_RADIUS 2
#define MIDDLE_RING_END_RADIUS   15

#define BOTTOM_RECT_POSITION_X    320
#define BOTTOM_RECT_POSITION_Y    265
#define BOTTOM_RECT_WIDTH         170
#define BOTTOM_RECT_HEIGHT        20
#define BOTTOM_RECT_ANGLE         180

/* Tolerance parameters */
#define PERPENDICULARITY_MIN      0.5
#define PERPENDICULARITY_MAX      0.5

```

```

/* Color definitions */
#define FAIL_COLOR      M_RGB888(255,0,0)
#define PASS_COLOR      M_RGB888(0,255,0)
#define REGION_COLOR    M_RGB888(0,100,255)
#define FEATURE_COLOR    M_RGB888(255,0,255)

void SimpleImageExample(MIL_ID MilSystem, MIL_ID MilDisplay)
{
    MIL_ID MilImage,           /* Image buffer identifier. */
    MilOverlayImage,          /* Overlay image.           */
    MilMetroContext,           /* Metrology Context        */
    MilMetroResult;           /* Metrology Result         */

    MIL_DOUBLE  Status,
               Value;

    MIL_INT FeatureIndexForTopConstructedPoint    =M_FEATURE_INDEX(1);
    MIL_INT FeatureIndexForMiddleConstructedPoint =M_FEATURE_INDEX(2);
    MIL_INT FeatureIndexForConstructedSegment[2]  ={M_FEATURE_INDEX(3),M_FEATURE_INDEX(4)};
    MIL_INT FeatureIndexForTolerance[2]           ={M_FEATURE_INDEX(5),M_FEATURE_INDEX(6)};

    /* Clear the display */
    MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

    /* Restore and display the source image. */
    MbufRestore(METROL_SIMPLE_IMAGE_FILE, MilSystem, &MilImage);
    MdispSelect(MilDisplay, MilImage);

    /* Prepare for overlay annotation. */
    MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
    MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

    /* Allocate metrology context and result. */
    MmetAlloc(MilSystem, M_DEFAULT, &MilMetroContext);
    MmetAllocResult(MilSystem, M_DEFAULT, &MilMetroResult);

    /* Add a first measured circle feature to context and set its search region */
    MmetAddFeature(MilMetroContext, M_MEASURED, M_CIRCLE, M_DEFAULT,
                  M_DEFAULT, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);

    MmetSetRegion(MilMetroContext, M_FEATURE_INDEX(1), M_DEFAULT, M_RING,
                  TOP_RING_POSITION_X, TOP_RING_POSITION_Y, TOP_RING_START_RADIUS,
                  TOP_RING_END_RADIUS, M_NULL, M_NULL);

    /* Add a second measured circle feature to context and set its search region */
    MmetAddFeature(MilMetroContext, M_MEASURED, M_CIRCLE, M_DEFAULT,
                  M_DEFAULT, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);

    MmetSetRegion(MilMetroContext, M_FEATURE_INDEX(2), M_DEFAULT, M_RING,
                  MIDDLE_RING_POSITION_X, MIDDLE_RING_POSITION_Y,
                  MIDDLE_RING_START_RADIUS, MIDDLE_RING_END_RADIUS, M_NULL, M_NULL);

    /* Add a first constructed point feature to context */

```

```

MmetAddFeature(MilMetroContext, M_CONSTRUCTED, M_POINT, M_DEFAULT,
               M_CENTER, &FeatureIndexForTopConstructedPoint, M_NULL, 1, M_DEFAULT);

/* Add a second constructed point feature to context */
MmetAddFeature(MilMetroContext, M_CONSTRUCTED, M_POINT, M_DEFAULT,
               M_CENTER, &FeatureIndexForMiddleConstructedPoint, M_NULL, 1, M_DEFAULT);

/* Add a constructed segment feature to context passing through the two points */
MmetAddFeature(MilMetroContext, M_CONSTRUCTED, M_SEGMENT, M_DEFAULT,
               M_CONSTRUCTION, FeatureIndexForConstructedSegment, M_NULL, 2, M_DEFAULT);

/* Add a first segment feature to context and set its search region */
MmetAddFeature(MilMetroContext, M_MEASURED, M_SEGMENT, M_DEFAULT,
               M_DEFAULT, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);

MmetSetRegion(MilMetroContext, M_FEATURE_INDEX(6), M_DEFAULT, M_RECTANGLE,
              BOTTOM_RECT_POSITION_X, BOTTOM_RECT_POSITION_Y, BOTTOM_RECT_WIDTH,
              BOTTOM_RECT_HEIGHT, BOTTOM_RECT_ANGLE, M_NULL);

/* Add perpendicularity tolerance */
MmetAddTolerance(MilMetroContext, M_PERPENDICULARITY, M_DEFAULT, PERPENDICULARITY_MIN,
                PERPENDICULARITY_MAX, FeatureIndexForTolerance, M_NULL, 2, M_DEFAULT);

/* Calculate */
MmetCalculate(MilMetroContext, MilImage, MilMetroResult, M_DEFAULT);

/* Draw region */
MgraColor(M_DEFAULT, REGION_COLOR);
MmetDraw(M_DEFAULT, MilMetroResult, MilOverlayImage, M_DRAW_REGION, M_DEFAULT,
          M_DEFAULT);

MosPrintf(MIL_TEXT("Regions used to calculate measured features:\n"));
MosPrintf(MIL_TEXT("- two measured circles\n"));
MosPrintf(MIL_TEXT("- one measured segment\n\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Clear the overlay to transparent. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

MgraColor(M_DEFAULT, FEATURE_COLOR);
MmetDraw(M_DEFAULT, MilMetroResult, MilOverlayImage, M_DRAW_FEATURE, M_DEFAULT,
          M_DEFAULT);

MosPrintf(MIL_TEXT("Calculated features:\n"));

MmetGetResult(MilMetroResult, M_FEATURE_INDEX(1), M_RADIUS, &Value);
MosPrintf(MIL_TEXT("- first measured circle: radius=%.2f\n"), Value);

MmetGetResult(MilMetroResult, M_FEATURE_INDEX(2), M_RADIUS, &Value);
MosPrintf(MIL_TEXT("- second measured circle: radius=%.2f\n"), Value);

MmetGetResult(MilMetroResult, M_FEATURE_INDEX(5), M_LENGTH, &Value);
MosPrintf(MIL_TEXT("- constructed segment between the two circle centers: ")
          MIL_TEXT("length=%.2f\n\n"), Value);

```

```

MmetGetResult(MilMetro1Result, M_FEATURE_INDEX(6), M_LENGTH, &Value);
MosPrintf(MIL_TEXT("- measured segment: length=%.2f\n"), Value);

MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Get angularity tolerance status and value*/
MmetGetResult(MilMetro1Result, M_TOLERANCE_INDEX(0), M_STATUS, &Status);
MmetGetResult(MilMetro1Result, M_TOLERANCE_INDEX(0), M_VALUE, &Value);

if(Status==M_PASS)
{
    MgraColor(M_DEFAULT, PASS_COLOR);
    MosPrintf(MIL_TEXT("Perpendicularity between the two segments: %.2f ")
        MIL_TEXT("degrees.\n\n"), Value);
}
else
{
    MgraColor(M_DEFAULT, FAIL_COLOR);
    MosPrintf(MIL_TEXT("Perpendicularity between the two segments - Fail.\n\n"));
}
MmetDraw(M_DEFAULT, MilMetro1Result, MilOverlayImage, M_DRAW_TOLERANCE,
    M_TOLERANCE_INDEX(0), M_DEFAULT);

MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Free all allocations. */
MmetFree(MilMetro1Result);
MmetFree(MilMetro1Context);
MbufFree(MilImage);
}

/*****
Complete example.
*****/
/* Source MIL image, calibration and model finder context file specification. */
#define METROL_CALIBRATION_FILE    M_IMAGE_PATH MIL_TEXT("Hook.mca")
#define METROL_COMPLETE_IMAGE_FILE M_IMAGE_PATH MIL_TEXT("Hook.tif")
#define METROL_MODEL_FINDER_FILE   M_IMAGE_PATH MIL_TEXT("Hook.mmf")

/* Region parameters */
#define CIRCLE1_LABEL              1
#define RING1_POSITION_X          1.10
#define RING1_POSITION_Y          0.80
#define RING1_START_RADIUS        0.20
#define RING1_END_RADIUS          0.50

#define CIRCLE2_LABEL              2
#define RING2_POSITION_X          1.10
#define RING2_POSITION_Y          3.00
#define RING2_START_RADIUS        0.10

```



```

#define RING2_END_RADIUS          0.40

#define SEGMENT1_LABEL            3
#define RECT1_POSITION_X          0.10
#define RECT1_POSITION_Y          2.40
#define RECT1_WIDTH                1.40
#define RECT1_HEIGHT              0.30
#define RECT1_ANGLE               90.0

#define SEGMENT2_LABEL            4
#define RECT2_POSITION_X          0.90
#define RECT2_POSITION_Y          2.80
#define RECT2_WIDTH                0.40
#define RECT2_HEIGHT              0.20
#define RECT2_ANGLE               165.0

#define POINT1_LABEL              5
#define SEG1_START_X              1.60
#define SEG1_START_Y              1.50
#define SEG1_END_X                1.60
#define SEG1_END_Y                2.40

/* Tolerance parameters */
#define MIN_DISTANCE_LABEL        1
#define ANGULARITY_LABEL          2
#define MAX_DISTANCE_LABEL        3

#define MIN_DISTANCE_VALUE_MIN    1.40
#define MIN_DISTANCE_VALUE_MAX    1.60
#define MAX_DISTANCE_VALUE_MIN    0.40
#define MAX_DISTANCE_VALUE_MAX    0.60
#define ANGULARITY_VALUE_MIN      65.0
#define ANGULARITY_VALUE_MAX      75.0

void CompleteImageExample(MIL_ID MilSystem, MIL_ID MilDisplay)
{
    MIL_ID MilImage,                /* Image buffer identifier. */
    MilOverlayImage,               /* Overlay image. */
    MilCalibration,                /* Calibration object */
    MilMetroContext,               /* Metrology Context */
    MilMetroResult,                /* Metrology Result */
    MilModelFinderContext,         /* Model Finder Context */
    MilModelFinderResult;          /* Model Finder Result */

    MIL_DOUBLE Status,
        Value;

    MIL_INT MinDistanceFeatureLabels[2] = {CIRCLE1_LABEL, CIRCLE2_LABEL};
    MIL_INT AngularityFeatureLabels[2] = {SEGMENT1_LABEL, SEGMENT2_LABEL};
    MIL_INT MaxDistanceFeatureLabels[2] = {POINT1_LABEL, POINT1_LABEL};
    MIL_INT MaxDistanceFeatureIndices[2] = {0, 1};

    /* Clear the display */

```

```

MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

/* Restore and display the source image. */
MbufRestore(METROL_COMPLETE_IMAGE_FILE, MilSystem, &MilImage);
MdispSelect(MilDisplay, MilImage);

/* Restore and associate calibration object to source image */
McalRestore(METROL_CALIBRATION_FILE, MilSystem, M_DEFAULT, &MilCalibration);
McalAssociate(MilCalibration, MilImage, M_DEFAULT);

/* Prepare for overlay annotation. */
MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

/* Allocate metrology context and result. */
MmetAlloc(MilSystem, M_DEFAULT, &MilMetroContext);
MmetAllocResult(MilSystem, M_DEFAULT, &MilMetroResult);

/* Add a first measured circle feature to context and set its search region */
MmetAddFeature(MilMetroContext, M_MEASURED, M_CIRCLE, CIRCLE1_LABEL,
               M_DEFAULT, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);

MmetSetRegion(MilMetroContext, M_FEATURE_LABEL(CIRCLE1_LABEL), M_DEFAULT, M_RING,
              RING1_POSITION_X, RING1_POSITION_Y, RING1_START_RADIUS, RING1_END_RADIUS,
              M_NULL, M_NULL);

/* Add a second measured circle feature to context and set its search region */
MmetAddFeature(MilMetroContext, M_MEASURED, M_CIRCLE, CIRCLE2_LABEL,
               M_DEFAULT, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);

MmetSetRegion(MilMetroContext, M_FEATURE_LABEL(CIRCLE2_LABEL), M_DEFAULT, M_RING,
              RING2_POSITION_X, RING2_POSITION_Y, RING2_START_RADIUS, RING2_END_RADIUS,
              M_NULL, M_NULL);

/* Add a first measured segment feature to context and set its search region */
MmetAddFeature(MilMetroContext, M_MEASURED, M_SEGMENT, SEGMENT1_LABEL,
               M_DEFAULT, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);

MmetSetRegion(MilMetroContext, M_FEATURE_LABEL(SEGMENT1_LABEL), M_DEFAULT, M_RECTANGLE,
              RECT1_POSITION_X, RECT1_POSITION_Y, RECT1_WIDTH, RECT1_HEIGHT,
              RECT1_ANGLE, M_NULL);

MmetControl(MilMetroContext, M_FEATURE_LABEL(SEGMENT1_LABEL), M_EDGE_ANGLE_RANGE, 10);

/* Add a second measured segment feature to context and set its search region */
MmetAddFeature(MilMetroContext, M_MEASURED, M_SEGMENT, SEGMENT2_LABEL,
               M_INNER_FIT, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);

MmetSetRegion(MilMetroContext, M_FEATURE_LABEL(SEGMENT2_LABEL), M_DEFAULT, M_RECTANGLE,
              RECT2_POSITION_X, RECT2_POSITION_Y, RECT2_WIDTH, RECT2_HEIGHT,
              RECT2_ANGLE, M_NULL);

```

```

/* Add a measured point feature to context and set its search region */
MmetAddFeature(MilMetroContext, M_MEASURED, M_POINT, POINT1_LABEL,
               M_DEFAULT, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);

MmetSetRegion(MilMetroContext, M_FEATURE_LABEL(POINT1_LABEL), M_DEFAULT, M_SEGMENT,
              SEG1_START_X, SEG1_START_Y, SEG1_END_X, SEG1_END_Y,
              M_NULL, M_NULL);

MmetControl(MilMetroContext, M_FEATURE_LABEL(POINT1_LABEL), M_FILTER_MODE, M_KERNEL);

/*Set the polarity and the maximum number of points to detect along the segment region*/
MmetControl(MilMetroContext, M_FEATURE_LABEL(POINT1_LABEL),
            M_EDGE_RELATIVE_ANGLE, M_SAME_OR_REVERSE);
MmetControl(MilMetroContext, M_FEATURE_LABEL(POINT1_LABEL), M_NUMBER_MAX, 2);

/* Add minimum distance tolerance */
MmetAddTolerance(MilMetroContext, M_DISTANCE_MIN, MIN_DISTANCE_LABEL,
                 MIN_DISTANCE_VALUE_MIN, MIN_DISTANCE_VALUE_MAX,
                 MinDistanceFeatureLabels, M_NULL, 2, M_DEFAULT);

/* Add angularity tolerance */
MmetAddTolerance(MilMetroContext, M_ANGULARITY, ANGULARITY_LABEL,
                 ANGULARITY_VALUE_MIN, ANGULARITY_VALUE_MAX,
                 AngularityFeatureLabels, M_NULL, 2, M_DEFAULT);

/* Add maximum distance tolerance */
MmetAddTolerance(MilMetroContext, M_DISTANCE_MAX, MAX_DISTANCE_LABEL,
                 MAX_DISTANCE_VALUE_MIN, MAX_DISTANCE_VALUE_MAX,
                 MaxDistanceFeatureLabels, MaxDistanceFeatureIndices, 2, M_DEFAULT);

/* Calculate */
MmetCalculate(MilMetroContext, MilImage, MilMetroResult, M_DEFAULT);

/* Draw features */
MgraColor(M_DEFAULT, REGION_COLOR);
MmetDraw(M_DEFAULT, MilMetroResult, MilOverlayImage, M_DRAW_REGION, M_DEFAULT,
          M_DEFAULT);

MosPrintf(MIL_TEXT("Regions used to calculate measured features:\n"));
MosPrintf(MIL_TEXT("- two measured circle features\n"));
MosPrintf(MIL_TEXT("- two measured segment features\n"));
MosPrintf(MIL_TEXT("- one measured points feature\n\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Clear the overlay to transparent. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

MgraColor(M_DEFAULT, FEATURE_COLOR);
MmetDraw(M_DEFAULT, MilMetroResult, MilOverlayImage, M_DRAW_FEATURE, M_DEFAULT,
          M_DEFAULT);

MosPrintf(MIL_TEXT("Calculated features:\n"));

```

```

MmetGetResult(MilMetro1Result, M_FEATURE_LABEL(CIRCLE1_LABEL), M_RADIUS,      &Value);
MosPrintf(MIL_TEXT("- first measured circle: radius=%.2fmm\n"), Value);

MmetGetResult(MilMetro1Result, M_FEATURE_LABEL(CIRCLE2_LABEL), M_RADIUS,      &Value);
MosPrintf(MIL_TEXT("- second measured circle: radius=%.2fmm\n"), Value);

MmetGetResult(MilMetro1Result, M_FEATURE_LABEL(SEGMENT1_LABEL), M_LENGTH,      &Value);
MosPrintf(MIL_TEXT("- first measured segment: length=%.2fmm\n"), Value);

MmetGetResult(MilMetro1Result, M_FEATURE_LABEL(SEGMENT2_LABEL), M_LENGTH,      &Value);
MosPrintf(MIL_TEXT("- second measured segment: length=%.2fmm\n"), Value);

MosPrintf(MIL_TEXT("- two measured points\n\n"));

MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Get angularity tolerance status and value */
MmetGetResult(MilMetro1Result, M_TOLERANCE_LABEL(ANGULARITY_LABEL), M_STATUS, &Status);
MmetGetResult(MilMetro1Result, M_TOLERANCE_LABEL(ANGULARITY_LABEL), M_VALUE, &Value);

if(Status==M_PASS)
{
    MgraColor(M_DEFAULT, PASS_COLOR);
    MosPrintf(MIL_TEXT("Angularity value: %.2f degrees.\n"), Value);
}
else
{
    MgraColor(M_DEFAULT, FAIL_COLOR);
    MosPrintf(MIL_TEXT("Angularity value - Fail.\n"));
}
MmetDraw(M_DEFAULT, MilMetro1Result, MilOverlayImage, M_DRAW_TOLERANCE,
          M_TOLERANCE_LABEL(ANGULARITY_LABEL), M_DEFAULT);

/* Get min distance tolerance status and value */
MmetGetResult(MilMetro1Result, M_TOLERANCE_LABEL(MIN_DISTANCE_LABEL), M_STATUS,
              &Status);
MmetGetResult(MilMetro1Result, M_TOLERANCE_LABEL(MIN_DISTANCE_LABEL), M_VALUE,
              &Value);

if(Status==M_PASS)
{
    MgraColor(M_DEFAULT, PASS_COLOR);
    MosPrintf(MIL_TEXT("Min distance tolerance value: %.2f mm.\n"), Value);
}
else
{
    MgraColor(M_DEFAULT, FAIL_COLOR);
    MosPrintf(MIL_TEXT("Min distance tolerance value - Fail.\n"));
}
MmetDraw(M_DEFAULT, MilMetro1Result, MilOverlayImage, M_DRAW_TOLERANCE,
          M_TOLERANCE_LABEL(MIN_DISTANCE_LABEL), M_DEFAULT);

```

```

/* Get max distance tolerance status and value */
MmetGetResult(MilMetroResult, M_TOLERANCE_LABEL(MAX_DISTANCE_LABEL), M_STATUS,
                                                       &Status);
MmetGetResult(MilMetroResult, M_TOLERANCE_LABEL(MAX_DISTANCE_LABEL), M_VALUE,
                                                       &Value);

if(Status==M_PASS)
{
    MgraColor(M_DEFAULT, PASS_COLOR);
    MosPrintf(MIL_TEXT("Max distance tolerance value: %.2f mm.\n"), Value);
}
else
{
    MgraColor(M_DEFAULT, FAIL_COLOR);
    MosPrintf(MIL_TEXT("Max distance tolerance value - Fail.\n"));
}
MmetDraw(M_DEFAULT, MilMetroResult, MilOverlayImage, M_DRAW_TOLERANCE,
          M_TOLERANCE_LABEL(MAX_DISTANCE_LABEL), M_DEFAULT);

MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Clear the overlay to transparent. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

/* Restore the model finder context and calibrate it */
MmodRestore(METROL_MODEL_FINDER_FILE, MilSystem, M_DEFAULT, &MilModelFinderContext);
MmodControl(MilModelFinderContext, 0, M_ASSOCIATED_CALIBRATION, MilCalibration);

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

/* Find object occurrence */
MmodPreprocess(MilModelFinderContext, M_DEFAULT);
MmodFind(MilModelFinderContext, MilImage, MilModelFinderResult);

/* Get number of found occurrences */
MmodGetResult(MilModelFinderResult, M_GENERAL, M_NUMBER, &Value);

if(Value==1)
{
    MmodDraw(M_DEFAULT, MilModelFinderResult, MilOverlayImage,
              M_DRAW_POSITION+M_DRAW_BOX, M_DEFAULT, M_DEFAULT);
    MosPrintf(MIL_TEXT("Found occurrence using MIL Model Finder.\n"));
    MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
    MosGetch();

    /* Clear the overlay to transparent. */
    MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

    /* Set the new context position */
    MmetSetPosition(MilMetroContext, M_DEFAULT, M_RESULT, MilModelFinderResult,

```

```

0, M_NULL, M_NULL, M_DEFAULT);

/* Calculate */
MmetCalculate(MilMetroContext, MilImage, MilMetroResult, M_DEFAULT);

/* Draw features */
MgraColor(M_DEFAULT, REGION_COLOR);
MmetDraw(M_DEFAULT, MilMetroResult, MilOverlayImage,
          M_DRAW_REGION, M_DEFAULT, M_DEFAULT);
MosPrintf(MIL_TEXT("Regions used to calculate measured ")
          MIL_TEXT("features at the new location.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Clear the overlay to transparent. */
MdispcControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

MgraColor(M_DEFAULT, FEATURE_COLOR);
MmetDraw(M_DEFAULT, MilMetroResult, MilOverlayImage,
          M_DRAW_FEATURE, M_DEFAULT, M_DEFAULT);
MosPrintf(MIL_TEXT("Calculated features.\n"));

MmetGetResult(MilMetroResult, M_FEATURE_LABEL(CIRCLE1_LABEL), M_RADIUS, &Value);
MosPrintf(MIL_TEXT("- first measured circle: radius=%.2fmm\n"), Value);

MmetGetResult(MilMetroResult, M_FEATURE_LABEL(CIRCLE2_LABEL), M_RADIUS, &Value);
MosPrintf(MIL_TEXT("- second measured circle: radius=%.2fmm\n"), Value);

MmetGetResult(MilMetroResult, M_FEATURE_LABEL(SEGMENT1_LABEL), M_LENGTH, &Value);
MosPrintf(MIL_TEXT("- first measured segment: length=%.2fmm\n"), Value);

MmetGetResult(MilMetroResult, M_FEATURE_LABEL(SEGMENT2_LABEL), M_LENGTH, &Value);
MosPrintf(MIL_TEXT("- second measured segment: length=%.2fmm\n"), Value);

MosPrintf(MIL_TEXT("- two measured points\n\n"));

MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Get angularity tolerance status and value */
MmetGetResult(MilMetroResult, M_TOLERANCE_LABEL(ANGULARITY_LABEL), M_STATUS,
              &Status);
MmetGetResult(MilMetroResult, M_TOLERANCE_LABEL(ANGULARITY_LABEL), M_VALUE,
              &Value);

/* Get angularity tolerance status and value */
MmetGetResult(MilMetroResult, M_TOLERANCE_LABEL(ANGULARITY_LABEL), M_STATUS,
              &Status);
MmetGetResult(MilMetroResult, M_TOLERANCE_LABEL(ANGULARITY_LABEL), M_VALUE,
              &Value);

if(Status==M_PASS)
{

```

```

        MgraColor(M_DEFAULT, PASS_COLOR);
        MosPrintf(MIL_TEXT("Angularity value: %.2f degrees.\n"), Value);
    }
else
{
    MgraColor(M_DEFAULT, FAIL_COLOR);
    MosPrintf(MIL_TEXT("Angularity value - Fail.\n"));
}
MmetDraw(M_DEFAULT, MilMetro1Result, MilOverlayImage, M_DRAW_TOLERANCE,
          M_TOLERANCE_LABEL(ANGULARITY_LABEL), M_DEFAULT);

/* Get min distance tolerance status and value */
MmetGetResult(MilMetro1Result, M_TOLERANCE_LABEL(MIN_DISTANCE_LABEL), M_STATUS,
              &Status);
MmetGetResult(MilMetro1Result, M_TOLERANCE_LABEL(MIN_DISTANCE_LABEL), M_VALUE,
              &Value);

if(Status==M_PASS)
{
    MgraColor(M_DEFAULT, PASS_COLOR);
    MosPrintf(MIL_TEXT("Min distance tolerance value: %.2f mm.\n"), Value);
}
else
{
    MgraColor(M_DEFAULT, FAIL_COLOR);
    MosPrintf(MIL_TEXT("Min distance tolerance value - Fail.\n"));
}
MmetDraw(M_DEFAULT, MilMetro1Result, MilOverlayImage, M_DRAW_TOLERANCE,
          M_TOLERANCE_LABEL(MIN_DISTANCE_LABEL), M_DEFAULT);

/* Get max distance tolerance status and value */
MmetGetResult(MilMetro1Result, M_TOLERANCE_LABEL(MAX_DISTANCE_LABEL), M_STATUS,
              &Status);
MmetGetResult(MilMetro1Result, M_TOLERANCE_LABEL(MAX_DISTANCE_LABEL), M_VALUE,
              &Value);

if(Status==M_PASS)
{
    MgraColor(M_DEFAULT, PASS_COLOR);
    MosPrintf(MIL_TEXT("Max distance tolerance value: %.2f mm.\n"), Value);
}
else
{
    MgraColor(M_DEFAULT, FAIL_COLOR);
    MosPrintf(MIL_TEXT("Max distance tolerance value - Fail.\n"));
}
MmetDraw(M_DEFAULT, MilMetro1Result, MilOverlayImage, M_DRAW_TOLERANCE,
          M_TOLERANCE_LABEL(MAX_DISTANCE_LABEL), M_DEFAULT);

MosPrintf(MIL_TEXT("Press <Enter> to quit.\n\n"));
MosGetch();
}
else

```

```
{
    MosPrintf(MIL_TEXT("Occurrence not found.\n"));
    MosPrintf(MIL_TEXT("Press <Enter> to quit.\n\n"));
    MosGetch();
}

/* Free all allocations. */
MmodFree(MilModelFinderContext);
MmodFree(MilModelFinderResult);
MmetFree(MilMetrolResult);
MmetFree(MilMetrolContext);
McalFree(MilCalibration);
MbufFree(MilImage);
}
```


Chapter

16

3D Reconstruction

This chapter explains how to perform laser line profiling, calibration and reconstruction with the MIL 3D Reconstruction module.

MIL 3D Reconstruction module

The MIL 3D Reconstruction module allows you to extract 3D (three-dimensional) information from 2D (two-dimensional) images of an object taken using a 3D reconstruction setup. The module supports extracting 3D information from images taken from two types of 3D reconstruction setups:

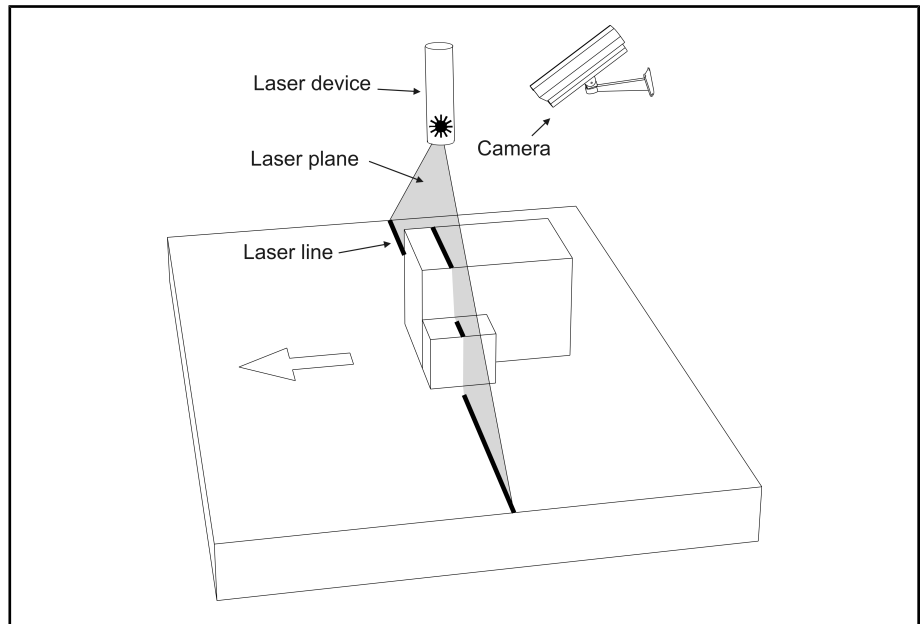
- A laser line profiling setup.
- A triangulation of points setup.

Using images taken from a 3D reconstruction setup built for laser line profiling, the module can generate a depth map or a cloud of 3D points of an object.

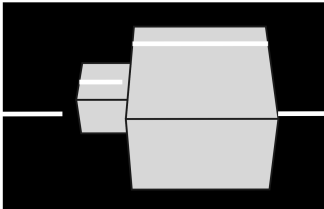
Using images taken from a 3D reconstruction setup for the triangulation of points, the module can calculate the world coordinates of a point when two or more cameras, for which you have calibration objects, are used.

Laser line profiling

A basic laser line profiling setup consists of: a device projecting a sheet of light (usually a laser diode), a camera, and a mechanism to move the object under the laser plane. For example, a conveyor belt can be used to move the object in a linear motion and at a specified speed. As an object moves under the laser plane, the camera is used to grab images of the intersection of that laser plane with the object.



As a result, multiple images are grabbed where the laser line in each image represents a "slice" of the object being scanned. The position of the laser line is then analyzed to generate a depth map. A depth map is an image where each pixel's intensity is proportional to its depth in the world.



Example of a laser line across an object.



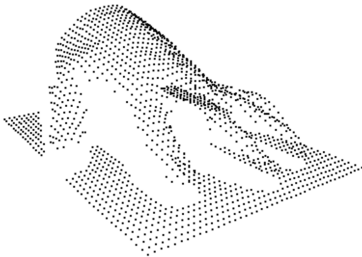
Image of example setup on the left.



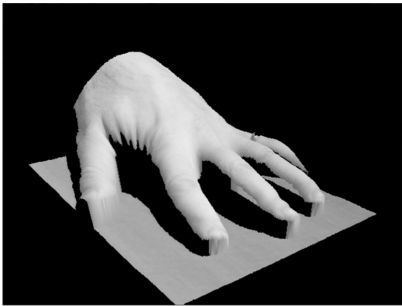
Generated depth map based on multiple images of the laser line at different positions across the object.

You can use the depth map to calculate the volume of the object or as an input to other MIL modules to find defects.

The 3D Reconstruction module can also use the position of the laser lines to generate a points cloud of the object. This is a 3D representation of the object's surface. You can further process and refine it with third-party tools to generate a mesh.



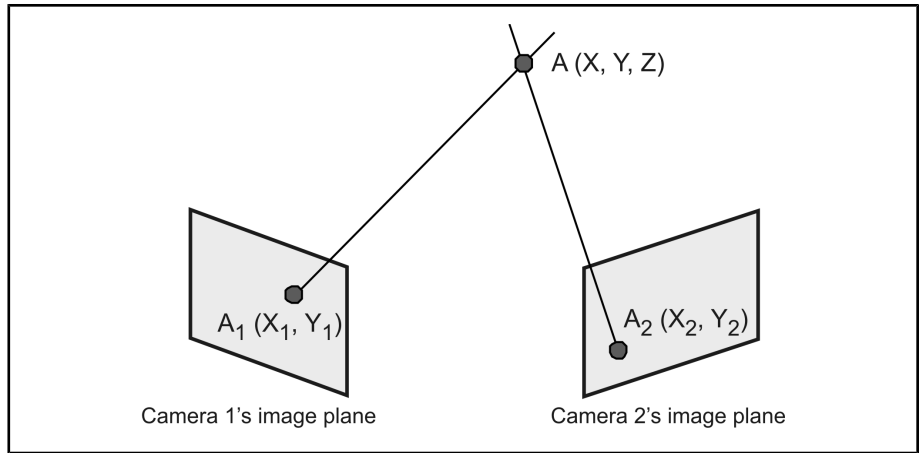
Cloud of 3d points



Mesh

Triangulation

The 3D Reconstruction module is also capable of performing stereo vision triangulation if two or more cameras, for which you have calibration objects, are used. Triangulation calculates the X-, Y-, and Z-world coordinate of a point from the point's X and Y coordinates on the image plane of each camera.



Steps to extracting a depth map using laser line profiling

The following steps provide a basic methodology for using laser line profiling with the MIL 3D Reconstruction module:

1. Allocate a 3D reconstruction context to hold your laser line profiling settings, using **M3dmapAlloc()** with **M_LASER** and the required 3D reconstruction mode.
2. Allocate a 3D reconstruction result buffer to hold the results, using **M3dmapAllocResult()**.
3. Optionally, use **M3dmapControl()** to change the settings of the context and result buffer.
4. Calibrate your 3D reconstruction setup depending on the 3D reconstruction mode.

5. For every slice required to model the target object, grab an image of a laser line at that corresponding location and then extract the laser line information from the image with **M3dmapAddScan()**.
6. Allocate a 2D image buffer for the depth map and, optionally, another 2D image buffer for the intensity map.
7. Use **M3dmapExtract()** to generate the depth map and, if required, the intensity map.
8. If in **M_CALIBRATED_CAMERA_LINEAR_MOTION** 3D reconstruction mode, you can call **M3dmapGetResult()** to obtain the coordinates of a cloud of 3D points representing your object.
9. Free all your allocated objects using **M3dmapFree()** and **McalFree()**.

Steps 1 to 4 are usually performed only once for a given 3D reconstruction setup. You can repeat steps 5 to 8 for every object you want to reconstruct. Note that for every new object being scanned, you should clear the contents of the result buffer using **M3dmapAddScan()** with **M_RESET**.

Basic concepts

The basic concepts and vocabulary conventions for the MIL 3D Reconstruction module are:

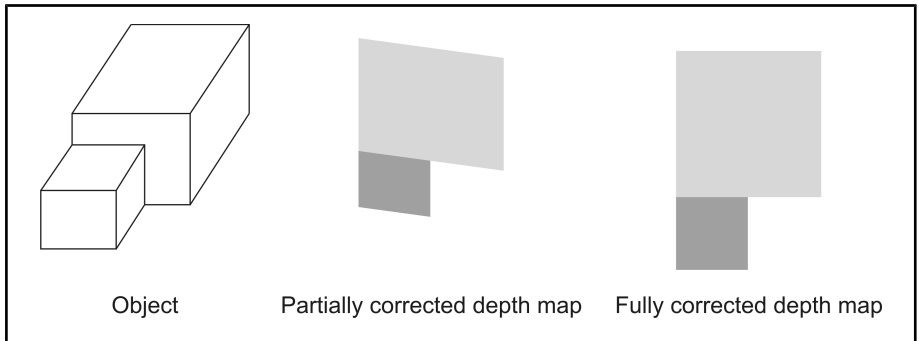
- **3D reconstruction context.** A container for information regarding the reconstruction setup, such as the 3D reconstruction calibration information, camera position, and object speed.
- **3D reconstruction setup.** A physical setup used to extract 3D information from 2D images.
- **Calibration object.** An object, allocated using **McalAlloc()**, which allows you to map pixel coordinates to real-world coordinates for a given camera setup.
- **Camera setup.** A setup which describes how the camera is positioned relative to the object and the intrinsic distortion that the camera introduces.

- **Depth.** A measurement extending from the surface of the object to the x-y plane (that is, the $z=0$ plane), in world units.
- **Depth map.** An image where the gray value of a pixel represents its depth in the world.
- **Fully corrected depth map.** A depth map where the shape of the object is free of any geometrical distortions.
- **Gap.** A range of data which is missing in the depth map because the data could not be extracted from the image at this location.
- **Intensity map.** An image where the gray value of each pixel represents the luminous intensity of the laser line at this point.
- **Laser line profiling.** A process which extracts 3D spatial information from the position and intensity of a laser line in an image.
- **Partially corrected depth map.** A depth map where the shape is not corrected.
- **Triangulation.** A process which calculates a point's (X, Y, Z) coordinates in the world if it is seen by at least two calibrated camera setups.

Laser line profiling requirements

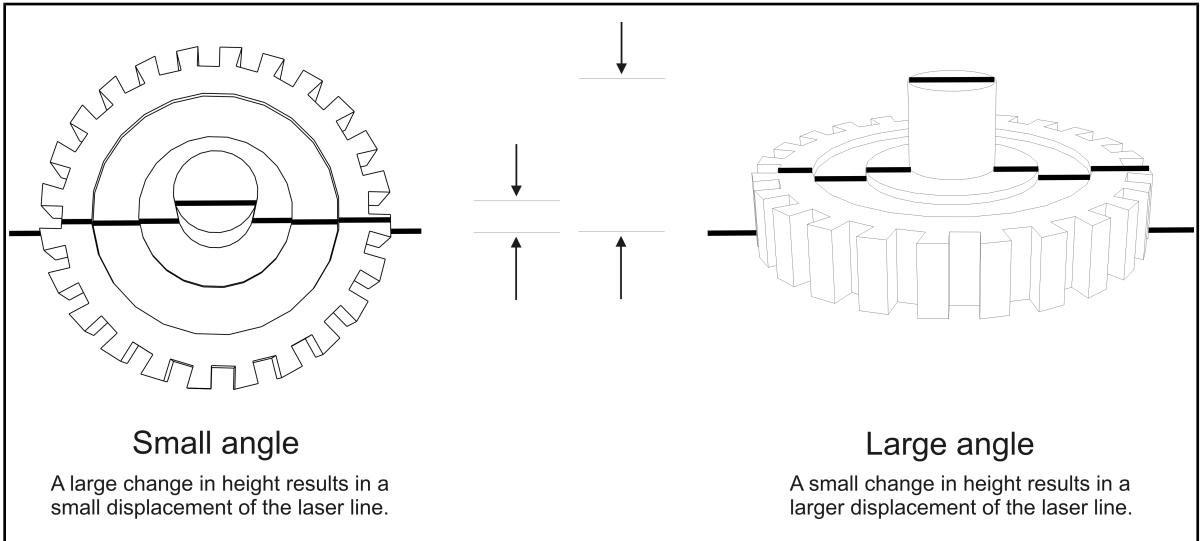
The objective of laser line profiling is to extract a laser line's position in an image and to use this information to generate a depth map. The resulting depth map is a gray image whereby each pixel's gray value is proportional with the depth at this point.

To extract this information, some calibration of the 3D reconstruction setup needs to be performed. The number of constraints and amount of setup required to perform the calibration depend on the 3D reconstruction mode, selected during context allocation (**M3dmapAlloc()**). In general, if you only require a partially corrected depth map, that is, a depth map where the gray level is corrected, but the shape of the object is not, then the constraints and amount of setup are less than if a fully corrected depth map was to be generated.



Camera angle requirement

Before starting the calibration process of your 3D reconstruction setup, ensure that the angle of the camera is sufficient with respect to the Z-axis to capture the laser line for each slice of the object. The angle of the camera should be large enough so that a small change in height is enough to displace the laser line in the grabbed image. The larger the displacement of the laser line in the grabbed image, the more accurate your results will be.



To get very accurate results, you want the camera to be at an angle where the laser line, when located on the highest feature of your object, is located at the top of your laser line image buffer and, when located on the XY plane ($Z=0$) plane, it is located at the bottom of your laser line image buffer. In general, this means a large angle. However, when your camera angle is large, you increase the number of locations in the image where the object's geometry can occlude the laser line from the camera's view.

Partially corrected depth map requirements

If you are creating a partially corrected depth map, the following requirements must be met:

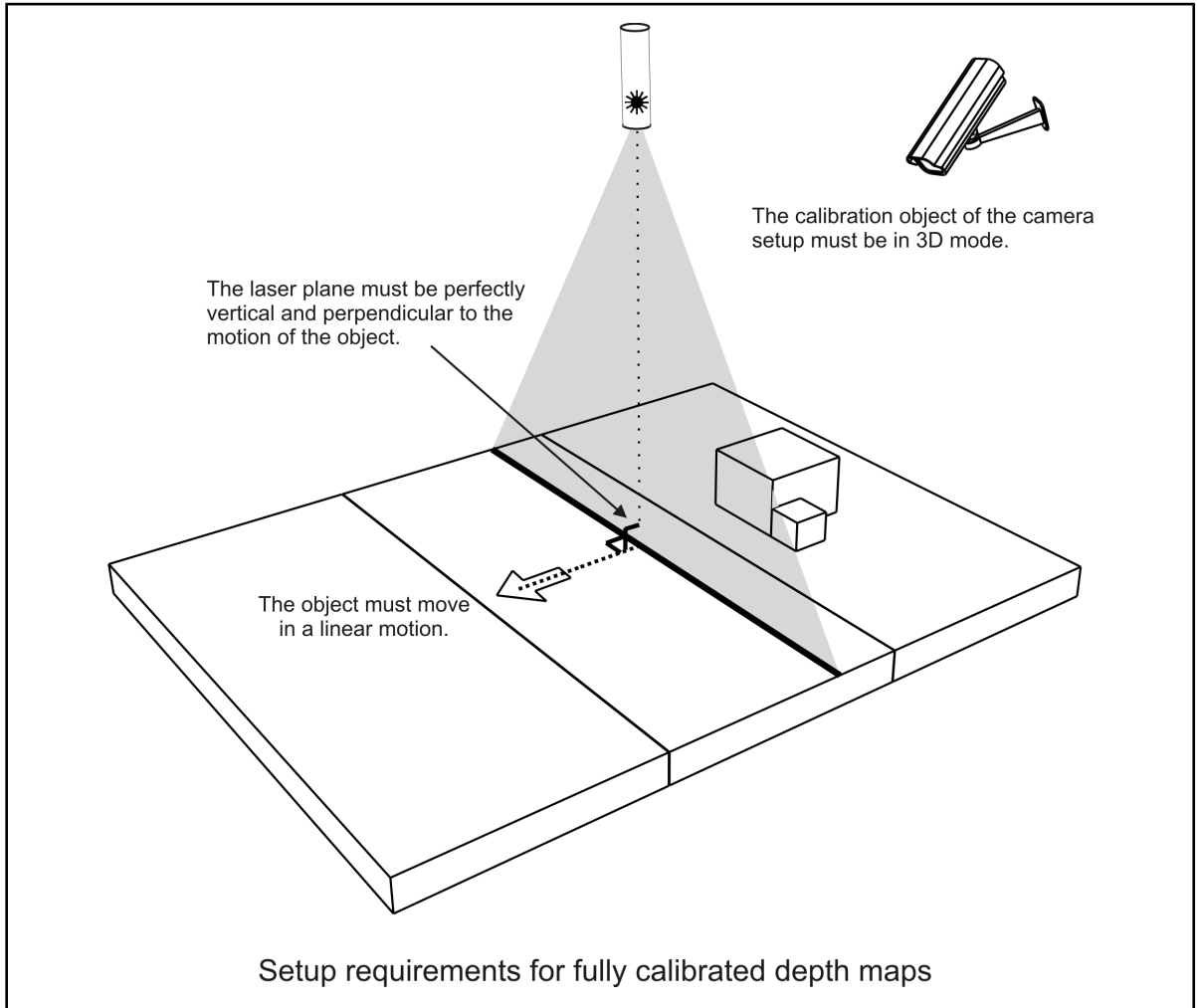
- The angle of the camera must be sufficient. For details, see the *Camera angle requirement* subsection in the *Laser line profiling* section in *Chapter 16: 3D Reconstruction*.
- You need to calibrate the 3D reconstruction setup, but not its camera setup. To perform the required calibration operations, see the *Calibrating for a partially corrected depth map* subsection in the *Calibrating your reconstruction setup* section in *Chapter 16: 3D Reconstruction*.

Fully corrected depth map requirements

If you are creating a fully corrected depth map, the following requirements must be met:

- The angle of the camera must be sufficient. For details, see the *Camera angle requirement* subsection in the *Laser line profiling* section in *Chapter 16: 3D Reconstruction*.
- You must calibrate the camera setup of the 3D reconstruction setup (in 3D mode), as well as calibrate the rest of the 3D reconstruction setup. To perform the required calibration operations, see the *Calibrating for a fully corrected depth map* subsection in the *Calibrating your reconstruction setup* section in *Chapter 16: 3D Reconstruction*.
- The object must move in a linear motion under the laser plane.
- The linear motion of the object must be perpendicular to the laser plane; that is, the laser plane must be perfectly vertical to the XY plane ($Z=0$ plane) on which the object is resting.

The displacement of the object between each grab of the laser line must be specified. However, the distance between each displacement does not need to be the same; only the direction needs to be the same.



Calibrating your 3D reconstruction setup

After ensuring that your 3D reconstruction setup meets the requirements described in the *Fully corrected depth map requirements* subsection in the *Laser line profiling* section in *Chapter 16: 3D Reconstruction*, you must calibrate it.

Calibrating your 3D reconstruction setup is a mandatory and important step to perform, before generating a depth map. The steps to perform are different depending on the 3D reconstruction mode (partially corrected or fully corrected). In both cases, the objective is to model the specifics of your 3D reconstruction setup inside the 3D reconstruction context.

Calibrating for a partially corrected depth map

For **M_DEPTH_CORRECTION** 3D reconstruction mode, you need to calibrate the 3D reconstruction setup, but it is not necessary to calibrate its camera setup.

You must specify certain characteristics of the laser line and the laser device. You must specify the laser line's minimum intensity, orientation, and peak width, using **M3dmapControl()** with **M_MIN_INTENSITY**, **M_ORIENTATION**, and **M_PEAK_WIDTH** respectively.

You must associate a gray value to the position of the laser line grabbed at different heights. To do so, perform the following:

1. Grab an image of the laser line on a planar object and at a known height.
2. Associate the gray value to use for the depth map for this laser line position using **M3dmapControl()** with **M_CORRECTED_DEPTH**. For an 8-bit depth map, this value is restricted between 0 and 254. For a 16-bit depth map, this value is restricted between 0 and 65534. The specified gray value must be relative to the height of the object you are scanning. For example, you should specify a gray value of 254, for an 8-bit depth map image buffer, if the grabbed laser line is located on a plane with a height equal to the highest point of the object.
3. Extract the laser line using **M3dmapAddScan()**.

4. Repeat steps 1 to 3 for every plane height that you want to use to calibrate your 3D reconstruction setup. A minimum of 2 plane heights are required to calibrate your 3D reconstruction setup. However, your results will be more accurate if you calibrate using more plane heights.
5. Call **M3dmapCalibrate()**.

For heights between those defined during calibration, the module will perform linear interpolation between the gray values that you specify.

- ❖ Note that the module will not be able to determine heights smaller or larger than those specified during calibration (that is, the module will not extrapolate).

The following code snippet is an example of the steps required to calibrate in **M_DEPTH_CORRECTION** 3D reconstruction mode.

```
/* Allocate 3dmap objects. */
M3dmapAlloc(MilSystemId, M_LASER, M_DEPTH_CORRECTION, &LaserId);
M3dmapAllocResult(MilSystemId, M_LASER_DATA, M_DEFAULT, &ScanId);

/* Set settings with M3dmapControl() if needed. */
/* ... */

for (n = 0; n < NB_CALIBRATION_IMAGES; n++)
{
    /* Grab next image. */
    /* MbufLoad() can also be used to load an image. */
    MdigGrab(DigId, MilImageId);

    /* Set desired corrected depth of next calibration plane. */
    /* CorrectedDepths is an array which has been filled with known depths. */
    M3dmapControl(LaserId, M_DEFAULT, M_CORRECTED_DEPTH, CorrectedDepths[n]);

    /* Analyze the image to extract laser line. */
    M3dmapAddScan(LaserId, ScanId, MilImageId, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);
}

/* Calibrate the laser profiling context using plane of known heights. */
M3dmapCalibrate(LaserId, ScanId, M_NULL, M_DEFAULT);
```

- ❖ Note that you can use **M3dmapInquire()** with **M_CALIBRATION_STATUS** to ensure that the calibration of your 3D reconstruction setup has been successfully performed.

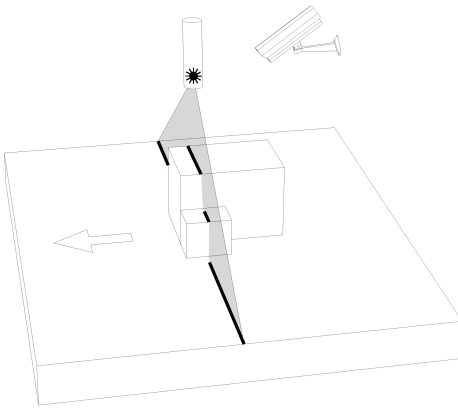
Calibrating for a fully corrected depth map

For **M_CALIBRATED_CAMERA_LINEAR_MOTION** 3D reconstruction mode, you must calibrate the entire 3D reconstruction setup, including its camera setup.

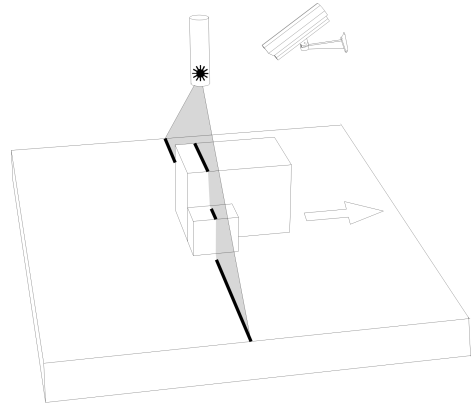
You must specify certain characteristics of the laser line and the laser device. You must specify the laser line's minimum intensity, orientation, and peak width, using **M3dmapControl()** with **M_MIN_INTENSITY**, **M_ORIENTATION**, and **M_PEAK_WIDTH** respectively. Then perform the following:

1. Allocate a calibration object using **McalAlloc()** with **M_TSAI_BASED** or **M_3D_ROBOTICS** and calibrate your camera setup with **McalGrid()** or **McalList()**. For more details, see the *Calibrating your camera setup* section in *Chapter 5: Camera calibration*.
2. Grab an image of the laser line on the XY plane (Z=0 plane).
3. Use **M3dmapAddScan()** to extract and add the laser line information to the 3D reconstruction result buffer.
4. Call **M3dmapCalibrate()**.

After you have calibrated your 3D reconstruction setup, it is also important to set the speed, in world units per frame, at which the object is being scanned, using **M3dmapControl()** with **M_SCAN_SPEED**. Note that if your object is moving away from the camera, you must specify a negative speed.



Negative scan speed



Positive scan speed

The following code snippet is an example of the steps required to calibrate in **M_CALIBRATED_CAMERA_LINEAR_MOTION** 3D reconstruction mode.

```
/* Allocate 3dmap objects. */
M3dmapAlloc(MilSystemId, M_LASER, M_CALIBRATED_CAMERA_LINEAR_MOTION, &LaserId);
M3dmapAllocResult(MilSystemId, M_LASER_DATA, M_DEFAULT, &ScanId);

/* Set settings with M3dmapControl() if needed. */
/* ... */

/* Get image of laser line on Z=0 plane. */
/* MbufLoad can also be used to load an image. */
MdigGrab(DigId, MilImageId);

/* Calibrate laser profiling context. */
/* MilCalibrationId is the identifier of the calibration object calibrated by McalGrid() or
McalList() in a 3D mode. */
M3dmapAddScan(LaserId, ScanId, MilImageId, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);
M3dmapCalibrate(LaserId, ScanId, MilCalibrationId, M_DEFAULT);
```

- ❖ Note that you can use **McalInquire()** and **M3dmapInquire()** with **M_CALIBRATION_STATUS** to ensure that the calibration of the camera setup and 3D reconstruction setup, respectively, have been successfully performed.

Obtaining results

After calibrating your 3D reconstruction setup with **M3dmapCalibrate()** and scanning your target object with calls to **M3dmapAddScan()**, you can extract the laser line data from the result buffer to create a depth map and an intensity map with **M3dmapExtract()**.

In **M_CALIBRATED_CAMERA_LINEAR_MOTION** 3D reconstruction mode, you can also retrieve the coordinates of a cloud's points with **M3dmapGetResult()**.

See the *Calibrating your 3D reconstruction setup* section earlier in this chapter for more information on calibrating your 3D reconstruction setup.

Generating the depth map

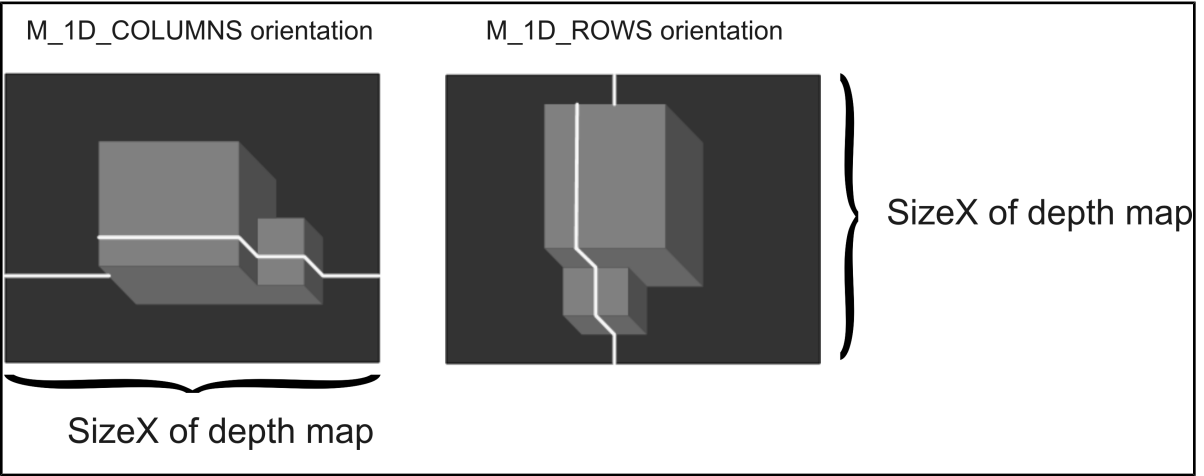
To generate the depth map, use **M3dmapExtract()**. This function will take all the accumulated laser line data stored in the result buffer and project them on the XY plane (Z=0 plane) to create the depth map. Depending on the 3D reconstruction mode, a partially corrected depth map or a fully corrected depth map will be created and stored in the image buffer specified for the depth map.

Partially corrected depth map

In **M_DEPTH_CORRECTION** 3D reconstruction mode, **M3dmapExtract()** generates a partially corrected depth map and stores it in the specified buffer.

There are restrictions on the dimensions of the image buffer used to store the generated depth map.

- The width of the depth map image buffer must match the width of the laser line image if **M3dmapControl()** with **M_ORIENTATION** is set to **M_1D_COLUMNS** or the height of the laser line image if **M3dmapControl()** with **M_ORIENTATION** is set to **M_1D_ROWS**.



- The required height of the depth map image buffer depends on the number of laser scan lines stored in the result buffer. As such, the height of the depth map image buffer must be equal to either the number of times **M3dmapAddScan()** was called to extract the laser lines or the value of **M3dmapControl()** with **M_MAX_FRAMES**, whichever is smaller.
- ❖ Note that you can call **M3dmapGetResult()** with **M_CORRECTED_DEPTH_MAP_SIZE_X** or **M_CORRECTED_DEPTH_MAP_SIZE_Y** to determine the required width and height of the depth map image buffer, respectively.

The depth map image buffer must be a 16-bit unsigned buffer.

The following code snippet shows how to generate a partially corrected depth map.

```
/* Retrieve the expected SizeX and SizeY of the depth map and type of the intensity map. */
M3dmapGetResult(ScanId, M_DEFAULT, M_CORRECTED_DEPTH_MAP_SIZE_X      +M_TYPE_MIL_INT, &SizeX
);
M3dmapGetResult(ScanId, M_DEFAULT, M_CORRECTED_DEPTH_MAP_SIZE_Y      +M_TYPE_MIL_INT, &SizeY
);
M3dmapGetResult(ScanId, M_DEFAULT, M_INTENSITY_MAP_BUFFER_TYPE        +M_TYPE_MIL_INT,
&IntensityMapType);

/* Allocate image buffers for the depth map and intensity map. */
MbufAlloc2d(MilSystemId, SizeX, SizeY, 16+M_UNSIGNED, M_IMAGE+M_PROC, &DepthMapImageId
);
MbufAlloc2d(MilSystemId, SizeX, SizeY, IntensityMapType, M_IMAGE+M_PROC, &IntensityMapImageId);

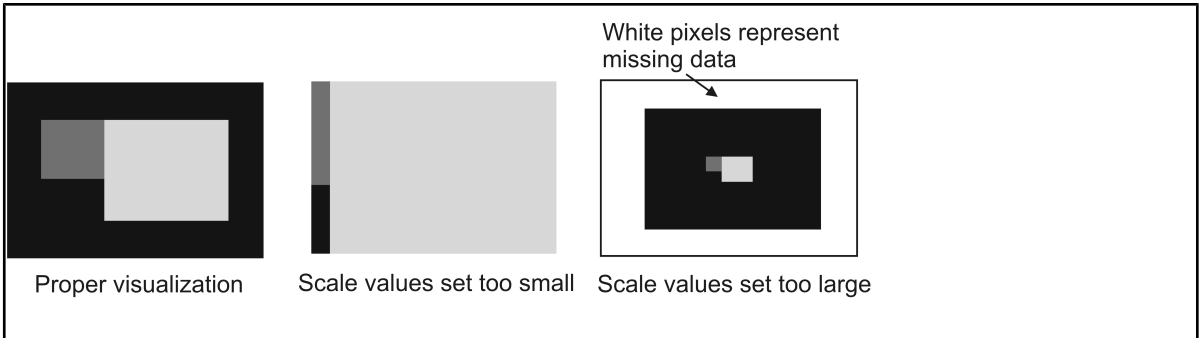
/* Generate the depth map and intensity map. */
M3dmapExtract(ScanId, DepthMapImageId, IntensityMapImageId, M_CORRECTED_DEPTH_MAP, M_DEFAULT,
M_DEFAULT);
```

Fully corrected depth map

In **M_CALIBRATED_CAMERA_LINEAR_MOTION** 3D reconstruction mode, **M3dmapExtract()** generates a fully corrected depth map and stores it in the specified buffer.

There are no restrictions on the width and height of the image buffer for a fully corrected depth map and the buffer can be either an 8- or 16-bit unsigned buffer. However, in **M_CALIBRATED_CAMERA_LINEAR_MOTION** 3D reconstruction mode, results are computed in world units and pixels in the original laser line images don't all have the same size because of perspective distortion introduced by the angle of the camera. Therefore, to properly visualize the results, you must adjust the **M_PIXEL_SIZE_X** and **M_PIXEL_SIZE_Y** scale values using

M3dmapControl(), before you call **M3dmapExtract()**. If you set the scaling too small relative to your image buffer, your object might not fit inside the depth map image buffer. Conversely, if your scale values are set too large, the object will appear very small in the image buffer.



To prevent distortion when visualizing your image, it is recommended that you set **M_PIXEL_SIZE_X** and **M_PIXEL_SIZE_Y** to the same value. To determine an appropriate value for proper visualization, use the following equations to calculate the minimum required pixel size, in X and Y, to draw the entire region in the depth map image buffer; then set **M_PIXEL_SIZE_X** and **M_PIXEL_SIZE_Y** to the largest of these values.

- Min pixel X-size = *Width of region being drawn in world units / width of buffer* .
- Min pixel Y-size = *Length of region being drawn in world units / height of buffer* .

To determine an appropriate **M_GRAY_LEVEL_SIZE_Z** scale value for proper visualization, divide the height of your object by the number of gray levels of your depth map image buffer.

For example, if you want to use an 8-bit image buffer that has a size of 500 x 200 pixels to visualize an object measuring 25 cm x 10 cm x 5 cm, set the scale values as follows:

- $\mathbf{M_PIXEL_SIZE_X} = \max(25 \text{ cm}/500 \text{ pixels}, 10 \text{ cm}/200 \text{ pixels}) = \max(0.05, 0.04) = 0.05 \text{ cm/pixel}.$
 - $\mathbf{M_PIXEL_SIZE_Y} = \mathbf{M_PIXEL_SIZE_X} = 0.05 \text{ cm/pixel}.$
 - $\mathbf{M_GRAY_LEVEL_SIZE_Z} = 5 \text{ cm}/254 \text{ gray levels} = 0.02 \text{ cm per gray level}.$
- ❖ Note that it is recommended that you specify a negative $\mathbf{M_GRAY_LEVEL_SIZE_Z}$ value. If you specify a negative $\mathbf{M_GRAY_LEVEL_SIZE_Z}$ value, you will not need to specify a value for $\mathbf{M_WORLD_OFFSET_Z}$ to visualize the results. When you specify a negative $\mathbf{M_GRAY_LEVEL_SIZE_Z}$ value, points which are higher in depth will be brighter than points located lower in depth. In the previous example, you would set $\mathbf{M_GRAY_LEVEL_SIZE_Z}$ to -0.02.

Depending on the direction that your object moves while scanning, either towards or away from the camera, you might need to specify an $\mathbf{M_WORLD_OFFSET_Y}$ offset value. In general, if the object is moving away from the camera, you do not have to specify an offset value. However, if the object is moving towards the camera, you need to specify a negative $\mathbf{M_WORLD_OFFSET_Y}$ offset. For example, if the object is moving towards the camera and it has a length of 10 cm, you would need to specify a $\mathbf{M_WORLD_OFFSET_Y}$ value of -10 or less.

All pixels in the depth map image buffer can be found in the world. That is, from a (X, Y, I) triplet of the generated depth map, where I is the intensity, you can obtain the world coordinates (X_w, Y_w, Z_w).

Once you have specified scale and offset values, you can map pixel coordinates in the depth map image to real-world coordinates. To calculate the world coordinates of a pixel from a (X, Y, I) triplet, use the following formulae:

- $\mathbf{X_w} = \mathbf{M_WORLD_OFFSET_X} + \mathbf{X} * \mathbf{M_PIXEL_SIZE_X}.$
- $\mathbf{Y_w} = \mathbf{M_WORLD_OFFSET_Y} + \mathbf{Y} * \mathbf{M_PIXEL_SIZE_Y}.$
- $\mathbf{Z_w} = \mathbf{M_WORLD_OFFSET_Z} + \mathbf{I} * \mathbf{M_GRAY_LEVEL_SIZE_Z}.$

You can obtain the pixel scale and offset values used in the formulae with **M3dmapInquire()**.

Similarly, you can calculate the region of the world (interval) your depth map image is covering with the following formulae:

- Xw range: $[M_WORLD_OFFSET_X, M_WORLD_OFFSET_X + (SizeXOfDepthMap - 1) * M_PIXEL_SIZE_X]$.
- Yw range: $[M_WORLD_OFFSET_Y, M_WORLD_OFFSET_Y + (SizeYOfDepthMap - 1) * M_PIXEL_SIZE_Y]$.

SizeXOfDepthMap and **SizeYOfDepthMap** are the width and height of your depth map image buffer, respectively.

You can obtain the pixel scale and offset values used in the previous formulae with **M3dmapInquire()**.

Generating the intensity map

In an intensity map, the gray value of each pixel represents the luminous intensity of the laser line at this point. This information can be used to determine, for example, if the surface under the laser line at this point is dark or light, or if the surface is a reflecting one.

To generate the intensity map, allocate a buffer and supply its identifier to **M3dmapExtract()**. The width and height of the intensity map buffer must be the same of the depth map. If you specify **M_NULL** as the identifier of the intensity map buffer, an intensity map is not generated.

In **M_DEPTH_CORRECTION** 3D reconstruction mode, you can determine the expected type of the intensity map buffer using **M3dmapGetResult()** with **M_INTENSITY_MAP_BUFFER_TYPE**.

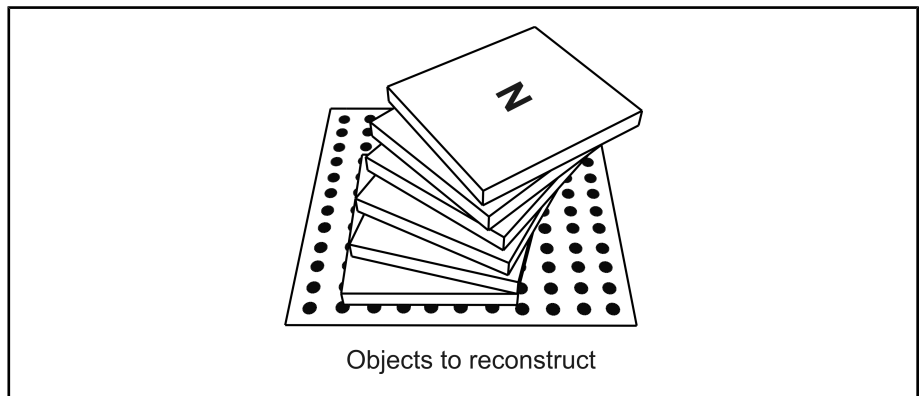
In **M_CALIBRATED_CAMERA_LINEAR_MOTION** 3D reconstruction mode, the intensity map buffer must be an 8- or 16-bit unsigned buffer depending on the value of **M_GRAY_LEVEL_SIZE_Z**.

Retrieving the coordinates of a cloud of points

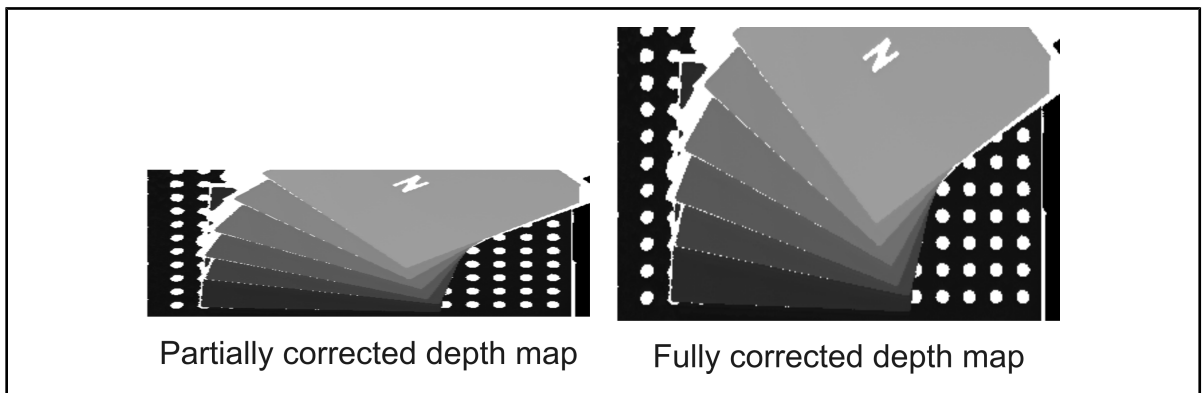
In `M_CALIBRATED_CAMERA_LINEAR_MOTION` 3D reconstruction mode, you can retrieve the coordinates of a cloud of points. These coordinates can be used as an input to a third party tool to generate a mesh of the object. To retrieve the coordinates of a cloud of points, use `M3dmapGetResult()` with `M_3D_POINTS_I`, `M_3D_POINTS_X`, `M_3D_POINTS_Y`, and `M_3D_POINTS_Z`.

Example

The following image shows a computer reconstructed scene which consists of a stack of blocks on a camera calibration grid.



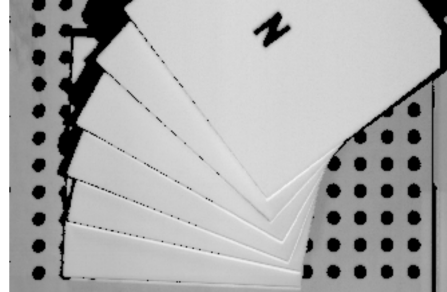
After scanning the objects to reconstruct and using `M3dmapExtract()` to generate the depth map, the following images are produced depending on whether a partially or fully corrected depth map was chosen to be created.



The following image shows the intensity map of a partially and fully corrected depth map.



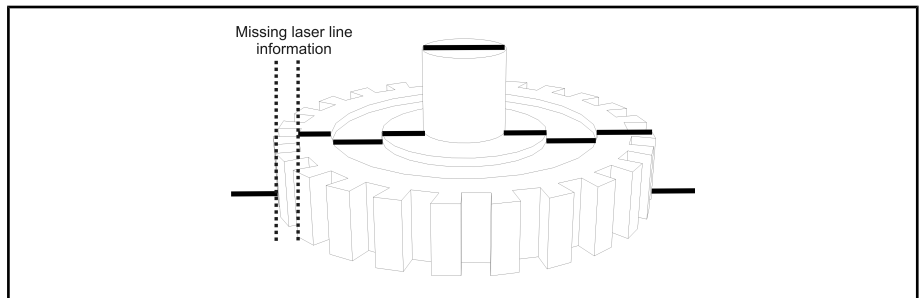
Partially correct intensity map



Fully corrected intensity map

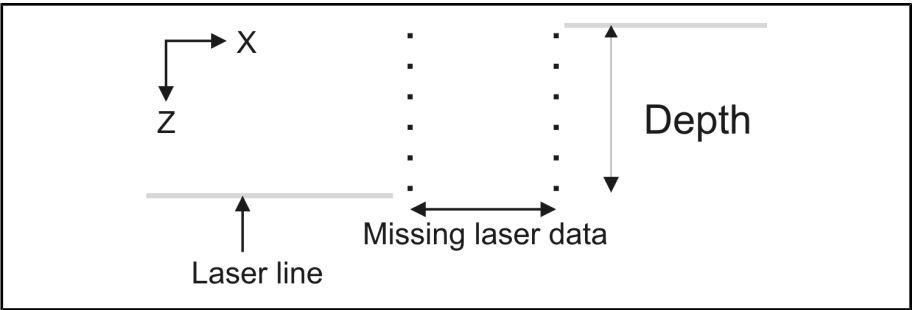
Filling missing data points (gaps)

There might be locations in the laser line images where the 3D Reconstruction module cannot find the laser line when it is scanning the rows or columns (depending on the orientation of the laser line) of the image.



These missing data points form gaps in the generated depth map. There are two types of gaps: a gradual elevation gap and a sharp elevation gap. Gaps are categorized into one of the two depending on their underlying surface. If the depth of the gap's underlying surface is considered to be gradual, the gap is considered to be a gradual elevation gap; if the depth of the gap's underlying surface is considered to be sharp, the gap is considered to be a sharp elevation gap.

The 3D Reconstruction module makes assumptions about the type of gap based on the difference between boundary values of the gap in the depth map. If there is a small difference, the module assumes that the gap is a gradual elevation gap and if there is a large difference, it assumes that the gap is a sharp elevation gap.



Filling mode

The 3D Reconstruction module can either ignore gaps or fill them using the X-then-Y filling mode.

The X-then-Y filling mode first analyzes each depth map row and fills any missing data points it encounters in that row, according to the specified filling operation. It then analyzes each column and fills any gaps it finds in that column, also according to the specified filling operation. Note that gaps whose boundaries touch the border of the image are not filled.

: filled gap 255: missing data point

20	20	255	255	255
255	255	255	255	255
50	255	255	190	189

Original

20	20	255	255	255
255	255	255	255	255
50	50	50	190	189

After X pass

20	20	255	255	255
35	35	255	255	255
50	50	50	190	189

After Y pass

For an 8- or 16-bit depth map image buffer, the gray values 255 and 65535 are used to indicate missing data points (gaps), respectively.

The gray values used to fill the gaps are approximations. If you require accurate results, see the *Phenomenae causing missing data* subsection in the *Filling missing data* section in *Chapter 16: 3D Reconstruction* for suggestions on improving your 3D reconstruction setup to minimize occurrences of gaps.

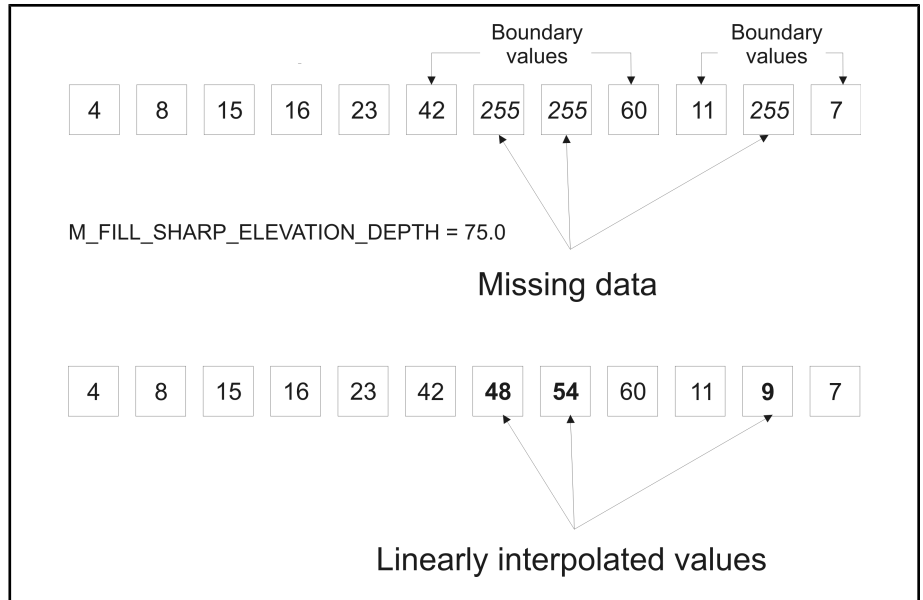
Filling operations

To fill gaps, two filling operations are available. The 3D Reconstruction module either propagates the value of one of the boundaries or does a linear interpolation between the two boundaries on each row or column of the gap. The module chooses one of these filling operations based on whether the gap is considered a gradual or a sharp elevation gap.

Linear interpolation

A gradual elevation gap is a range of missing data where the difference of the boundary values of that gap's row or column is less than the threshold specified using `M3dmapControl()` with `M_FILL_SHARP_ELEVATION_DEPTH`. The underlying surface of such a gap is considered to be part of the same surface as its boundaries, so this type of gap is always filled with linearly interpolated values.

For example, in the following image, **M_FILL_SHARP_ELEVATION_DEPTH** is set to 75.0. The values at the gap boundary are 42 and 60. Since the difference, 18, is less than the specified threshold of 75.0, the gap will be filled with linearly interpolated values.



You can specify to always use linear interpolation to fill missing data points, regardless of the depth difference at the boundaries, by setting **M_FILL_SHARP_ELEVATION_DEPTH** to **M_INFINITE**. This treats any elevation gap as being a gradual elevation gap.

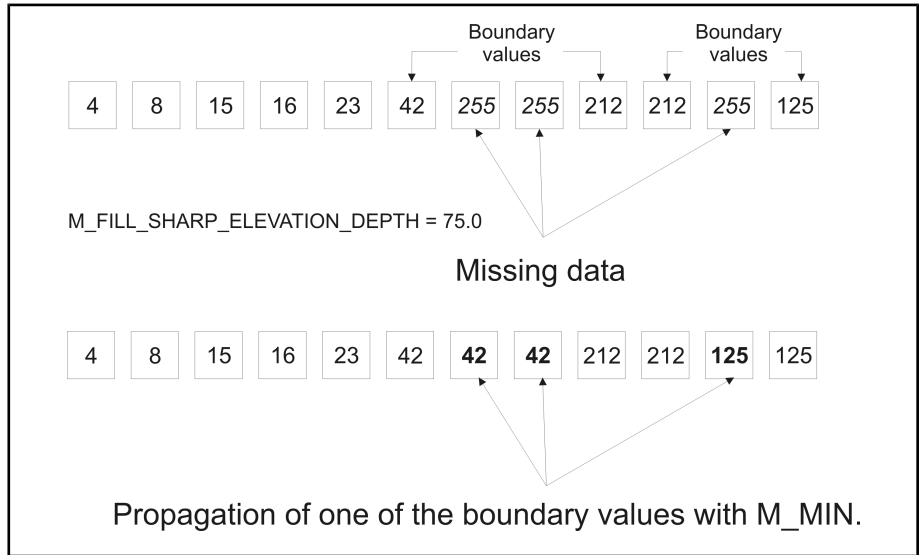
Propagation of one of the boundary values

A sharp elevation gap is a range of missing data where the underlying surface is considered to be discontinuous at least at one of its boundaries. The depth difference of the boundaries of the gap's row or column must be greater or equal to **M_FILL_SHARP_ELEVATION_DEPTH** for the gap to be considered a sharp elevation gap. For these types of gaps, you can select to either:

- Fill the sharp elevation gaps by propagating one of their boundaries. You should choose this fill operation if the gaps are not due to holes in the object.
- Leave the gaps as is by setting **M_FILL_SHARP_ELEVATION** to **M_DISABLE** because the gaps are due to holes in the object.

To propagate one of the boundary values, you can choose to fill the gaps with either the minimum or maximum pixel value of the gaps' boundaries. To do so, set **M_FILL_SHARP_ELEVATION** to **M_MIN** or **M_MAX**, respectively. If the X-then-Y filling mode is used, then rows will be filled first by propagating one of the boundaries on either side of the row, followed by the filling of the columns.

Note if one of the boundaries is invalid (it has a value of 255), then the gap will not be filled.



To use propagation of the boundary values to fill any type of gap, set **M_FILL_SHARP_ELEVATION_DEPTH** to 0.0. All gaps will be treated as being sharp elevation gaps.

Phenomenae causing missing data

Laser line information might be missing from some locations in the image due to different phenomenae. Depending on your reconstruction setup, you might be able to reduce the amount of missing data.

- **Camera occlusion and laser occlusion.** Camera occlusion occurs when the laser line is hidden from the camera's field of view by the object's geometry. Laser occlusion occurs when the object's geometry prevents the laser line from illuminating parts of the object's surface.

To fix camera occlusion problems, try to position your camera in such a way that it is able to capture the laser line at all times. If this is not possible, add a camera to grab the laser lines which are hidden from the first camera.

To fix laser occlusion problems, add a second laser line which perfectly overlaps the first one to cover the regions of missing data.

- **Reflectance.** The object's surface can absorb enough light to prevent laser line detection. For example, dark markings might not reflect enough light back to the camera.

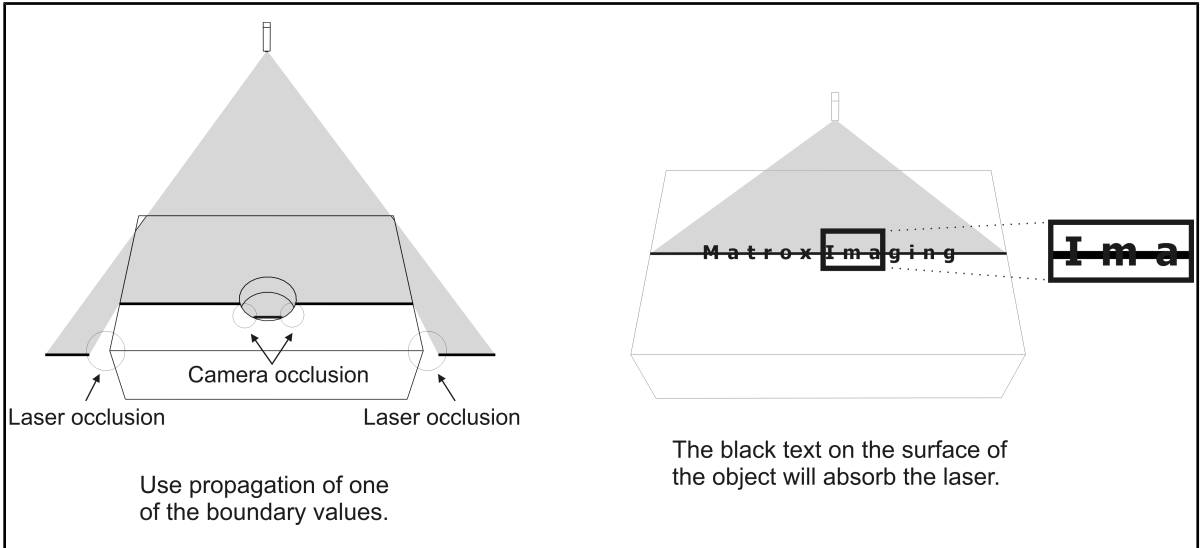
Increase the laser line's intensity to compensate for the light being absorbed. If this is not possible, adjust the iris opening (aperture) of your camera to gain more exposure.

- **Scaling.** The aspect ratio of the object in the depth map might be maintained by filling rows with missing data (255). This occurs when the camera is not able to grab an image with the laser line for each slice of the object.

Decrease the speed at which the object is moving under the laser plane or increase the frame rate or resolution of the camera to provide more 3D data to the module. You can use `M3dmapControl()` with `M_PIXEL_SIZE_X`, `M_PIXEL_SIZE_Y`, and `M_GRAY_LEVEL_SIZE_Z` to adjust the scale of the pixels and obtain a smaller sized depth map. You might need to use the offset parameters `M_WORLD_OFFSET_X`, `M_WORLD_OFFSET_Y`, and `M_WORLD_OFFSET_Z` to recenter the object in the depth map, if required.

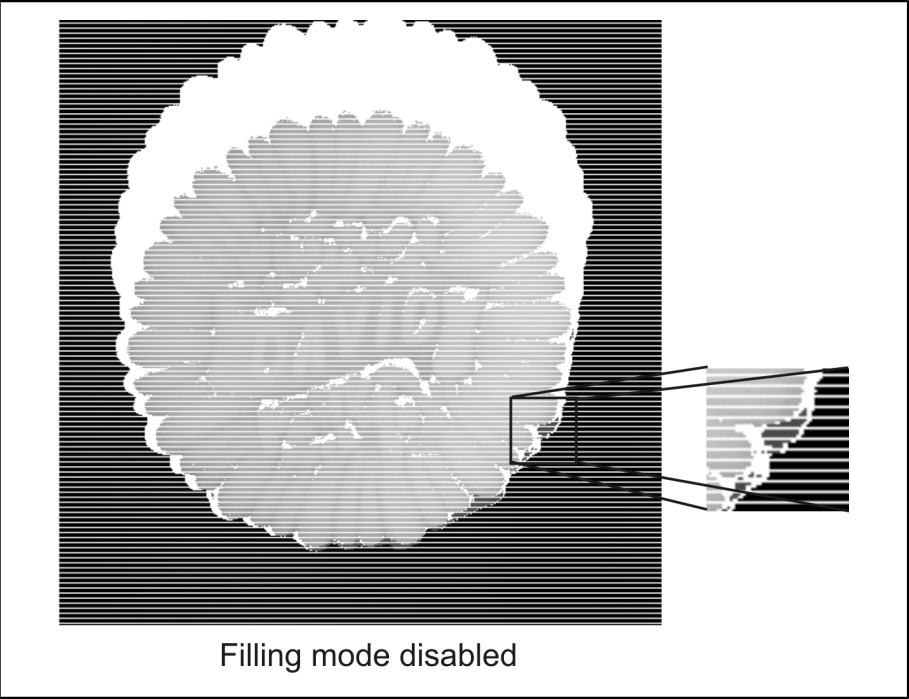
- ❖ This type of gap can only appear if the 3D reconstruction context was allocated in `M_CALIBRATED_CAMERA_LINEAR_MOTION` 3D reconstruction mode.

The following image shows an example of camera occlusion, laser occlusion, and reflectance. In the case of camera occlusion and laser occlusion, you could use propagation of one of the boundary values because the gaps might be caused by sharp elevation differences. For the gaps due to reflectance, for example, you could use linear interpolation because the underlying surface of the gap is flat.



Examples

The following image shows the generated depth map of a cookie with missing laser data due to scaling as well as camera and laser occlusion.



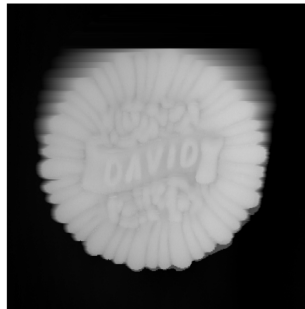
To fill these gaps, you can choose linear interpolation, propagation of one of the boundary values, or a mix of both. You can use **M3dmapControl()** with **M_FILL_SHARP_ELEVATION** and **M_FILL_SHARP_ELEVATION_DEPTH** to specify the filling operation.



Propagation of the maximum boundary value.



Propagation of the minimum boundary value.



Linear interpolation everywhere.

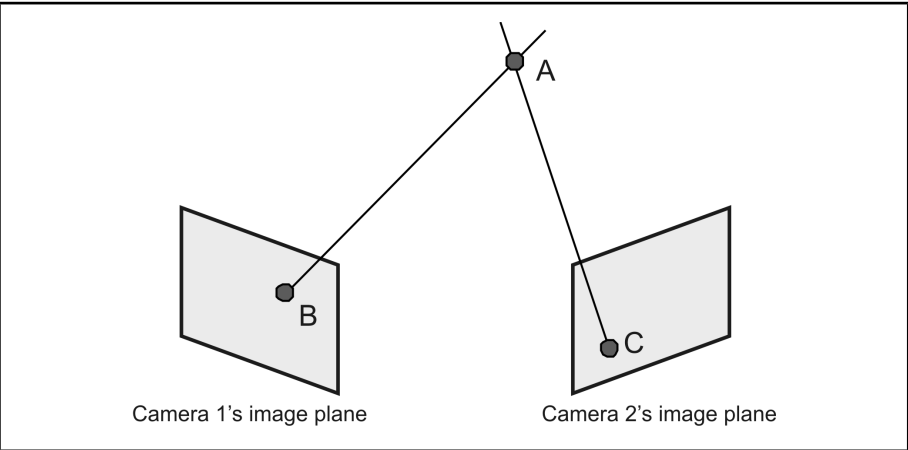


Linear interpolation for gradual elevations, and propagation of the minimum boundary value for sharp elevations.

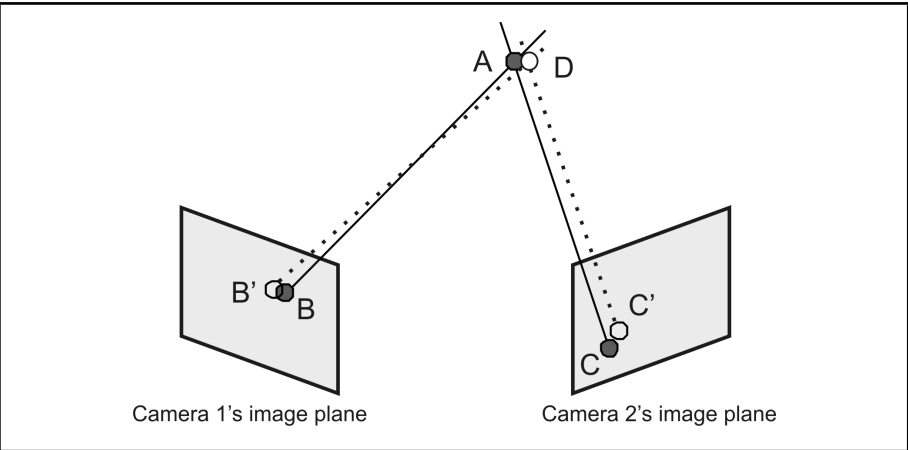
Triangulation

The 3D Reconstruction module supports 3D reconstruction from triangulation. Triangulation is the process of finding an object's location in 3D space, using two or more planar views of the object. In triangulation, an object's feature must be seen by at least two cameras. If the positions of the cameras in 3D space are known, the feature's location on each camera's image plane and the camera's location in the world are used to calculate the feature's location in the world.

The following image shows the projection of point A on the image plane of two cameras. The intersection of point B and C's projection line results in the location of point A in 3D space.



However, in reality, the projection of a point on a camera's image plane is not perfect. Rounding and extraction errors prevent point A from being projected with accuracy on the cameras' image planes.

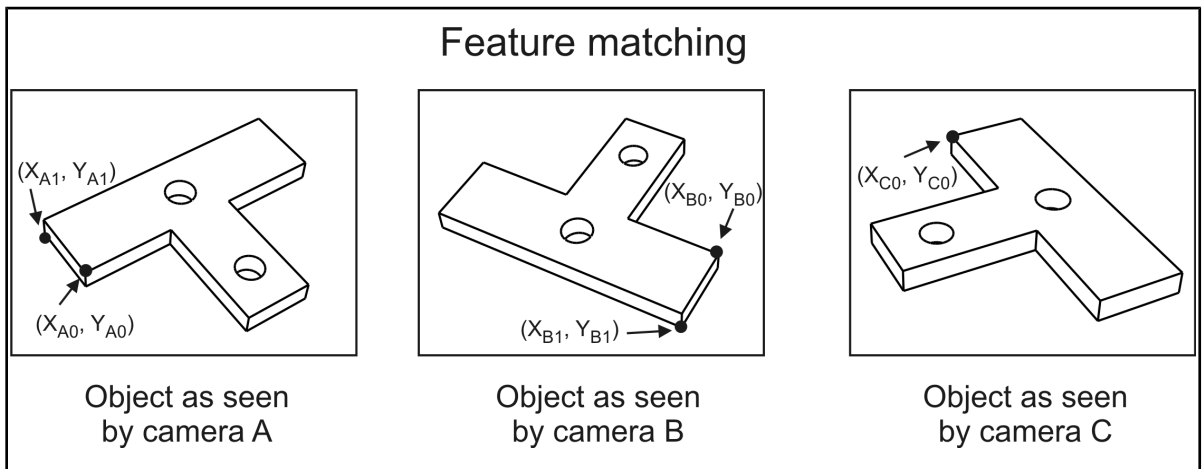


Points B' and C' will be the points used in the calculation and point D is the resulting approximation of A.

Additional cameras increase the accuracy of the result. However, not all cameras need to see the point; only two cameras are necessary for triangulation, but the more cameras involved, the more accurate the result.

Performing triangulation

To do triangulation, the 3D Reconstruction module requires that you match points (features) common in both images. That is, you must identify common points which are seen by at least two cameras, identify the points' coordinates in their respective images and supply the coordinates to **M3dmapTriangulate()**. For example, in the following image, the points (X_{A0}, Y_{A0}) , (X_{B0}, Y_{B0}) , and (X_{C0}, Y_{C0}) are paired together. (X_{A1}, Y_{A1}) and (X_{B1}, Y_{B1}) are paired together, but camera C does not see this point.



You can pair multiple points with a single call to **M3dmapTriangulate()** by specifying their coordinates. To pair the points given in the previous example, you would give the following arguments to **M3dmapTriangulate()**:

Point number	Array index	XPixelArrayPtr	YPixelArrayPtr	CalibrationIdArrayPtr
0	0	X _{A0}	Y _{A0}	CalibrationObjectA
	1	X _{B0}	Y _{B0}	CalibrationObjectB
	2	X _{C0}	Y _{C0}	CalibrationObjectC
1	3	X _{A1}	Y _{A1}	
	4	X _{B1}	Y _{B1}	
	5	M_INVALID_POINT	M_INVALID_POINT	

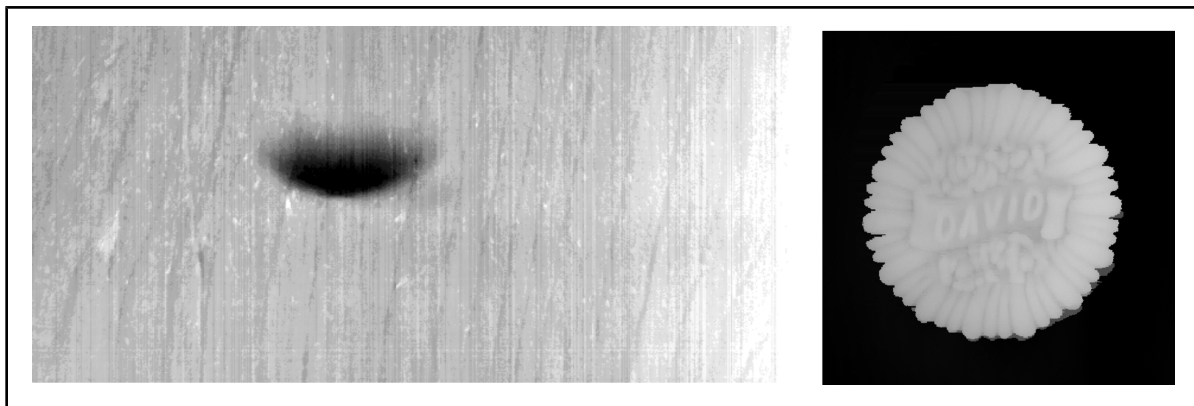
Since camera C cannot see one of the points, you need to specify **M_INVALID_POINT** for the appropriate indices.

Note that you also need to specify the identifier of the calibration objects for the different camera setups. The order you specify the identifiers must match the order in which you specify the coordinates.

- ❖ Note that triangulation requires that all the cameras be calibrated in a 3D mode using **McalAlloc()** with **M_TSAI_BASED** or **M_3D_ROBOTICS**. All cameras must also share a common 3D coordinate system. For more information on calibrating your camera setup, see *Chapter 5: Camera calibration*.

MIL 3D Reconstruction example

The 3D Reconstruction example *m3dmap.cpp* creates a partially corrected depth map of a piece of wood's surface and a fully corrected depth map of a cookie.



The partially corrected depth map of the wood's surface is then used by the Blob Analysis module to find any surface defects. The fully corrected depth map of the cookie is used to compute its volume.

```

/*****
/*
* File name: m3dmap.cpp
*
* Synopsis: This program inspects a wood surface using laser profiling to find
*           any depth defects.
*
*/

#include <mil.h>
#include <malloc.h>
#include <math.h>

#if M_MIL_USE_CE
#error Example not supported on this platform, AVI files are not supported under Windows CE.
#endif

#if M_MIL_USE_WINDOWS
/* Include DirectX display only on Windows. */
#include "..\mimlocatepeak1d\MdispD3D\MdispD3D.h"
#endif

```

```

/* Example functions declarations. */
void DepthCorrectionExample(MIL_ID MilSystem, MIL_ID MilDisplay);
void CalibratedCameraExample(MIL_ID MilSystem, MIL_ID MilDisplay);

/* Utility functions declarations. */
void PerformBlobAnalysis(MIL_ID MilSystem,
                        MIL_ID MilDisplay,
                        MIL_ID MilOverlayImage,
                        MIL_ID DepthMapId);
void SetupColorDisplay(MIL_ID MilSystem, MIL_ID MilDisplay, MIL_INT SizeBit);
MIL_DOUBLE ComputeVolume(MIL_ID MilSystem, MIL_ID DepthMapId);

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

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

    /* Run the depth correction example. */
    DepthCorrectionExample(MilSystem, MilDisplay);

    /* Run the calibrated camera example. */
    CalibratedCameraExample(MilSystem, MilDisplay);

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

    return 0;
}

/*****
Depth correction example.
*****/

/* Input sequence specifications. */
#define CALIBRATION_SEQUENCE_FILE    M_IMAGE_PATH MIL_TEXT("CalibrationPlanes.avi")
#define OBJECT_SEQUENCE_FILE        M_IMAGE_PATH MIL_TEXT("ScannedObject.avi")

/* Peak detection parameters. */
#define MAX_LINE_WIDTH                8
#define MIN_INTENSITY                 120

/* Calibration heights in mm. */
static const double CORRECTED_DEPTHS[] = {1.25, 2.50, 3.75, 5.00};

#define SCALE_FACTOR    10000.0 /* (depth in world units) * SCALE_FACTOR gives gray levels */

```

```

/* Annotation position. */
#define CALIB_TEXT_POS_X      450
#define CALIB_TEXT_POS_Y      15

void DepthCorrectionExample(MIL_ID MilSystem, MIL_ID MilDisplay)
{
    MIL_ID      MilOverlayImage, /* Overlay image buffer identifier. */
    MilImage,    /* Image buffer identifier (for processing). */
    DepthMapId,  /* Image buffer identifier (for results). */
    LaserId,     /* 3dmap laser profiling context identifier. */
    ScanId;      /* 3dmap result buffer identifier. */
    MIL_INT     SizeX,          /* Width of grabbed images. */
    SizeY,       /* Height of grabbed images. */
    NbCalibrationImages, /* Number of planes of known heights. */
    NbObjectImages,    /* Number of frames for scanned objects. */
    n;               /* Counter. */
    MIL_DOUBLE   FrameRate,     /* Number of grabbed frames per second (in AVI). */
    StartTime,    /* Time at the beginning of each iteration. */
    EndTime,      /* Time after processing for each iteration. */
    WaitTime;     /* Time to wait for next frame. */

    /* Inquire characteristics of the input sequences. */
    MbufDiskInquire(CALIBRATION_SEQUENCE_FILE, M_SIZE_X,      &SizeX);
    MbufDiskInquire(CALIBRATION_SEQUENCE_FILE, M_SIZE_Y,      &SizeY);
    MbufDiskInquire(CALIBRATION_SEQUENCE_FILE, M_NUMBER_OF_IMAGES, &NbCalibrationImages);
    MbufDiskInquire(CALIBRATION_SEQUENCE_FILE, M_FRAME_RATE,   &FrameRate);
    MbufDiskInquire(OBJECT_SEQUENCE_FILE,     M_NUMBER_OF_IMAGES, &NbObjectImages);

    /* Allocate buffer to hold images. */
    MbufAlloc2d(MilSystem, SizeX, SizeY, 8+M_UNSIGNED, M_IMAGE+M_DISP+M_PROC, &MilImage);
    MbufClear(MilImage, 0.0);

    MosPrintf(MIL_TEXT("\nDEPTH ANALYSIS:\n"));
    MosPrintf(MIL_TEXT("-----\n\n"));
    MosPrintf(MIL_TEXT("This program performs a surface inspection to detect depth\n"));
    MosPrintf(MIL_TEXT("defects on a wood surface using a laser profiling system.\n\n"));
    MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
    MosGetch();

    /* Select display. */
    MdispSelect(MilDisplay, MilImage);

    /* Prepare for overlay annotations. */
    MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
    MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);
    MgraControl(M_DEFAULT, M_BACKGROUND_MODE, M_TRANSPARENT);
    MgraColor(M_DEFAULT, M_COLOR_WHITE);

    /* Allocate 3dmap objects. */
    M3dmapAlloc(MilSystem, M_LASER, M_DEPTH_CORRECTION, &LaserId);
    M3dmapAllocResult(MilSystem, M_LASER_DATA, M_DEFAULT, &ScanId);

    /* Set laser line extraction options. */

```

```

M3dmapControl(LaserId, M_DEFAULT, M_PEAK_WIDTH, MAX_LINE_WIDTH);
M3dmapControl(LaserId, M_DEFAULT, M_MIN_INTENSITY, MIN_INTENSITY);

/* Open the calibration sequence file for reading. */
MbufImportSequence(CALIBRATION_SEQUENCE_FILE, M_DEFAULT, M_NULL, M_NULL, NULL, M_NULL,
M_NULL, M_OPEN);

/* Read and process all images in the input sequence. */
MappTimer(M_TIMER_READ+M_SYNCHRONOUS, &StartTime);

for (n = 0; n < NbCalibrationImages; n++)
{
    MIL_TEXT_CHAR CalibString[32];

    /* Read image from sequence. */
    MbufImportSequence(CALIBRATION_SEQUENCE_FILE, M_DEFAULT, M_LOAD, M_NULL, &MilImage,
M_DEFAULT, 1, M_READ);

    /* Annotate the image with the calibration height. */
    MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);
    M0s_sprintf_s2(CalibString, 32, MIL_TEXT("Plane %d: %.2f mm"),
n+1, CORRECTED_DEPTH[n]);
    MgraText(M_DEFAULT, MilOverlayImage, CALIB_TEXT_POS_X, CALIB_TEXT_POS_Y, CalibString);

    /* Set desired corrected depth of next calibration plane. */
    M3dmapControl(LaserId, M_DEFAULT, M_CORRECTED_DEPTH, CORRECTED_DEPTH[n]*SCALE_FACTOR);

    /* Analyze the image to extract laser line. */
    M3dmapAddScan(LaserId, ScanId, MilImage, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);

    /* Wait to have a proper frame rate, if necessary. */
    MappTimer(M_TIMER_READ+M_SYNCHRONOUS, &EndTime);
    WaitTime = (1.0/FrameRate) - (EndTime - StartTime);
    if (WaitTime > 0)
        MappTimer(M_TIMER_WAIT, &WaitTime);
    MappTimer(M_TIMER_READ+M_SYNCHRONOUS, &StartTime);
}

/* Close the calibration sequence file. */
MbufImportSequence(CALIBRATION_SEQUENCE_FILE, M_DEFAULT, M_NULL, M_NULL, NULL, M_NULL,
M_NULL, M_CLOSE);

/* Calibrate the laser profiling context using plane of known heights. */
M3dmapCalibrate(LaserId, ScanId, M_NULL, M_DEFAULT);

MosPrintf(MIL_TEXT("The laser profiling system has been calibrated using 4 planes\n"));
MosPrintf(MIL_TEXT("of known heights.\n\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

MosPrintf(MIL_TEXT("The wood surface is being scanned.\n\n"));

```



```

/* Empty all result buffer contents.
   It will now be reused for extracting corrected depths. */
M3dmapAddScan(M_NULL, ScanId, M_NULL, M_NULL, M_NULL, M_DEFAULT, M_RESET);

/* Open the object sequence file for reading. */
MbufDiskInquire(OBJECT_SEQUENCE_FILE, M_FRAME_RATE, &FrameRate);
MbufImportSequence(OBJECT_SEQUENCE_FILE, M_DEFAULT, M_NULL, M_NULL, NULL, M_NULL,
                  M_NULL, M_OPEN);

/* Read and process all images in the input sequence. */
MappTimer(M_TIMER_READ+M_SYNCHRONOUS, &StartTime);
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

for (n = 0; n < NbObjectImages; n++)
{
    /* Read image from sequence. */
    MbufImportSequence(OBJECT_SEQUENCE_FILE, M_DEFAULT, M_LOAD, M_NULL, &MilImage,
                    M_DEFAULT, 1, M_READ);

    /* Analyze the image to extract laser line and correct its depth. */
    M3dmapAddScan(LaserId, ScanId, MilImage, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);

    /* Wait to have a proper frame rate, if necessary. */
    MappTimer(M_TIMER_READ+M_SYNCHRONOUS, &EndTime);
    WaitTime = (1.0/FrameRate) - (EndTime - StartTime);
    if (WaitTime > 0)
        MappTimer(M_TIMER_WAIT, &WaitTime);
    MappTimer(M_TIMER_READ+M_SYNCHRONOUS, &StartTime);
}

/* Close the object sequence file. */
MbufImportSequence(OBJECT_SEQUENCE_FILE, M_DEFAULT, M_NULL, M_NULL, NULL, M_NULL,
                  M_NULL, M_CLOSE);

/* Allocate images for depth map. */
MbufAlloc2d(MilSystem, SizeX, NbObjectImages, 16+M_UNSIGNED,
            M_IMAGE+M_PROC+M_DISP, &DepthMapId);

/* Get depth map from accumulated information in the result buffer. */
M3dmapExtract(ScanId, DepthMapId, M_NULL, M_CORRECTED_DEPTH_MAP, M_DEFAULT, M_DEFAULT);

/* Show depth map and find defects. */
SetupColorDisplay(MilSystem, MilDisplay, MbufInquire(DepthMapId, M_SIZE_BIT, M_NULL));

/* Display corrected depth map. */
MdispSelect(MilDisplay, DepthMapId);
MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);

MosPrintf(MIL_TEXT("The pseudo-color depth map of the surface is displayed.\n\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

```

```

PerformBlobAnalysis(MilSystem, MilDisplay, MilOverlayImage, DepthMapId);

MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Disassociates display LUT and clear overlay. */
MdispLut(MilDisplay, M_DEFAULT);
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

/* Free all allocations. */
M3dmapFree(ScanId);
M3dmapFree(LaserId);
MbufFree(DepthMapId);
MbufFree(MilImage);
}

/* Values used for binarization. */
#define EXPECTED_HEIGHT      3.4 /* Inspected surface should be at this height (in mm) */
#define DEFECT_THRESHOLD     0.2 /* Max acceptable deviation from expected height (mm) */
#define SATURATED_DEFECT    1.0 /* Deviation at which defect will appear red (in mm) */

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

/* Pixel offset for drawing text. */
#define TEXT_H_OFFSET_1      -50
#define TEXT_V_OFFSET_1      -6
#define TEXT_H_OFFSET_2      -30
#define TEXT_V_OFFSET_2      6

/* Find defects in corrected depth map, compute max deviation and draw contours. */
void PerformBlobAnalysis(MIL_ID MilSystem,
                        MIL_ID MilDisplay,
                        MIL_ID MilOverlayImage,
                        MIL_ID DepthMapId)
{
    MIL_ID      MilBinImage, /* Binary image buffer identifier. */
    MilBlobFeatureList, /* Feature list identifier. */
    MilBlobResult; /* Blob result buffer identifier. */
    MIL_INT     SizeX, /* Width of depth map. */
    SizeY, /* Height of depth map. */
    TotalBlobs, /* Total number of blobs. */
    n, /* Counter. */
    *MinPixels; /* Maximum height of defects. */
    MIL_DOUBLE  DefectThreshold, /* A gray level below it is a defect. */
    *CogX, /* X coordinate of center of gravity. */
    *CogY; /* Y coordinate of center of gravity. */

    /* Get size of depth map. */
    MbufInquire(DepthMapId, M_SIZE_X, &SizeX);
    MbufInquire(DepthMapId, M_SIZE_Y, &SizeY);

    /* Allocate a binary image buffer for fast processing. */

```

```

MbufAlloc2d(MilSystem, SizeX, SizeY, 1+M_UNSIGNED, M_IMAGE+M_PROC, &MilBinImage);

/* Binarize image. */
DefectThreshold = (EXPECTED_HEIGHT-DEFECT_THRESHOLD) * SCALE_FACTOR;
MimBinarize(DepthMapId, MilBinImage, M_LESS_OR_EQUAL, DefectThreshold, M_NULL);

/* Remove small particles. */
MimOpen(MilBinImage, MilBinImage, MIN_BLOB_RADIUS, M_BINARY);

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

/* Enable the Center Of Gravity and Min Pixel features calculation. */
MblobSelectFeature(MilBlobFeatureList, M_CENTER_OF_GRAVITY);
MblobSelectFeature(MilBlobFeatureList, M_MIN_PIXEL);

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

/* Calculate selected features for each blob. */
MblobCalculate(MilBinImage, DepthMapId, MilBlobFeatureList, MilBlobResult);

/* Get the total number of selected blobs. */
MblobGetNumber(MilBlobResult, &TotalBlobs);
MosPrintf(MIL_TEXT("Number of defects: %ld\n"), TotalBlobs);

/* Read and print the blob characteristics. */
if ((CogX      = (MIL_DOUBLE *)malloc(TotalBlobs*sizeof(MIL_DOUBLE))) &&
    (CogY      = (MIL_DOUBLE *)malloc(TotalBlobs*sizeof(MIL_DOUBLE))) &&
    (MinPixels = (MIL_INT   *)malloc(TotalBlobs*sizeof(MIL_INT   )))
    )
{
    /* Get the results. */
    MblobGetResult(MilBlobResult, M_CENTER_OF_GRAVITY_X, CogX);
    MblobGetResult(MilBlobResult, M_CENTER_OF_GRAVITY_Y, CogY);
    MblobGetResult(MilBlobResult, M_MIN_PIXEL+M_TYPE_MIL_INT, MinPixels);

    /* Draw the defects. */
    MgraColor(M_DEFAULT, M_COLOR_RED);
    MblobDraw(M_DEFAULT, MilBlobResult, MilOverlayImage,
              M_DRAW_BLOBS, M_INCLUDED_BLOBS, M_DEFAULT);
    MgraColor(M_DEFAULT, M_COLOR_WHITE);

    /* Print the depth of each blob. */
    for(n=0; n < TotalBlobs; n++)
    {
        MIL_DOUBLE   DepthOfDefect;
        MIL_TEXT_CHAR DepthString[16];

        /* Write the depth of the defect in the overlay. */
        DepthOfDefect = EXPECTED_HEIGHT - (MinPixels[n]/SCALE_FACTOR);
        MOS_sprintf_s1(DepthString, 16, MIL_TEXT("%.2f mm"), DepthOfDefect);
    }
}

```

```

        MosPrintf(MIL_TEXT("Defect #d: depth =%5.2f mm\n\n"),
            n, DepthOfDefect);
        MgraText(M_DEFAULT, MilOverlayImage, CogX[n]+TEXT_H_OFFSET_1,
            CogY[n]+TEXT_V_OFFSET_1, MIL_TEXT("Defect depth"));
        MgraText(M_DEFAULT, MilOverlayImage, CogX[n]+TEXT_H_OFFSET_2,
            CogY[n]+TEXT_V_OFFSET_2, DepthString);
    }

    free(CogX);
    free(CogY);
    free(MinPixels);
}
else
    MosPrintf(MIL_TEXT("Error: Not enough memory.\n\n"));

/* Free all allocations. */
MblobFree(MilBlobResult);
MblobFree(MilBlobFeatureList);
MbufFree(MilBinImage);
}

/* Color constants for display LUT. */
#define BLUE_HUE  171.0      /* Expected depths will be blue.  */
#define RED_HUE   0.0        /* Worst defects will be red.    */
#define FULL_SATURATION 255  /* All colors are fully saturated. */
#define HALF_LUMINANCE 128   /* All colors have half luminance. */

/* Creates a color display LUT to show defects in red. */
void SetupColorDisplay(MIL_ID MilSystem, MIL_ID MilDisplay, MIL_INT SizeBit)
{
    MIL_ID  RampLut1BandId,      /* LUT containing hue values.      */
            RampLut3BandId,      /* RGB LUT used by display.        */
            ColorImageId;        /* Image used for HSL to RGB conversion. */
    MIL_INT DefectGrayLevel,     /* Gray level under which all is red. */
            ExpectedGrayLevel,   /* Gray level over which all is blue.  */
            NbGrayLevels;

    /* Number of possible gray levels in corrected depth map. */
    NbGrayLevels = (MIL_INT)(1 << SizeBit);

    /* Allocate 1-band LUT that will contain hue values. */
    MbufAllocId(MilSystem, NbGrayLevels, 8+M_UNSIGNED, M_LUT, &RampLut1BandId);

    /* Compute limit gray values. */
    DefectGrayLevel  = (MIL_INT)((EXPECTED_HEIGHT-SATURATED_DEFECT)*SCALE_FACTOR);
    ExpectedGrayLevel = (MIL_INT)(EXPECTED_HEIGHT*SCALE_FACTOR);

    /* Create hue values for each possible gray level. */
    MgenLutRamp(RampLut1BandId, 0, RED_HUE, DefectGrayLevel, RED_HUE);
    MgenLutRamp(RampLut1BandId, DefectGrayLevel, RED_HUE,
        ExpectedGrayLevel, BLUE_HUE);
    MgenLutRamp(RampLut1BandId, ExpectedGrayLevel, BLUE_HUE, NbGrayLevels-1, BLUE_HUE);
}

```

```

/* Create a HSL image buffer. */
MbufAllocColor(MilSystem, 3, NbGrayLevels, 1, 8+M_UNSIGNED, M_IMAGE, &ColorImageId);
MbufClear(ColorImageId, M_RGB888(0, FULL_SATURATION, HALF_LUMINANCE));

/* Set its H band (hue) to the LUT contents and convert the image to RGB. */
MbufCopyColor2d(RampLut1BandId, ColorImageId, 0, 0, 0, 0, 0, 0, NbGrayLevels, 1);
MimConvert(ColorImageId, ColorImageId, M_HSL_TO_RGB);

/* Create RGB LUT to give to display and copy image contents. */
MbufAllocColor(MilSystem, 3, NbGrayLevels, 1, 8+M_UNSIGNED, M_LUT, &RampLut3BandId);
MbufCopy(ColorImageId, RampLut3BandId);

/* Associates LUT to display. */
MdispLut(MilDisplay, RampLut3BandId);

/* Free all allocations. */
MbufFree(RampLut1BandId);
MbufFree(RampLut3BandId);
MbufFree(ColorImageId);
}

/*****
Calibrated camera example.
*****/

/* Input sequence specifications. */
#define GRID_FILENAME           M_IMAGE_PATH MIL_TEXT("GridForLaser.mim")
#define LASERLINE_FILENAME     M_IMAGE_PATH MIL_TEXT("LaserLine.mim")
#define OBJECT2_SEQUENCE_FILE  M_IMAGE_PATH MIL_TEXT("Cookie.avi")

/* Calibration grid parameters. */
#define GRID_NB_ROWS           13
#define GRID_NB_COLS           12
#define GRID_ROW_SPACING       5.0      /* in mm */
#define GRID_COL_SPACING       5.0      /* in mm */

/* Laser device setup parameters. */
#define CONVEYOR_SPEED          -0.2     /* in mm/frame */

/* Depth map generation parameters. */
#define DEPTH_MAP_SIZE_X        480      /* in pixels */
#define DEPTH_MAP_SIZE_Y        480      /* in pixels */
#define GAP_DEPTH               1.5      /* in mm */
#define SCALE_X                 0.15     /* in mm/pixel */
#define SCALE_Y                 0.15     /* in mm/pixel */
#define SCALE_Z                 -0.0003  /* in mm/gray level */
#define WORLD_OFFSET_X          -12.0    /* in mm */
#define WORLD_OFFSET_Y          4.0      /* in mm */
#define WORLD_OFFSET_Z          0.0      /* in mm */

/* D3D display parameters */
#define D3D_DISPLAY_SIZE_X      640
#define D3D_DISPLAY_SIZE_Y      480

```

```

/* Peak detection parameters. */
#define MAX_LINE_WIDTH_2      4
#define MIN_INTENSITY_2      100

void CalibratedCameraExample(MIL_ID MilSystem, MIL_ID MilDisplay)
{
    MIL_ID      MilOverlayImage, /* Overlay image buffer identifier. */
    MilImage,   /* Image buffer identifier (for processing). */
    MilCalibration, /* Calibration object. */
    DepthMapId, /* Image buffer identifier (for results). */
    LaserId,    /* 3dmap laser profiling context identifier. */
    ScanId;     /* 3dmap result buffer identifier. */
    MIL_INT     CalibrationStatus, /* Used to ensure if McalGrid() worked. */
    SizeX,      /* Width of grabbed images. */
    SizeY,      /* Height of grabbed images. */
    NumberOfImages, /* Number of frames for scanned objects. */
    n;          /* Counter. */
    MIL_DOUBLE  FrameRate, /* Number of grabbed frames per second (in AVI). */
    StartTime, /* Time at the beginning of each iteration. */
    EndTime,   /* Time after processing for each iteration. */
    WaitTime,  /* Time to wait for next frame. */
    Volume;    /* Volume of scanned object. */

    MosPrintf(MIL_TEXT("\n3D PROFILING AND VOLUME ANALYSIS:\n"));
    MosPrintf(MIL_TEXT("-----\n\n"));
    MosPrintf(MIL_TEXT("This program generates corrected 3d data of a scanned cookie\n"));
    MosPrintf(MIL_TEXT("and computes its volume. The laser profiling system uses a\n"));
    MosPrintf(MIL_TEXT("3d-calibrated camera.\n\n"));

    /* Load grid image for camera calibration. */
    MbufRestore(GRID_FILENAME, MilSystem, &MilImage);

    /* Select display. */
    MdispSelect(MilDisplay, MilImage);

    MosPrintf(MIL_TEXT("Calibrating the camera...\n\n"));

    MbufInquire(MilImage, M_SIZE_X, &SizeX);
    MbufInquire(MilImage, M_SIZE_Y, &SizeY);

    /* Allocate calibration object in 3D mode. */
    McalAlloc(MilSystem, M_TSAI_BASED, M_DEFAULT, &MilCalibration);

    /* Calibrate the camera. */
    McalGrid(MilCalibration, MilImage, 0.0, 0.0, 0.0, GRID_NB_ROWS, GRID_NB_COLS,
        GRID_ROW_SPACING, GRID_COL_SPACING, M_DEFAULT, M_CHESSBOARD_GRID);

    McalInquire(MilCalibration, M_CALIBRATION_STATUS+M_TYPE_MIL_INT, &CalibrationStatus);
    if (CalibrationStatus != M_CALIBRATED)
    {
        McalFree(MilCalibration);
        MbufFree(MilImage);
    }
}

```

```

    MosPrintf(MIL_TEXT("Camera calibration failed.\n"));
    MosPrintf(MIL_TEXT("Press <Enter> to end.\n\n"));
    MosGetch();
    return;
}

/* Prepare for overlay annotations. */
MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
MdispInquire(MilDisplay, M_OVERLAY_ID, &MilOverlayImage);
MgraColor(M_DEFAULT, M_COLOR_GREEN);

/* Draw calibration points. */
McalDraw(M_DEFAULT, MilCalibration, MilOverlayImage, M_DRAW_IMAGE_POINTS,
                                                M_DEFAULT, M_DEFAULT);

MosPrintf(MIL_TEXT("The camera was calibrated using a chessboard grid.\n\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Disable overlay. */
MdispControl(MilDisplay, M_OVERLAY, M_DISABLE);

/* Load laser line image. */
MbufLoad(LASERLINE_FILENAME, MilImage);

/* Allocate 3dmap objects. */
M3dmapAlloc(MilSystem, M_LASER, M_CALIBRATED_CAMERA_LINEAR_MOTION, &LaserId);
M3dmapAllocResult(MilSystem, M_LASER_DATA, M_DEFAULT, &ScanId);

/* Set laser line extraction options. */
M3dmapControl(LaserId, M_DEFAULT, M_PEAK_WIDTH    , MAX_LINE_WIDTH_2);
M3dmapControl(LaserId, M_DEFAULT, M_MIN_INTENSITY, MIN_INTENSITY_2 );

/* Calibrate laser profiling context. */
M3dmapAddScan(LaserId, ScanId, MilImage, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);
M3dmapCalibrate(LaserId, ScanId, MilCalibration, M_DEFAULT);

MosPrintf(MIL_TEXT("The laser profiling system has been calibrated using the image\n"));
MosPrintf(MIL_TEXT("of one laser line.\n\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Empty all result buffer contents.
   It will now be reused for extracting 3d points. */
M3dmapAddScan(M_NULL, ScanId, M_NULL, M_NULL, M_NULL, M_DEFAULT, M_RESET);

/* Set speed of scanned object (speed in mm/frame is constant). */
M3dmapControl(LaserId, M_DEFAULT, M_SCAN_SPEED, CONVEYOR_SPEED);

/* Inquire characteristics of the input sequence. */
MbufDiskInquire(OBJECT2_SEQUENCE_FILE, M_NUMBER_OF_IMAGES, &NumberOfImages);
MbufDiskInquire(OBJECT2_SEQUENCE_FILE, M_FRAME_RATE, &FrameRate);

```

```

/* Open the object sequence file for reading. */
MbufImportSequence(OBJECT2_SEQUENCE_FILE, M_DEFAULT, M_NULL, M_NULL, NULL, M_NULL,
                  M_NULL, M_OPEN);

MosPrintf(MIL_TEXT("The cookie is being scanned to generate 3d data.\n\n"));

/* Read and process all images in the input sequence. */
MappTimer(M_TIMER_READ+M_SYNCHRONOUS, &StartTime);

for (n = 0; n < NumberOfImages; n++)
{
    /* Read image from sequence. */
    MbufImportSequence(OBJECT2_SEQUENCE_FILE, M_DEFAULT, M_LOAD, M_NULL, &MilImage,
                    M_DEFAULT, 1, M_READ);

    /* Analyze the image to extract laser line and correct its depth. */
    M3dmapAddScan(LaserId, ScanId, MilImage, M_NULL, M_NULL, M_DEFAULT, M_DEFAULT);

    /* Wait to have a proper frame rate, if necessary. */
    MappTimer(M_TIMER_READ+M_SYNCHRONOUS, &EndTime);
    WaitTime = (1.0/FrameRate) - (EndTime - StartTime);
    if (WaitTime > 0)
        MappTimer(M_TIMER_WAIT, &WaitTime);
    MappTimer(M_TIMER_READ+M_SYNCHRONOUS, &StartTime);
}

/* Close the object sequence file. */
MbufImportSequence(OBJECT2_SEQUENCE_FILE, M_DEFAULT, M_NULL, M_NULL, NULL, M_NULL,
                  M_NULL, M_CLOSE);

/* Allocate images for depth map. */
MbufAlloc2d(MilSystem, DEPTH_MAP_SIZE_X, DEPTH_MAP_SIZE_Y, 16+M_UNSIGNED,
            M_IMAGE+M_PROC+M_DISP, &DepthMapId);

/* Set depth map generation parameters. */
M3dmapControl(ScanId, M_DEFAULT, M_FILL_MODE, M_X_THEN_Y);
M3dmapControl(ScanId, M_DEFAULT, M_FILL_SHARP_ELEVATION, M_MIN);
M3dmapControl(ScanId, M_DEFAULT, M_FILL_SHARP_ELEVATION_DEPTH, GAP_DEPTH);
M3dmapControl(ScanId, M_DEFAULT, M_PIXEL_SIZE_X, SCALE_X);
M3dmapControl(ScanId, M_DEFAULT, M_PIXEL_SIZE_Y, SCALE_Y);
M3dmapControl(ScanId, M_DEFAULT, M_GRAY_LEVEL_SIZE_Z, SCALE_Z);
M3dmapControl(ScanId, M_DEFAULT, M_WORLD_OFFSET_X, WORLD_OFFSET_X);
M3dmapControl(ScanId, M_DEFAULT, M_WORLD_OFFSET_Y, WORLD_OFFSET_Y);
M3dmapControl(ScanId, M_DEFAULT, M_WORLD_OFFSET_Z, WORLD_OFFSET_Z);

/* Get depth map from accumulated information in the result buffer. */
M3dmapExtract(ScanId, DepthMapId, M_NULL, M_CORRECTED_DEPTH_MAP, M_DEFAULT, M_DEFAULT);

/* Compute the volume of the cookie using simple image analysis functions. */
Volume = ComputeVolume(MilSystem, DepthMapId);

MosPrintf(MIL_TEXT("3d data of the cookie is displayed.\n\n"));

```



```

#if M_MIL_USE_WINDOWS
/* Try to allocate D3D display. */
MIL_DISP_D3D_HANDLE DispHandle;
DispHandle = MdispD3DAlloc(DepthMapId, M_NULL,
                           D3D_DISPLAY_SIZE_X,
                           D3D_DISPLAY_SIZE_Y,
                           SCALE_X,
                           SCALE_Y,
                           -SCALE_Z);

    if (DispHandle != NULL)
    {
        /* Hide Mil Display. */
        MdispControl(MilDisplay, M_WINDOW_SHOW, M_DISABLE );
        MdispD3DShow(DispHandle);
        MdispD3DPrintHelp(DispHandle);
    }
    else
#endif
    {
        MdispControl(MilDisplay, M_VIEW_MODE, M_AUTO_SCALE);
        MdispSelect(MilDisplay, DepthMapId);
    }

    MosPrintf(MIL_TEXT("Volume of the cookie is %4.1f cm^3.\n\n"), Volume/1000.0);
    MosPrintf(MIL_TEXT("Press <Enter> to end.\n\n"));
    MosGetch();

#if M_MIL_USE_WINDOWS
    if (DispHandle != M_NULL)
    {
        MdispD3DHide(DispHandle);
        MdispD3DFree(DispHandle);
    }
#endif

    /* Free all allocations. */
    M3dmapFree(ScanId);
    M3dmapFree(LaserId);
    McalFree(MilCalibration);
    MbufFree(DepthMapId);
    MbufFree(MilImage);
}

#define MIN_HEIGHT_THRESHOLD -1.0 /* in mm, everything below this is considered as noise. */

/* Computes the volume of everything above ground level in a depth map. */
MIL_DOUBLE ComputeVolume(MIL_ID MilSystem, MIL_ID DepthMapId)
{
    MIL_ID      MilStatResult;    /* Stat result identifier. */
    MIL_DOUBLE  Volume,          /* Volume to compute in mm^3. */
               MaxValidValue;    /* Maximum valid depth map value. */
    MIL_INT     SizeBit;         /* Bits per pixel of depth map. */

```

```

/* Biggest possible value is considered invalid, so last valid value is 254 or 65534. */
MbufInquire(DepthMapId, M_SIZE_BIT, &SizeBit);
MaxValidValue = (MIL_DOUBLE)((1 << SizeBit) - 2);

/* Sum all gray levels to obtain the volume. */
MimAllocResult(MilSystem, M_DEFAULT, M_STAT_LIST, &MilStatResult);
MimStat(DepthMapId, MilStatResult, M_SUM, M_IN_RANGE, MIN_HEIGHT_THRESHOLD/SCALE_Z,
                                               MaxValidValue);
MimGetResult(MilStatResult, M_SUM, &Volume);
MimFree(MilStatResult);

/* Rescale the volume to express it in mm^3. */
Volume *= fabs(SCALE_X * SCALE_Y * SCALE_Z);

return Volume;
}

```

Chapter

17

Color processing and analysis

This chapter explains how to use the MIL Color Analysis module to perform color-based operations such as matching, projection, and distance. This chapter also explains how to process and convert colors.

Color processing and analysis overview

Although most cameras can acquire color images, most analysis modules only operate on monochrome images. In these cases, you must do one of the following: convert the color to grayscale, create a child buffer from one of the three color bands, or copy one of the three color bands to a 1-band buffer. Using MIL-Lite, you can perform these operations and store the transformed data in a monochrome buffer. This data can then be passed to an analysis module, which typically requires the full version of MIL.

You can, however, implement advanced color processing and analysis on color images directly, with the MIL Color Analysis module. You can use this module to perform the following color-based operations:

- **Distance.** The distance operation allows you to calculate the difference in color between two images or to calculate the distance between a color image and a color constant. This can be useful to, for example, visualize the difference in color between images, or detect color defects.



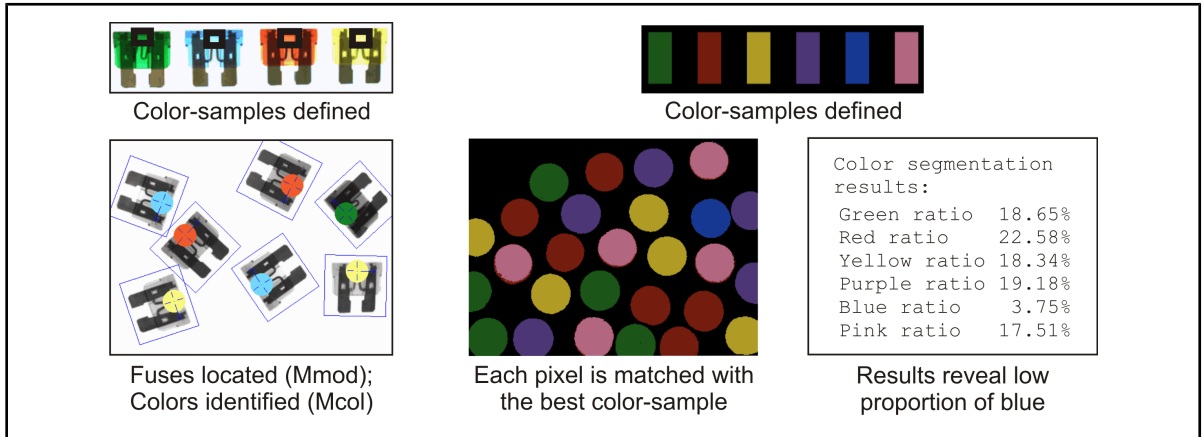
Visualize difference



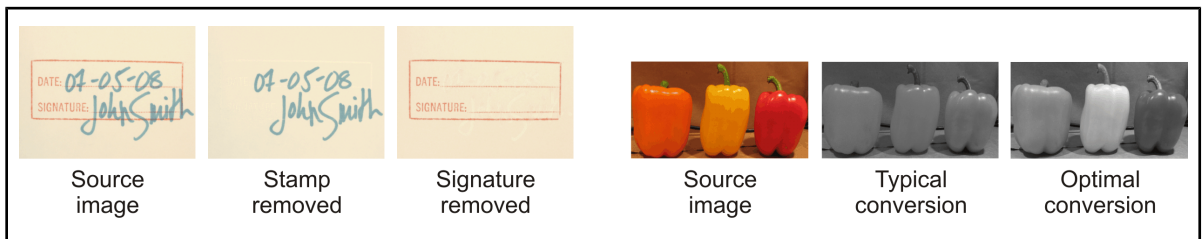
Detect defects

- **Matching.** The matching operation allows you to define color-samples and use them for color identification or supervised color segmentation. This can be useful to, for example, identify the color in an object (which can be found with a pattern

recognition module, such as MIL Model Finder), or segment colors to determine the percentage of color in an image.



- **Projection.** The projection operation allows you to separate a selected color from a rejected one, convert color images to grayscale, or calculate an image's covariance matrix or its principal color components. This can be useful to, for example, discard an unwanted color from an image, or perform a mathematically optimal color-to-grayscale transformation.



❖ The full version of MIL is required to use the MIL Color Analysis module.

Basic concepts

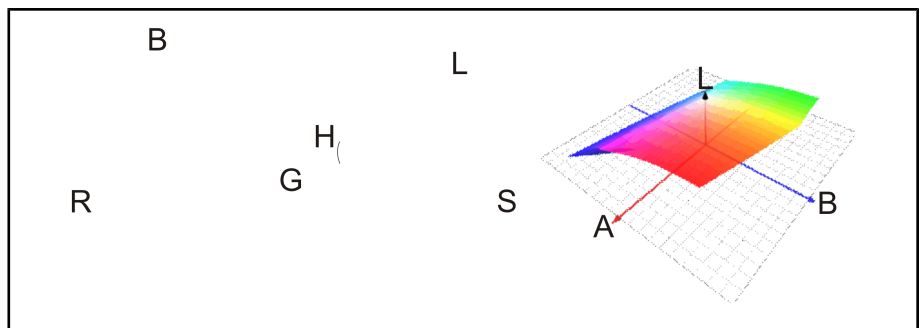
The basic concepts and vocabulary conventions for the Color Analysis module are:

- **Area identifier image.** An image, provided with the matching operation, that specifies the target areas.
- **Background pixels.** Pixels outside the target areas.
- **Best-matched color-sample.** The color-sample that most matches a target area's color, with respect to all color matching constraints.
- **Color distance.** The numerical difference between two colors.
- **Color identification.** Matching the color of each target area with the best color-sample.
- **Color matching context.** The container for all color-samples and match settings.
- **Color-sample.** The information, in the color matching context, that defines the color to be matched by the target area.
- **Color space.** A mathematical model, typically containing 3 components (for example, RGB, HSL, LAB), with which to describe colors.
- **Color space encoding.** Color transformation from the range represented in an image buffer (for example, 0 to 255) to its native (theoretical) range (for example, all real numbers between 0 and 1).
- **IEC.** Refers to the International Electrotechnical Commission, a globally-recognized standards organization for the field of electrotechnology.
- **Match score.** A measure of similarity between the color of the target area and the color of the color-sample.
- **Outliers.** Pixels, within a target area, that do not match with any color-sample.

- **Principal component.** A vector, within a color space, that represents the direction of the greatest degree of color variation in an image.
- **Reference color space.** The standard used to interpret the color space data; for example, sRGB.
- **Relevance score.** A measure of confidence associated to the match score.
- **sRGB.** Standard RGB specifications, as defined by the IEC Project Team 61966-2-1.
- **Supervised color segmentation.** Matching the color of each target area pixel with the best color-sample.
- **Target.** The image with which to match color-samples.
- **Target area.** A section of the target with which to match a color-sample.
- **Triplet.** A color-sample defined with three explicit color component values.
- **Working color space.** The color space with which the MIL Color Analysis module interprets color.

Color spaces and converting between them

A color space is a mathematical model, typically containing 3 components (for example, RGB), with which to describe color data. The color matching context supports the following color spaces: RGB, HSL, and CIELAB.



All the color data that you provide to the Color Analysis module must have a consistent format. For example, you will not receive an error if you match an RGB target area with an HSL color-sample, or a 16-bit target area with an 8-bit color-sample. All color data is used as is; there is no automatic conversion. Color data is interpreted band per band, independently of the color buffer's format. For matching, color data is considered to be within the context's color space, which you can set with **McolAlloc()**.

When matching, you must set the color space encoding according to the actual dynamic range of your color data, using **McolControl()** with **M_ENCODING**. By default, an 8-bit color space encoding is assumed, regardless of the depth of your buffers. If this default is not appropriate, you should modify the encoding control accordingly. For example, if your color data was acquired with a 16-bit camera, you should set **M_ENCODING** to **M_16BIT**. For more information, see the *Advanced color matching settings* section in *Chapter 17: Color processing and analysis*.

Since color data is used as is, you must be careful when restoring color images into an automatically allocated data buffer using **MbufRestore()**, where the color's type is not explicitly known. For example, if you are working in an RGB color space, and use a restored image file that did not contain RGB data (for example, it contained YUV data), you can end up with unexpected results. Instead, you should use **MbufAllocColor()** to allocate a color data buffer with an explicit data type (for example, **M_RGB24**), and then use **MbufLoad()** to load the data from the color image into the specified color data buffer. In this case, MIL will convert the image's color data to the proper type, as defined by the buffer (for example, **M_RGB24**).

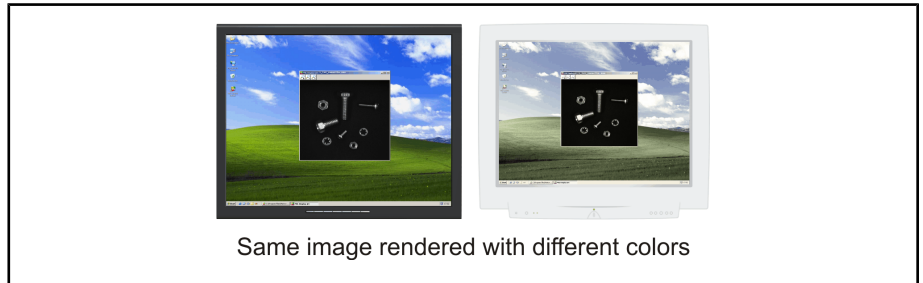
Color space

The color space represents all possible colors from black to white. For example, the RGB color space can be seen as a cube with a red, green, and blue axis. If a color is located at the origin, [0, 0, 0], its value is black. If a color is located at the maximum limit, such as [255, 255, 255] for an 8-bit image, its value is white. All other colors are in between this range and represented as 3 coordinate values within this cube. Generally, color images are in RGB format.

RGB

RGB is based on red, green, and blue color component (band) values. Typically, these components are directly used for acquiring and displaying color. For example, when displaying a color image buffer, the first band is routed to the monitor's first output channel (usually red), the second band to the second channel (usually green), and the third band to the third channel (usually blue).

Acquisition and display devices can render RGB data differently. Since RGB maps to such devices, it is a device-dependent color space.



Theoretically, there are as many RGB color spaces as there are color devices. Though there will always be some variance, internationally accepted standards (reference color spaces) by which color devices should adhere have therefore been created. To interpret the color space data, the MIL Color Analysis module uses standard RGB specifications (sRGB), as defined by the International Electrotechnical Commission (IEC) Project Team 61966-2-1.

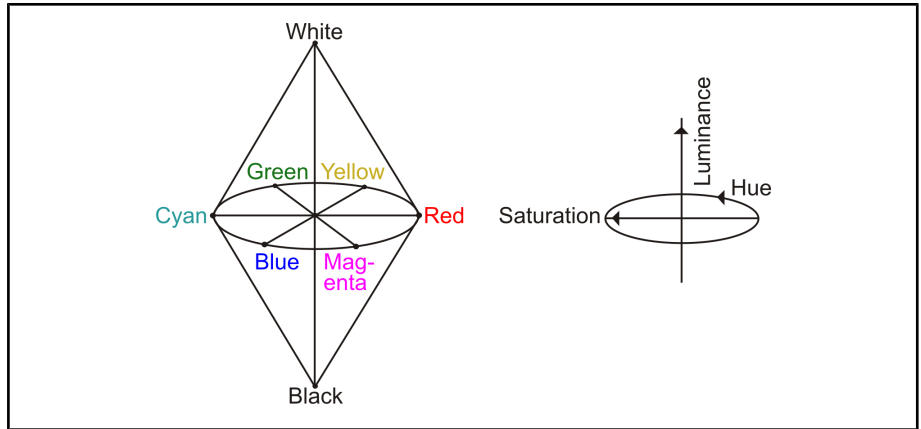
Typically, the reference color space only affects advanced applications, such as internally converting your color data to CIELAB before the match operation. For more information, see the *Advanced color matching settings* section in *Chapter 17: Color processing and analysis*.

HSL

HSL is based on hue, saturation, and luminance color component (band) values. HSL can be seen as a representation of points, within an RGB color space, designed to mimic the human way of describing colors. Like RGB, HSL is device-dependent.

In RGB, every color is a mixture of red, green, and blue, which can make it difficult to attain the exact component values of a particular color. However, in HSL, the color's actual hue is stored as a separate component (H), which is represented as an angular position on a circular color disk. The other components control only

the color's purity (S) and brightness (L), which can be used to alter the color's quality, but not the color's basic hue. This makes color manipulation much simpler with HSL. Most applications that allow an interactive manipulation of colors will therefore represent the color with HSL.

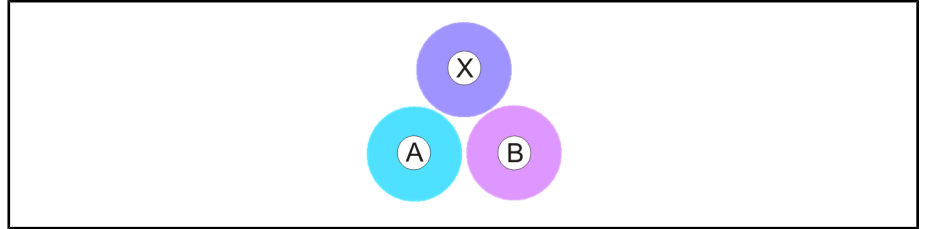


In some cases, this type of band independence can make HSL better to work with than RGB. You can, for example, match with the hue (color) band only, using **McolControl()** with **M_BAND_MODE** set to **M_COLOR_BAND_0**. This can solve certain problems, such as matching dark orange and bright orange, which can be difficult in RGB. Also, matching the hue independently of the luminance can be useful if your image has non-uniform lighting, shadows, or highlights.

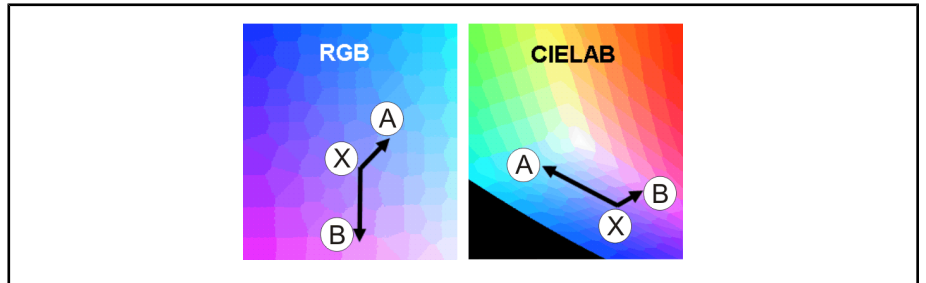
CIELAB

CIELAB (or LAB) is based on the color's luminance (L), its position between red and green (A), and its position between yellow and blue (B). Unlike RGB and HSL, LAB is intended to be device-independent, and was developed to represent a human interpretation of color by using statistical data taken from visual experiments. As such, LAB was designed to be perceptually uniform, making it a good space to measure color difference. That is, color differences in LAB vary proportionally with human perception. For example, if two colors are at a distance of 5, they will appear roughly 5 times as different as two colors at a distance of 1.

Since LAB is based on color perception, its mathematical model better represents how humans interpret color. This can be seen in the following example, where you have to choose which color, A or B, is closer to color X.



Mathematically, color A is the closest color, in an RGB color space. However, color B is intuitively closer, which is also what the distance in the LAB color space mathematically represents.



It can be preferable to use CIELAB over RGB since, like HSL, you can use CIELAB to discard the luminance, by matching with only band A and band B. Also, since CIELAB represents a human interpretation of color, it can be more robust with colors that are visually alike, especially for minor color differences. For example, when matching a red target among color-samples with similar shades of red, CIELAB can outperform RGB and HSL.

Since CIELAB relates to human perception, you might find color distances more meaningful than with RGB and HSL. For example, a color distance of 1 with CIELAB corresponds to the smallest possible color difference a human can perceive. With CIELAB, distances have been standardized by the International Commission on Illumination (CIE).

Converting between color spaces

You can convert between HSL and RGB color spaces using **MimConvert()** with **M_HSL_TO_RGB** or **M_RGB_TO_HSL**. For efficiency when converting from a 3-band (RGB) buffer, you can calculate just the hue component of the HSL color space into a 1-band buffer (**M_RGB_TO_H**). For more information on the algorithm used to convert between HSL and RGB color spaces, see the *Converting to grayscale* subsection in the *Converting to grayscale* section in *Chapter 17: Color processing and analysis*.

For internal calculations when working in an RGB color space, you can convert your RGB data to CIELAB before the color matching operation, using **McolSetMethod()**. This can be useful if, for example, you have allocated an RGB color space, but you want to occasionally calculate with CIELAB colors. For more information, see the *Advanced color matching settings* section in *Chapter 17: Color processing and analysis*.

- ❖ **MimConvert()** is available with MIL-Lite; **McolSetMethod()** requires the full version of MIL.

Converting to grayscale

Converting color images to grayscale can be very useful, since grayscale images are used with most processing and analysis modules, such as Blob Analysis, Model Finder, Metrology, and String Reader. For example, a picture of a license plate can be taken in color, but before it can be used in an Automatic Number Plate Recognition (ANPR) application written with String Reader, it must be converted to grayscale.

To convert color images to grayscale, you can either use **MimConvert()** to extract the luminance, or **McolProject()** to perform a principal component projection. **MimConvert()** is available with MIL-Lite, while **McolProject()** requires the full version of MIL.

Note that, in addition to converting your color images to grayscale, you might also want to process them, such as performing binary thresholding or blob analysis. This type of processing can enhance image quality and improve subsequent operations.

Extracting the luminance

You can use **MimConvert()** to convert color images to grayscale by extracting the luminance (intensity) from an RGB image (**M_RGB_TO_L**), or copying the luminance component of an image into a 3-band RGB buffer (**M_L_TO_RGB**), to create a monochromatic (gray) RGB buffer.

To convert between RGB and HSL color spaces, **MimConvert()** uses the following algorithm:

```
#define min(a, b) (((a) < (b)) ? (a) : (b))
#define max(a, b) (((a) > (b)) ? (a) : (b))

// Input and output between 0 and 1.
void RGBToHSL(float R, float G, float B, float* H, float* S, float* L)
{
    float MinVal, MaxVal, Delta;

    // Min and max values.
    MaxVal = max(R, max(G, B));
    MinVal = min(R, min(G, B));

    Delta = MaxVal - MinVal;

    // Compute the luminance.
    *L = 0.5f * (MaxVal + MinVal);

    if (Delta == 0.0f)
    {
        *S = *H = 0.0f;
    }
    else
    {
        // Compute the saturation.
        if (*L <= 0.5f)
        {
            *S = Delta / (MaxVal + MinVal);
        }
        else
        {
            *S = Delta / (2.0f - MaxVal - MinVal);
        }

        // Compute the hue.
        if (MaxVal == R)
        {
            *H = 60.0f * ((G - B) / Delta);
        }
        else if (MaxVal == G)
        {

```

```

        *H = 60.0f * (2.0f + ((B - R) / Delta));
    }
    else
    {
        *H = 60.0f * (4.0f + ((R - G) / Delta));
    }

    if (*H < 0.0f)
    {
        *H += 360.0f;
    }
}

// Remap the angle between 0 and 1.
*H = *H / 360.0f;
}

float HueToRGB(float Temp1, float Temp2, float Hue)
{
    if (Hue > 360.0f)
    {
        Hue = Hue - 360.0f;
    }
    else if (Hue < 0.0f)
    {
        Hue = Hue + 360.0f;
    }

    if (Hue < 60.0f)
    {
        return (Temp1 + (Temp2 - Temp1) * Hue / 60.0f);
    }
    else if (Hue < 180.0f)
    {
        return Temp2;
    }
    else if (Hue < 240.0f)
    {
        return (Temp1 + (Temp2 - Temp1) * (240.0f - Hue) / 60.0f);
    }

    return Temp1;
}

// Input and output between 0 and 1.
void HSLToRGB(float H, float S, float L, float* R, float* G, float* B)
{
    float Temp1, Temp2;

    // Remap the angle between 0 and 360.
    H = H * 360.0f;

    // Achromatic case.

```

```

if (S == 0.0f)
{
    *R = *G = *B = L;
}
else
{
    if (L <= 0.5f)
    {
        Temp2 = L * (1.0f + S);
    }
    else
    {
        Temp2 = L + S - (L * S);
    }

    Temp1 = 2.0f * L - Temp2;

    *R = HueToRGB(Temp1, Temp2, H + 120.0f);
    *G = HueToRGB(Temp1, Temp2, H);
    *B = HueToRGB(Temp1, Temp2, H - 120.0f);
}
}

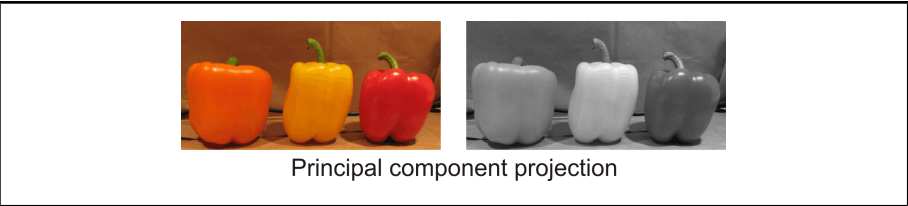
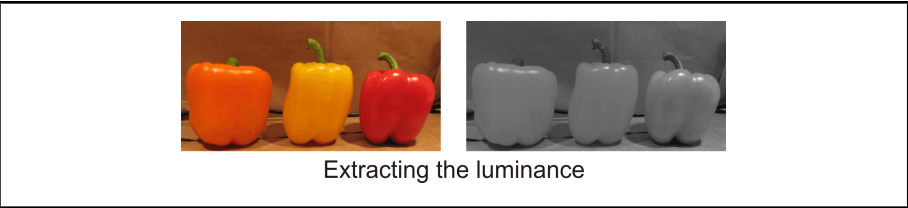
```

For more technical information, refer to *Computer Graphics: Principles and Practice in C*, James D. Foley et al., Addison-Wesley Publishing Company, United States, 1995.

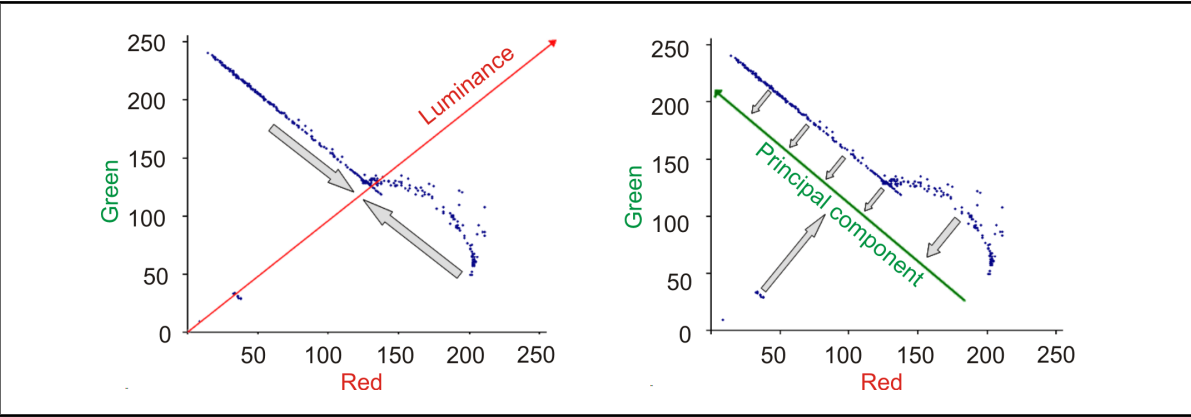
Principal component projection

You can convert a color image to grayscale using **McolProject()** with **M_PRINCIPAL_COMPONENT_PROJECTION**. Unlike **MimConvert()**, **M_PRINCIPAL_COMPONENT_PROJECTION** does not extract one band (the luminance component in HSL), which is actually independent of the color (the

hue component). Instead, **M_PRINCIPAL_COMPONENT_PROJECTION** uses the image's color data to calculate the best grayscale conversion possible, minimizing the loss of information. This results in a grayscale image that better differentiates the color in the original source image.

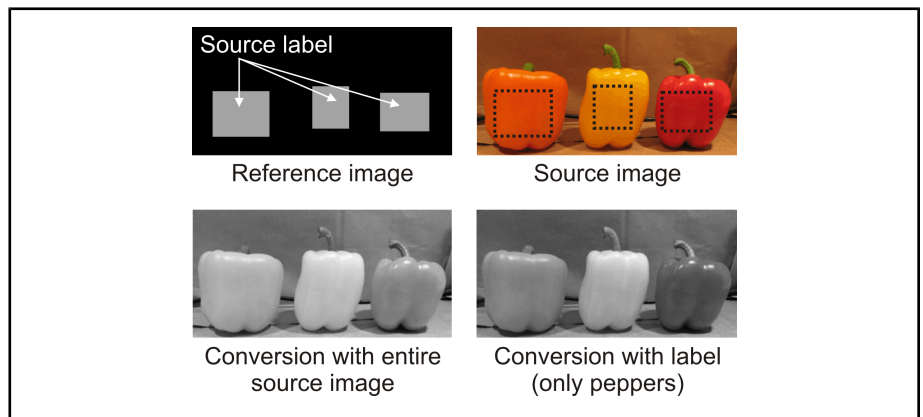


Extracting just the luminance can cause different colors to look the same in grayscale, which can make color analysis problematic. However, **McolProject()** calculates the color image's principal component vector, which is a line, within the color space, representing the greatest degree of color variation. Each color pixel is then projected to a point on this vector, resulting in a grayscale image that conveys more information than a luminance extraction.

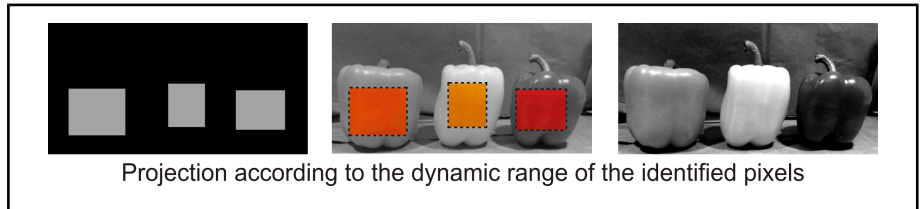


If the principal component vector is the same as the luminance vector (black-to-white), the principal component projection will be very similar to extracting the luminance band. This can happen if, for example, the number of different colors increases a lot. In this case, it can still be advantageous to calculate the principal component, since this vector, which was calculated from the data distribution itself, will be optimally oriented (black-to-white/white-to-black) for the projection calculation.

The principal component can either be calculated on the entire source image (**SrcId** parameter), or on a specified subset of pixels, which you can identify using a type of mask/reference image (**ColorsId** parameter), and setting the appropriate pixels to **M_SOURCE_LABEL**. By using such an image, you can choose which colors in the source will be used to define the principal component. This allows you to increase the difference between grayscale values.



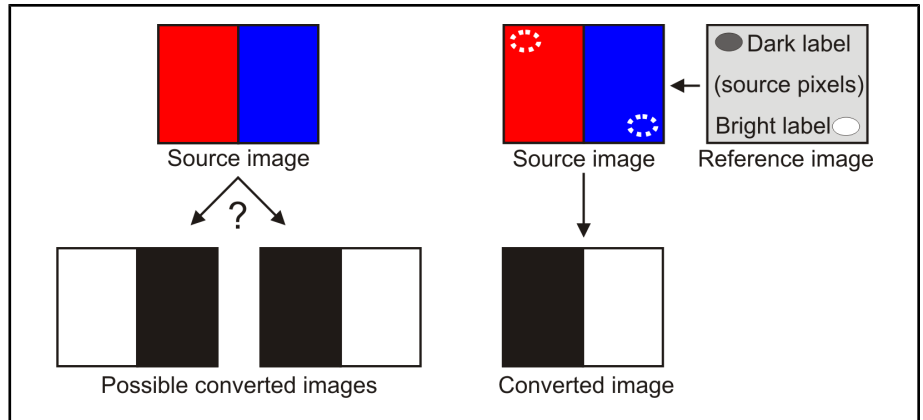
By default, the projection result is mapped according to the dynamic range of all the pixels in the source image, even if you are using a mask/reference image to calculate the principal component. If necessary, you can use **M_MASK_CONTRAST_ENHANCEMENT** to further improve the conversion to grayscale by mapping the projection according to the dynamic range of the source pixels identified with the mask/reference image. The result is similar to an increase in contrast.



Be careful when using **M_MASK_CONTRAST_ENHANCEMENT**, since source pixels that are outside the dynamic range of the mask/reference image might be saturated, and can cause unpredictable results. That is, the resulting image might not adequately represent the source image, since unidentified pixels (background) could saturate, causing the background in the resulting image to differ from the background in the source image.

When calculating the principal component, colors are automatically projected between the brightest (white) and darkest (black) side of the grayscale palette. However, in some cases, it is difficult to automatically determine which pixels should be projected to the white side, and which should be projected to the black side. In such cases, the direction of the projection is arbitrarily chosen, and the

resulting status of the projection operation is **M_UNSTABLE_POLARITY** (**StatusPtr** parameter). However, you can use the mask/reference image to specify the corresponding pixels in the source to project to bright or dark, using **M_BRIGHT_LABEL** and **M_DARK_LABEL**.



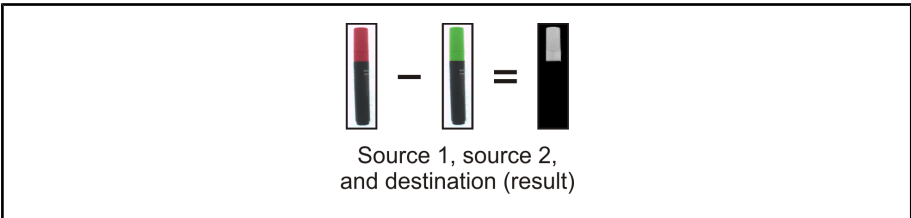
You can also specify a monochrome mask with **McolProject()**. The mask determines where to write the result of the projection operation. Only unmasked pixels (non-zero) will be written to, in the destination image. Since this is only a mask, it does not affect calculations. Note that you should not confuse this mask, set with the **DestMaskId** parameter, with the mask/reference image used to control how the source pixels are interpreted, set with the **ColorsId** parameter.

McolProject() can either produce the resulting grayscale image, or the transformation matrix to convert the image. You can use this matrix with **MimConvert()** to perform an optimal color-to-grayscale transformation of any 3-band color image. Be certain that the color distribution of all source images that use this matrix is similar, otherwise unpredictable results can occur.

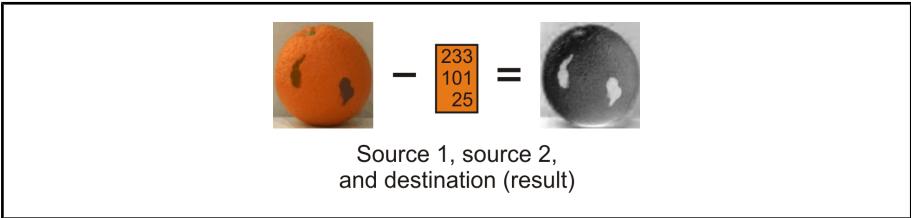
Distance between colors

Distances between colors can either be produced with **McolDistance()**, or when matching colors.

Using **McolDistance()**, you can calculate the point-to-point distance between colors in two sources. The first source must be an image, while the second source can be: an image, a color constant, a covariance matrix, or the covariance of a specified image. Results are written to the destination image.

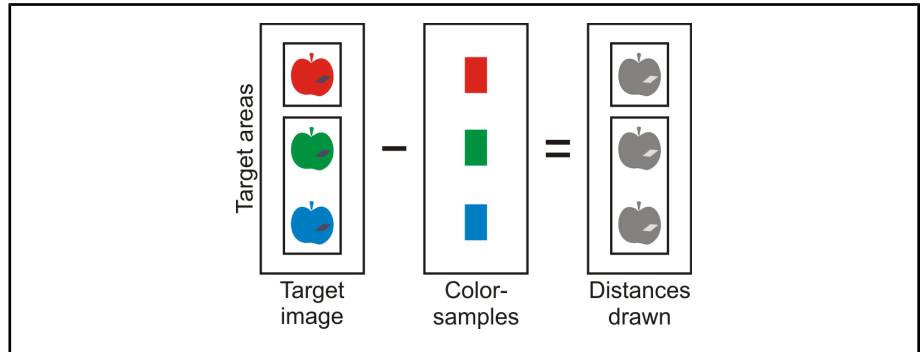


Calculating the color distance can be useful to, for example, find flaws in an image by comparing it to a perfect version of the image (golden template), or to a color constant.



To ignore unwanted pixels in the **McolDistance()** calculation, you can apply a mask. The color distance is typically calculated only for pixels within the intersection of the source, destination, and mask images, with the assumption of a common origin at the top-left corner.

Color distances are also used when matching colors. Therefore, you can retrieve distance results, after calling **McolMatch()**. The following image shows how distances can be drawn between the target areas and multiple color-samples. Note that this is unlike **McolDistance()**, which can only calculate the distance between 2 sources.



For more information on retrieving color distances when matching colors, see the *Image results* subsection in the *Color matching* section in *Chapter 17: Color processing and analysis*.

Differences in distances

The primary differences between distances calculated with **McolMatch()** and **McolDistance()** are:

- When the sources are images, **McolDistance()** calculates a point-to-point distance, while **McolMatch()** uses color statistics (average color).

When matching, the distance is actually calculated between each color-sample's statistic and each target area's statistic, or between each color-sample's statistic and each pixel in each target area, depending on the operation mode specified with **McolSetMethod()**.

- **McolDistance()** results are from two sources, as you have specified them. However, **McolMatch()** results can come from several color-samples, or even from background and outlying regions.

- If you are only interested in distance values, **McolDistance()** can be more convenient to use than **McolMatch()**, since there isn't a context, there isn't the matching process, and results are returned directly to the function.
- Unlike **McolDistance()**, **McolMatch()** takes the context's color space into account; it also offers more options than **McolDistance()**, such as converting to the CIELAB color space (using **McolSetMethod()** with **M_CIELAB**) and operating on specific color bands (using **McolControl()** with **M_BAND_MODE**).

Color distance types

Whether you are matching colors or using **McolDistance()**, the color distance will be calculated using one of the following distance types:

- Euclidean (**M_EUCLIDEAN**).
- Mahalanobis (**M_MANHATTAN**).
- Manhattan (**M_MAHALANOBIS**).

When matching, these types are set with **McolSetMethod()**, which also has a Delta-E distance type (**M_DELTA_E**), as defined by the International Commission on Illumination (CIE). A Delta-E color distance is the same as a Euclidean color distance, but specialized for the CIELAB color space.

With **McolDistance()**, distances are always calculated between the first and second source. If the sources are images, the distance is calculated pixel-by-pixel, unless otherwise specified.

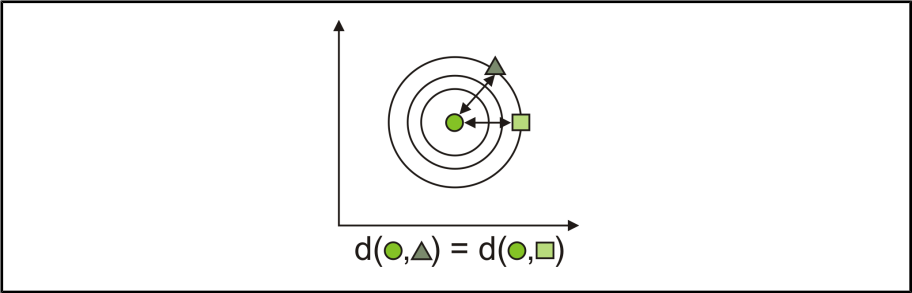
When matching, the distance is calculated between the target area and the color-sample, which are set with **McolMatch()** and **McolDefine()**. For explanation purposes, the target area can be seen as the first source, and the color-sample can be seen as the second source.

A Euclidean distance

A Euclidean distance is the square root of the sum of the squared differences between the color of the first source and the color of the second source. A Euclidean distance can be represented with the following formula:

$\sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$	<p>Where:</p> <ul style="list-style-type: none">• r_1 and r_2 represent the first component of the first and second source color.• g_1 and g_2 represent the second component of the first and second source color.• b_1 and b_2 represent the third component of the first and second source color.
--	--

The following example illustrates how the distance between a green point, indicated by a circle, and two other green points, indicated by a triangle and a square, is measured with a Euclidean calculation.



With **McolDistance()**, the first source must be an image, and the second source can be either an image or a color constant.

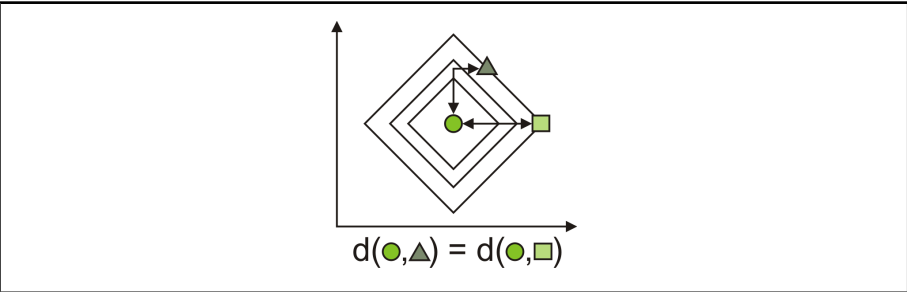
A Manhattan distance

A Manhattan distance is the sum of the absolute value of the differences between the color of the first source and the color of the second source. A Manhattan distance can be represented with the following formula:

$ r_1 - r_2 + g_1 - g_2 + b_1 - b_2 $	<div>Where:</div> <ul style="list-style-type: none"> • r_1 and r_2 represent the first component of the first and second source color. • g_1 and g_2 represent the second component of the first and second source color. • b_1 and b_2 represent the third component of the first and second source color.
---	--

For HSL colors, the distance between angular coordinates is equal to the smallest angular difference, rather than the absolute value of the difference.

The following example illustrates how the distance between a green point, indicated by a circle, and two other green points, indicated by a triangle and a square, is measured with a Manhattan calculation.



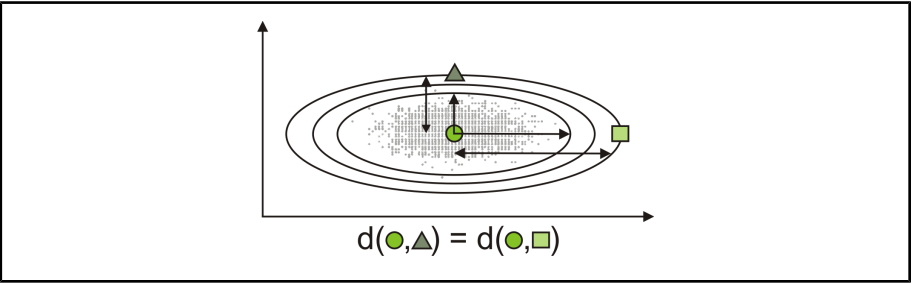
With `McolDistance()`, the first source must be an image, and the second source can be either an image or a color constant.

A Mahalanobis distance

A Mahalanobis distance is calculated between the color of the first source and the covariance of the second source. If you are using `McolDistance()` and the second source is an image, the covariance of that image, rather than the color of each pixel, is used to calculate the distance. If you are using `McolSetMethod()`, the covariance of the color-sample is used. A Mahalanobis distance can be represented with the following formula:

$\sqrt{([x-u]^t \sum^{-1} [x-u])}$	Where: <ul style="list-style-type: none">• x represents the first source color.• u represents the average of the second source color.• σ is for the covariance matrix of the second source color.
------------------------------------	---

The following example illustrates how the distance between a green point, indicated by a circle, and two other green points, indicated by a triangle and a square, is measured with a Mahalanobis calculation.



The distance calculated for Mahalanobis, between a color and a distribution of colors (covariance), is similar to a Euclidean distance between the mean of the two colors, but weighted by the inverse of the covariance of the distribution. This implies that the more a color distribution varies in a direction within the color space, the less important is the distance in that direction.

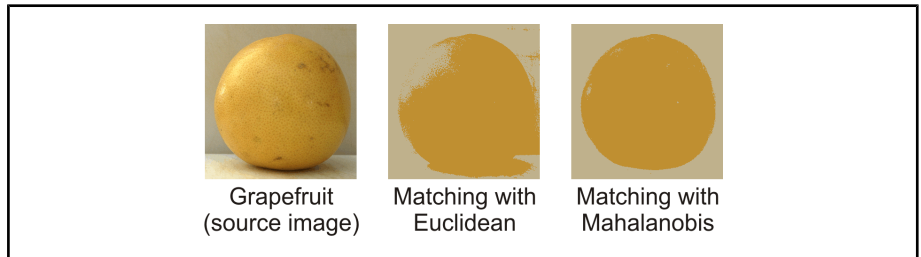
Unlike Euclidean and Manhattan distances, a Mahalanobis distance must be done between a point and a distribution.

Note that, though the color distance is typically calculated for pixels within the intersection of the source, destination, and mask images, this is not the case if you provide an image as the second source for **McolDistance()** (when you are using a Mahalanobis distance). The second source image does not take part in the intersection since that image's covariance matrix is calculated to obtain the distance.

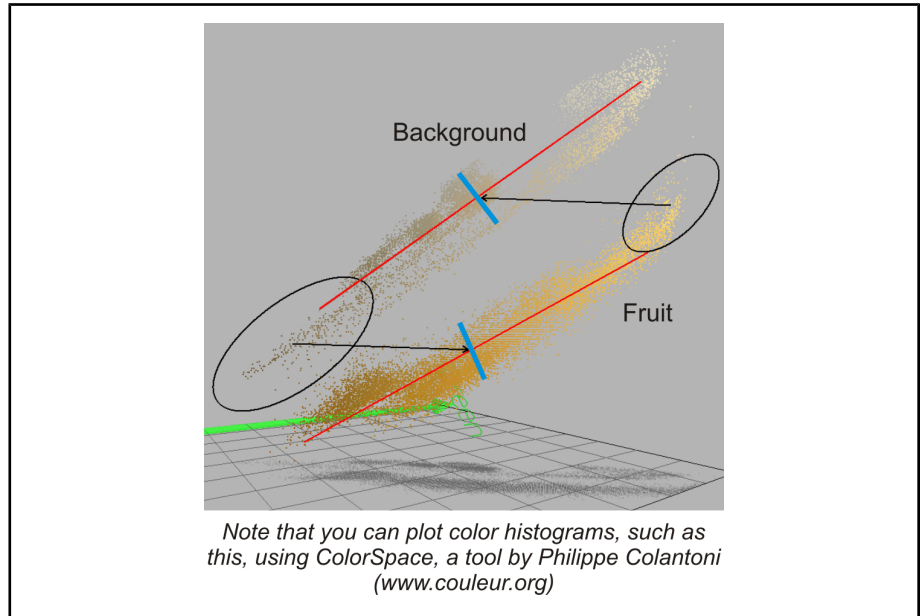
Choosing a distance type

Choosing the most appropriate distance type with which to calculate color distances depends on many factors, including the color space of your data, the background, and the particularities of your application. In general, a Euclidean distance should be used for RGB and CIELAB color spaces, while a Manhattan distance should be used for HSL. A Mahalanobis color distance should be used when dealing with closely-related colors, that are not HSL. Of course, these are only guidelines.

The following example illustrates an RGB source image of a grapefruit, which is to be used in a matching operation. For RGB colors, a Euclidean distance is typically sufficient. However, in this case, a Mahalanobis distance is preferable.



In the source image, the color of the background and some parts of the grapefruit are similar; this makes Mahalanobis yield better results, since the covariance of the image is used. To illustrate this point, the following image shows two groups of pixels displayed in RGB; one group is from the image's background, and the other is from the grapefruit.



For each group of pixels (background and grapefruit), this image shows:

1. The first principal component (the red lines), which represents the direction of greatest standard deviation.
2. The blue lines, whose intersection with the principal component (the red lines), is the mean color.

Encircled in black, on the left, are the background pixels that will match the grapefruit, being closer by Euclidean distance to the grapefruit's mean color. Encircled in black, on the right, are the grapefruit's pixels that will match the background, being closer by Euclidean distance to the background's mean color.

However, with Mahalanobis, any distance oriented parallel to the principal component (the red lines) will be scaled by the inverse of the standard deviation. Therefore the encircled pixels will match with the correct group (background or grapefruit), yielding a better matching result.

Color matching

Color matching is the process of finding a match between the color of a target area and the color of a predefined color-sample. You can affect this process in many ways, such as modifying the type of color distance (**McolSetMethod()**), the operation mode (**McolSetMethod()**), and various color matching controls (**McolControl()**).

The match is performed, in part, by calculating the color distance between the target area and the color-sample. This information can be retrieved after calling **McolMatch()**. However, if you are only interested in color distances, you can use **McolDistance()**. For more information, including the differences in distances between **McolMatch()** and **McolDistance()**, see the *Distance between colors* section in *Chapter 17: Color processing and analysis*.

Steps to performing color matching

The following steps provide a basic methodology for using the MIL Color Analysis module to match colors:

1. Allocate a color matching context to hold your color-samples and color matching settings, using **McolAlloc()** with **M_COLOR_MATCHING**.
2. Allocate a color matching result buffer to hold the color matching results, using **McolAllocResult()** with **M_COLOR_MATCHING_RESULT**.

This step is not required if you are calculating an image result directly, using **McolMatch()** with **M_DRAW_...** For more information, see the *Image results* subsection in this section.

3. Optionally, convert your colors to the appropriate color space, using **MimConvert()**. For example, you can convert from RGB to HSL.
4. Define the color-samples, using **McolDefine()**.

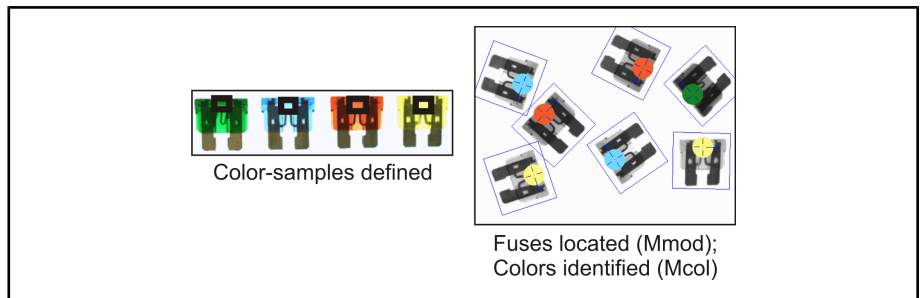
5. Optionally, change the operation mode and distance type used to perform the match operation, using **McolSetMethod()**.
6. Optionally, change context and result buffer settings, using **McolControl()**.
7. Preprocess the context, using **McolPreprocess()**.
8. Perform the color matching operation, using **McolMatch()** to calculate all results (**M_DEFAULT**).
9. Retrieve the required results from the result buffer, using **McolGetResult()**.
10. Draw image results, using **McolDraw()**.
11. Free all your allocated objects, using **McolFree()**.

The basics

Color matching can be used to perform one of two basic tasks: color identification or supervised color segmentation. In either case, the match is based on the defined color-samples, which can come from a source image or from explicit color values. For more information, see the *Defining and adding color-samples to your color matching context* subsection in this section.

Color identification

Color identification refers to matching the color of each target area with the best color-sample within a group of predefined color-samples. When performing color identification, you might find it useful to locate the target areas with the MIL Geometric Model Finder module before performing the match.



In this example, color-sample regions in a source image are used to define each possible fuse color. Fuses are then located with Model Finder, and their color is matched to the best color-sample, using **McolMatch()**. Note that to achieve this result, you must properly define the target areas in which to identify the colors. To produce image results for a target area (as illustrated above), use **McolMatch()** or **McolDraw()** with **M_DRAW_LABEL_AREA_IMAGE** or **M_DRAW_COLORED_LABEL_AREA_IMAGE**. For more information, see the *Area identifier image* subsection in the *Color matching* section in *Chapter 17: Color processing and analysis* and the *Image results* subsection in the *Color matching* section in *Chapter 17: Color processing and analysis*.

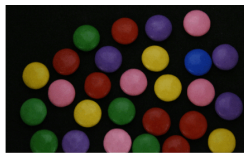
Supervised color segmentation

Supervised color segmentation refers to matching the color value of each pixel in the target area with the best predefined (supervising) color-sample. Since the match is done using each target pixel, you must use **McolSetMethod()** with **M_MIN_DIST_VOTE**, and produce image results on a pixel basis, using **M_DRAW_LABEL_PIXEL_IMAGE** or **M_DRAW_COLORED_LABEL_PIXEL_IMAGE**. For more information, see the *Operation mode* subsection in the *Color matching* section in *Chapter 17: Color processing and analysis* and the *Image results* subsection in the *Color matching* section in *Chapter 17: Color processing and analysis*.

By matching the color value of each pixel with the best predefined color-sample, you can use the Color Analysis module to separate objects by their color information. For example:



Color-samples defined

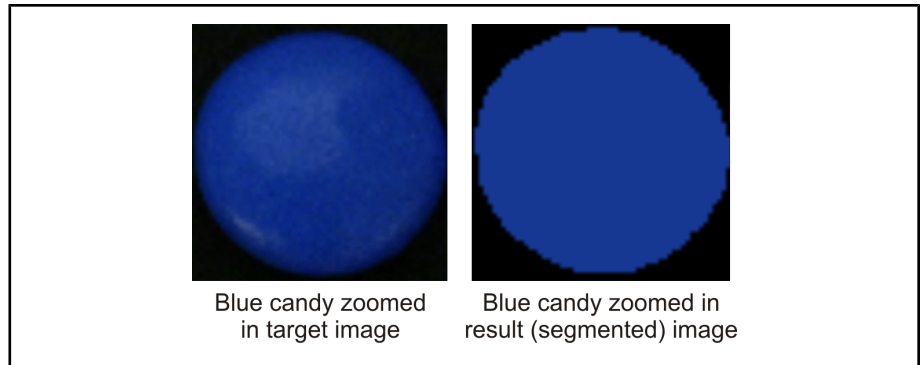


Target image

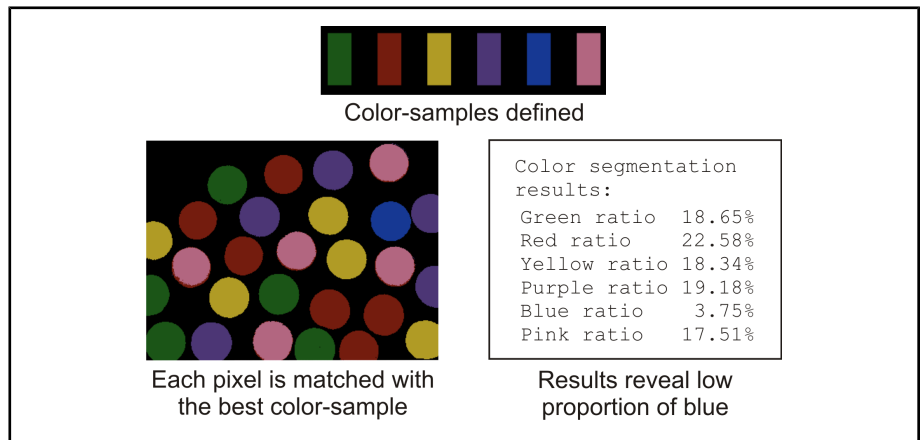


Result (segmented) image

As you can see, the colors of the objects in these two images are similar; however, in the result (segmented) image, each pixel value corresponds to the color of the best-matched color-sample, rather than the actual color of the object in the target area.



This type of matching is useful to, for example, retrieve the relative presence of each color in an image, which allows you to identify the colors that have been more or less produced. This is also referred to as the spatial coverage for each of the colors.



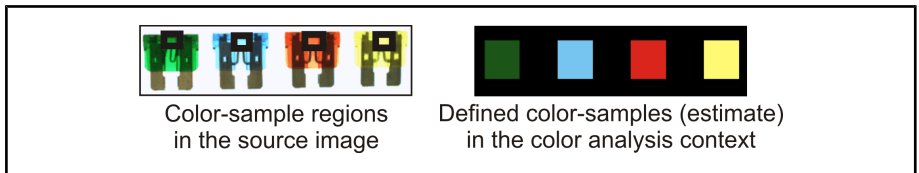
Defining and adding color-samples to your color matching context

You can either add a color-sample from an image (**M_IMAGE**), or from three explicit color component values (**M_TRIPLET**), using **McolDefine()**. Color-samples are added to the color matching context. You can also delete color-samples from the context, using **McolDefine()**. You must add color-samples one at a time; however you can delete all color-samples at once.

The color-sample index starts at 0, and each subsequent color-sample added to the color matching context is given a sequential index number, in the order that the color-sample was added. If a color-sample is deleted, all entries with higher indices are shifted down by one. You can also assign a label to a color-sample, or it can be assigned automatically. To change it afterwards, use **McolControl()** with **M_SAMPLE_LABEL_VALUE**. When you add or delete a color-sample, you must preprocess the color matching context (**McolPreprocess()**) before any subsequent call to **McolMatch()**.

When adding a color-sample with **M_IMAGE**, you must specify a source image, and within it, a rectangular color-sample region. You can set the entire source image as the color-sample region by setting all the required parameters to **M_DEFAULT**. For good match results, use the best source image possible. If required, you can also apply a mask to an image-type color-sample, using **McolMask()**. A mask is useful for ignoring unwanted data, or dealing with non-rectangular regions. Note that, when adding samples with **M_TRIPLET**, you must not specify a source image.

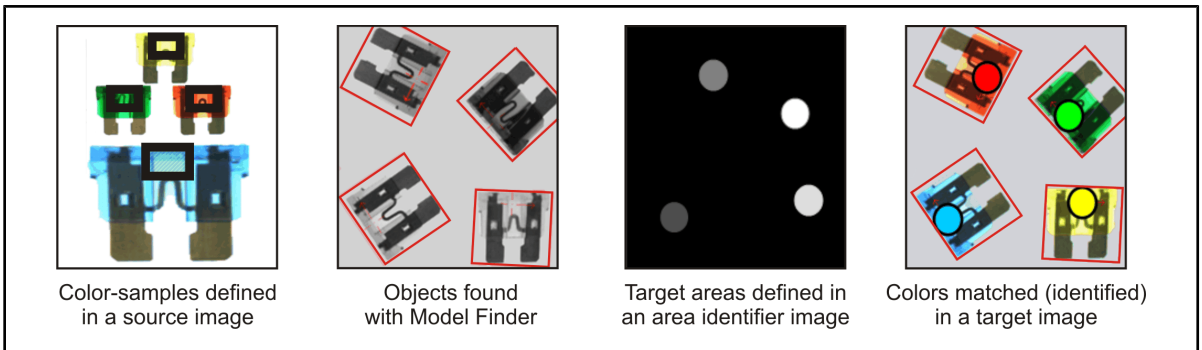
For color-samples defined from an image, an estimation of the color is determined to represent the color-sample. This estimate is based on color statistics, such as the mean; this estimate is considered the color of the color-sample and is used in the match. The size of the color-sample does not affect the match operation.



Area identifier image

The area identifier image is a grayscale image, specified with **McolMatch()**, that defines the independent target areas where color matching will occur in the target image. You must use an area identifier image to match multiple target areas. Each unique, non-zero label in the area identifier image defines an independent target area. Typically, the area identifier image is the same size as the target image. To match the entire target image as one target area, set the area identifier image to **M_NULL**.

In the following example, the 4 target areas in the area identifier image are defined according to the objects found with the MIL Model Finder module. The color of each corresponding target area, in the actual target image, is then matched (identified) with the best color-sample, using **McolMatch()**.



To produce image results on an area basis (as illustrated above), use **McolMatch()** or **McolDraw()** with **M_DRAW_LABEL_AREA_IMAGE** or **M_DRAW_COLORED_LABEL_AREA_IMAGE**. For more information, see the *Image results* subsection in this section.

Operation mode and distance type

Before calling `McolMatch()`, you can use `McolSetMethod()` to set the match's operation mode to either `M_STAT_MIN_DIST` (default) or `M_MIN_DIST_VOTE`. The operation mode specifies how to use the distance in color between the color-sample and the target area when matching colors.

You can also use `McolSetMethod()` to set the type of distance with which to perform the match. For RGB and CIELAB color spaces, a Euclidean color distance is calculated by default. For HSL color spaces, Manhattan is the default. For more information, see the *Color distance types* subsection in the *Distance between colors* section in *Chapter 17: Color processing and analysis*.

For either operation mode, the color of the color-sample used in the match will either be the average color (for `M_IMAGE`) or the color component values (for `M_TRIPLET`).

For `M_STAT_MIN_DIST`, the average color of the target area is used in the match. For `M_MIN_DIST_VOTE`, the color of each pixel in the target area is used.

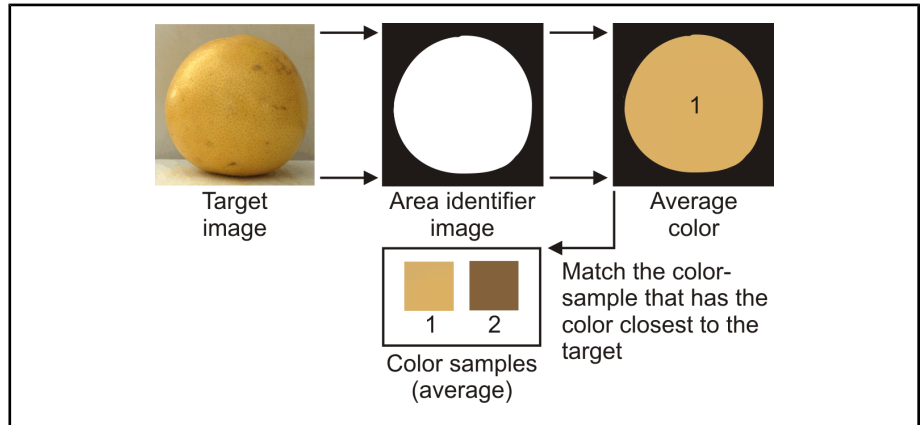
Regardless of the operation mode chosen, the distance tolerance (`McolControl()` with `M_DISTANCE_TOLERANCE`) and the acceptance level (`McolControl()` with `M_ACCEPTANCE`) can affect the color-samples that are matched. For more information, see the *Acceptance* subsection in this section and the *Tolerance* subsection in this section.

M_STAT_MIN_DIST

When using `M_STAT_MIN_DIST`, color statistics are calculated (typically the mean/average) for each target area and for each color-sample defined in the context, and the color distance between each target area and each color-sample is determined.

The resulting distances determine the score of the color-samples. The closer the colors, the higher the score. If a distance is not within a color-sample's distance tolerance (`McolControl()` with `M_DISTANCE_TOLERANCE`), the color-sample's score is 0%. The color-sample with the highest score above the acceptance (`McolControl()` with `M_ACCEPTANCE`) is the target area's best-matched color-sample.

In the following example, the target area's average color is calculated, and then matched with color-sample 1, which is the closest (best-match) color.

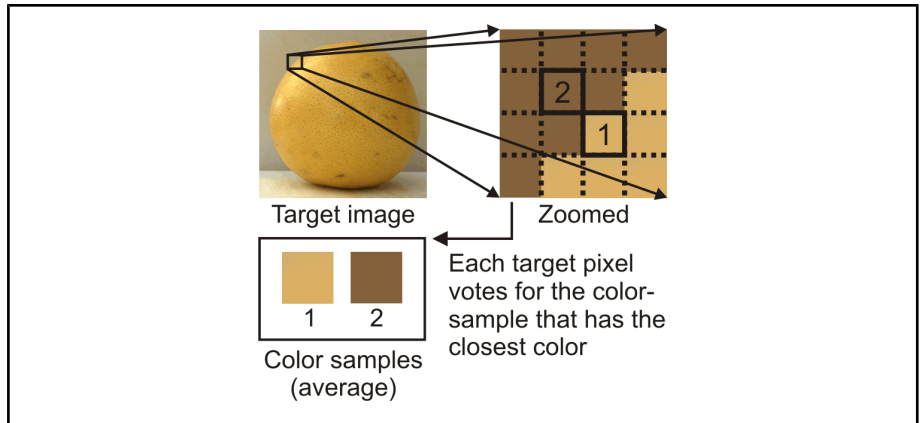


M_MIN_DIST_VOTE

When using **M_MIN_DIST_VOTE**, color statistics are calculated (typically the mean/average) for each color-sample defined in the context, and the color distance between each pixel in each target area and each color-sample is determined. Each target pixel then votes for the color-sample with the closest color, and which is also within the distance tolerance (**McolControl()** with **M_DISTANCE_TOLERANCE**).

The number of votes that a color-sample accumulates determines its score. The greater the number of votes, the higher the score. The color-sample with the highest score above the acceptance (**McolControl()** with **M_ACCEPTANCE**) is the target area's best-matched color-sample.

In the following example, each target pixel votes for the color-sample that has the closest color. In general, grapefruit pixels will vote for color-sample 1, while background pixels will vote for sample 2. Since there are more grapefruit pixels, color-sample 1 is the best-match.



Acceptance

The acceptance levels determine the minimum scores required for a successful match between a target area and a color-sample. You can set an acceptance level for the color-sample's match score, and for the target area's relevance score, using `McolControl()` with `M_ACCEPTANCE` and `M_ACCEPTANCE_RELEVANCE`.

Color-sample acceptance (for the color-sample's match score)

`M_ACCEPTANCE` is applied to the match score (`McolGetResult()` with `M_SCORE`), which indicates the similarity between the color of the color-sample and the color of the target area. The higher the acceptance, the closer the colors must be for them to match.

For example, if you are using an `M_STAT_MIN_DIST` operation mode (`McolSetMethod()`), and you set `M_ACCEPTANCE` to 100, the colors will only match if they are absolutely the same. For an acceptance of 100 while using an `M_MIN_DIST_VOTE` operation mode, all pixels in the target area that are not outliers must have voted for the same color-sample to have a successful match.

For more information on the actual match score calculated, see the *Match score and relevance score* subsection in the *Color matching* section in *Chapter 17: Color processing and analysis*.

Relevance acceptance (for the target area's relevance score)

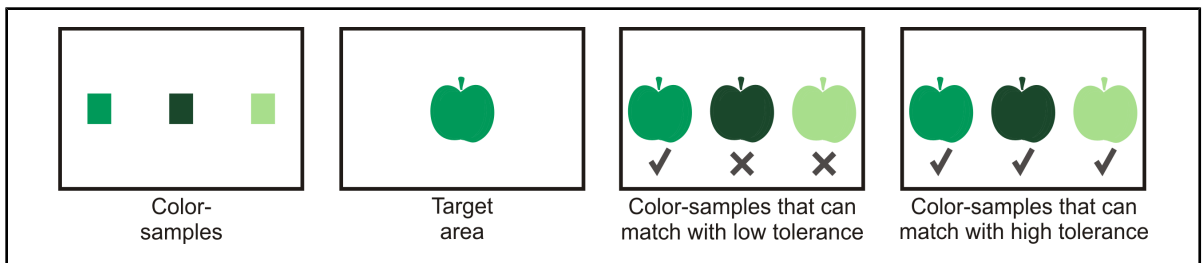
M_ACCEPTANCE_RELEVANCE is applied to the target area's relevance score (**McolGetResult()** with **M_SCORE_RELEVANCE**), which indicates the significance (relevance) of the match score (**M_SCORE**). In statistics, this is similar to the confidence level. The higher the relevance acceptance, the more significant the match score results must be (relative to other possible results), to have a successful match for the target area.

For example, if you are using an **M_STAT_MIN_DIST** operation mode (**McolSetMethod()**), and you set **M_ACCEPTANCE_RELEVANCE** to 100, the match will only be successful if the best-matched color-sample has a match score that is vastly superior to that of all other color-samples. For a relevance acceptance of 100 while using an **M_MIN_DIST_VOTE** operation mode, all pixels in the target area must have voted (no outliers) to have a successful match.

For more information on the actual relevance score calculated, see the *Match score and relevance score* subsection in the *Color matching* section in *Chapter 17: Color processing and analysis*.

Distance tolerance

The distance tolerance refers to the maximum color distance between the color-sample and the target area allowed for a successful match. The greater the distance tolerance, the greater the distance (difference) between matching colors can be. For example, if your target area is green, you can set the distance tolerance to 0 to only match with the exact same green. However, by increasing the tolerance, you can match with colors that are progressively different than the original.



To specify the distance tolerance, use **McolControl()** with **M_DISTANCE_TOLERANCE** set to **M_INFINITE**, a specific value, or **M_AUTO** (the default). When setting the tolerance, you must consider the specified distance type, the color space, and the color space encoding. For example, a distance tolerance of 1.0 when using an **M_MAHALANOBIS** distance type is not the same as when using an **M_MANHATTAN** distance type.

When setting a specific value, or when using **M_AUTO**, the distance tolerance is determined according to the tolerance mode, which you can set with **M_DISTANCE_TOLERANCE_MODE**.

If the tolerance mode is set to **M_RELATIVE**, the distance tolerance is calculated as a relative distance between all color-samples. In this case, the smallest distance between two color-samples is computed internally; the distance tolerance value is multiplied by half this distance. When using **M_RELATIVE**, the default distance tolerance (**M_AUTO**) is 1.

If the tolerance mode is set to **M_SAMPLE_STDDEV**, the distance tolerance is calculated as the color-sample's standard deviation of the distance (between each pixel and the average color of the color-sample). In this case, the distance tolerance value is multiplied by the distance's standard deviation. When using **M_SAMPLE_STDDEV**, the default distance tolerance (**M_AUTO**) is 3.

If the tolerance mode is set to **M_ABSOLUTE** (the default), the distance tolerance value is applied as is. When using **M_ABSOLUTE**, the default distance tolerance (**M_AUTO**) is **M_INFINITE**.

Basic results

Basic results refer to the results available with **McolGetResult()**, such as the:

- Best-matched color-sample.
- Match status.
- Match score and relevance score.
- Outlier coverage and color-sample coverage.
- Color distance.

To retrieve such results, you must have called **McolMatch()** with **M_DEFAULT** (**ControlFlag** parameter) to calculate all results, instead of just a specific image (**M_DRAW_...**). For more information on calculating image results, see the *Image results* subsection in this section.

Best-matched color-sample

You can either return the index or the label of each target area's best-matched color-sample, using **M_BEST_MATCH_INDEX** or **M_BEST_MATCH_LABEL**. If no color-sample has matched, -1 is returned for **M_BEST_MATCH_INDEX**; for **M_BEST_MATCH_LABEL**, the value you have set using **McolControl()** with **M_OUTLIER_LABEL** is returned.

Match status

You can either return the match status of a color-sample, using **M_SAMPLE_MATCH_STATUS**, or the match status of a target area, using **M_STATUS**.

The status of a color-sample (**M_SAMPLE_MATCH_STATUS**) returns **M_MATCH** if that color-sample fulfills the match conditions, with respect to the acceptance (**M_ACCEPTANCE**) and the distance tolerance (**M_DISTANCE_TOLERANCE**); otherwise, **M_NO_MATCH** is returned. The status of a target area (**M_STATUS**) returns **M_SUCCESS** if at least one color-sample fulfills the match conditions, and if the target area's relevance score passes the relevance acceptance level (**M_ACCEPTANCE_RELEVANCE**); otherwise, **M_FAILURE** is returned.

The color-sample with the highest score (**M_SCORE**) is referred to as the best-matched color-sample. To determine if a specific color-sample is the best-matched color-sample, compare the color-sample's index or label with **M_BEST_MATCH_INDEX** or **M_BEST_MATCH_LABEL**.

Match score and relevance score

The color-sample's match score is based on the operation mode specified, using **McolSetMethod()**:

- **M_STAT_MIN_DIST**.

$MatchScore = [k / (k + Distance)] \times [(MaxDistance - Distance) / MaxDistance]$, where *Distance* refers to the current color-sample distance and *MaxDistance* refers to the maximum distance possible in the working color space.

- **M_MIN_DIST_VOTE.**

$MatchScore = NumberOfVotes / \text{sum}(NumberOfVotes)$, where *NumberOfVotes* refers to the current color-sample's number of votes, and the sum is over the number of votes for all color-samples.

Similarly, the target area's relevance score is also based on the operation mode specified, using **McolSetMethod()**:

- **M_STAT_MIN_DIST.**

$RelevanceScore = (BestDistance)^{-1} / \text{sum}(Distance^{-1})$, where *BestDistance* is the distance of the best-matched color-sample, and the sum is over all color-samples that have been matched by the target area.

- **M_MIN_DIST_VOTE.**

$RelevanceScore = \text{sum}(NumberOfVotes) / NumberOfPixelsInTheTargetArea$, where *NumberOfVotes* refers to the current color-sample's number of votes, and the sum is over the number of votes for all color-samples.

A low relevance score indicates that you should be cautious about the match results, even if the match score is high. For example, a high match score and a low relevance score could mean that the target area came very close to matching with a different color-sample, which implies that a slightly different target image, or even a difference in lighting, could change your results. You should therefore set appropriate acceptance levels for each of these scores. For more information, see the *Acceptance* subsection in the *Color matching* section in *Chapter 17: Color processing and analysis*.

Outlier coverage and color-sample coverage

The outlier coverage (**M_OUTLIER_COVERAGE**) quantifies, as a percentage, the proportion of pixels in the target area that did not vote for any color-sample, while the sample coverage (**M_SAMPLE_COVERAGE**) quantifies the proportion of pixels that did vote for a specific color-sample.

Coverage results are only available when using an **M_MIN_DIST_VOTE** operation mode.

Color distance

The color distance (**M_COLOR_DISTANCE**) returns the distance between the color of the target area and its best-matched color-sample, when using an **M_STAT_MIN_DIST** operation mode.

You can also return the maximum color distance (**M_MAX_DISTANCE**), which is the greatest color distance between the target area and all its matching color-samples.

Image results

Image results can either be drawn using **McolDraw()** with **M_DRAW_...**, or they can be produced directly using **McolMatch()** with **M_DRAW_...**. If you use **McolMatch()** to produce an image directly, you will not be able to retrieve results using **McolGetResult()**. Producing images directly with **McolMatch()** is typically done to save time when you are just interested in a specific image result. Unless otherwise specified, all images are available with either **McolDraw()** or **McolMatch()**.

Images that produce pixel results (**M_DRAW_..._PIXEL_IMAGE**) can only be specified when using an **M_MIN_DIST_VOTE** distance type.

If you use **McolDraw()** to draw the images, you must first enable the required image controls, using **McolControl()** with **M_SAVE_AREA_IMAGE** and **M_GENERATE_...**. For example, to draw **M_DRAW_COLORED_LABEL_AREA_IMAGE**, you must first enable **M_SAVE_AREA_IMAGE** and **M_GENERATE_SAMPLE_COLOR_LUT**. All controls that you must enable are specified in the drawing operation's description in **McolDraw()**.

Target areas

For each target area, you can draw:

- The best-matched color-sample.

The image can contain either the color of the best-matched color-sample (**M_DRAW_COLORED_LABEL_AREA_IMAGE**) or the label of the best-matched color-sample (**M_DRAW_LABEL_AREA_IMAGE**).

- The color-sample for which each pixel voted.

The image can contain either the color of the color-sample for which each pixel voted (**M_DRAW_COLORED_LABEL_PIXEL_IMAGE**) or the label of the color-sample for which each pixel voted (**M_DRAW_LABEL_PIXEL_IMAGE**).

- The distance image.

The distance image (**M_DRAW_DISTANCE_IMAGE**) contains the distance between the color of the target area (for an **M_STAT_MIN_DIST** operation mode) or target pixel (for an **M_MIN_DIST_VOTE** operation mode), and the color of its best-matched color-sample. This distance value is dependent on the type of distance calculated. For more information, see the *Distance between colors* section in *Chapter 17: Color processing and analysis*.

With an **M_STAT_MIN_DIST** operation mode, the distance image will always contain one distance value per target area, since each target area's average color is used. However, with **M_MIN_DIST_VOTE**, the distance image can contain different distance values per target area, since individual pixels could have voted for different color-samples.

If required, you can normalize distance results, using **McolControl()** with **M_DISTANCE_IMAGE_NORMALIZE**.

The distance image (**M_DRAW_DISTANCE_IMAGE**) and the label images (**M_...LABEL_...**) draw numerical values and are therefore grayscale images. The other images (**M_...COLORED_...**) draw colors. For information on the data of these images, use **McolGetResult()** or **McolInquire()**, as required. For example, to return the depth per band (in bits) required for the image buffer in which to draw **M_DRAW_COLORED_LABEL_AREA_IMAGE**, use **McolGetResult()** with **M_COLORED_LABEL_AREA_IMAGE_SIZE_BIT**.

Each of these images also draws background and outlier pixels. For more information, see the *Background and outliers* subsection in the *Color matching* section in *Chapter 17: Color processing and analysis*.

Color-samples

For each color-sample, you can draw:

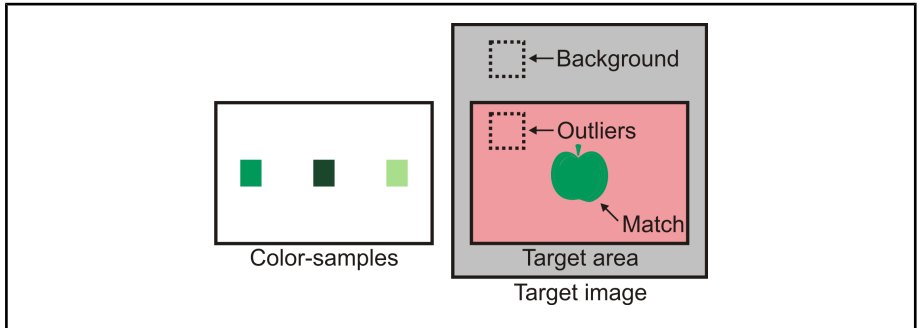
- A copy of the internal color-sample image specified with **McolDefine()** (**M_DRAW_SAMPLE_IMAGE**).
- A copy of the color-sample's mask specified with **McolMask()** (**M_DRAW_SAMPLE_MASK_IMAGE**).
- The 3-band color-sample label LUT, where the label value of each color-sample is associated to its average color (**M_DRAW_SAMPLE_COLOR_LUT**).

This image can be useful for supervised color segmentation. For example, you can draw the grayscale label image (**M_DRAW_LABEL_PIXEL_IMAGE**), and then apply the LUT (**M_DRAW_SAMPLE_COLOR_LUT**) to draw each pixel's color. Note that this produces the same result as drawing the colored label image (**M_DRAW_COLORED_LABEL_PIXEL_IMAGE**).

These images are only available with **McolDraw()**; they cannot be produced directly with **McolMatch()**.

Background and outliers

Background pixels are pixels that are outside the target areas, while outlier pixels are pixels inside a target area, but do not match with any color-sample. Note that background pixels can never match a color-sample.



When drawing image results, the destination buffer's bit depth must account for background and outlier pixels, if they are produced.

When drawing the distance image, the color of the outlier pixels might be difficult to identify. In this case, use **M_DRAW_LABEL...**; this draws the outlier labels, which you can set to a unique value and therefore distinguish them easily.

To set the outlier and background pixel value, use **McolControl()** with **M_BACKGROUND_DRAW_COLOR** and **M_OUTLIER_DRAW_COLOR**. To set the outlier label value, use **M_OUTLIER_LABEL**.

M_BACKGROUND_DRAW_COLOR is applied to the colored images (**M_DRAW_COLORED...**), the distance image (**M_DRAW_DISTANCE_IMAGE**), and to the grayscale label images (**M_DRAW_LABEL...**).

M_OUTLIER_DRAW_COLOR is applied to the colored images and the distance image. **M_OUTLIER_LABEL** is applied to the grayscale label images.

Advanced color matching settings

In addition to the fundamental color matching settings, the Color Analysis module also provides advanced settings that allow you to write highly customized color matching applications. These settings include:

- Internal conversion.
- Color band specification.
- Distance normalization.
- Color space encoding.

Internal conversion

When working in an RGB color space, you can convert your RGB data to CIELAB before the match operation, using **McolSetMethod()**. This can be useful if, for example, you have allocated an RGB color space, since it is more convenient for grabbing and displaying colors, and it is the format you typically use; however, due to your application requirements, you want to occasionally calculate with CIELAB colors.

To interpret the color space data, the color matching context uses, as its reference color space, standard RGB specifications (sRGB), which are defined by the International Electrotechnical Commission (IEC) Project Team 61966-2-1. The reference color space is specified with **McolAlloc()**.

Rather than converting your data before the match, you should consider providing images that have already been converted to CIELAB. In this case, you can allocate a CIELAB context and calculate directly in CIELAB, which might be faster.

Color band specification

By default, all bands are used when performing the match operation. You can however specify one or two specific bands, using **McolControl()** with **M_BAND_MODE**.

For example, if you are using HSL images, you can match with only the hue (h) component by setting **M_BAND_MODE** to **M_COLOR_BAND_0**. This can be useful if your image has non-uniform lighting, shadows, or highlights. A similar effect can also be produced, if you are matching in CIELAB and use only the chrominance components (bands A and B), by setting **M_BAND_MODE** to **M_COLOR_BAND_1 + M_COLOR_BAND_2**.

Distance normalization

Normalization refers to the uniform modification of distance values based on a specified multiplicative factor. It is applied when drawing the distance image (**M_DRAW_DISTANCE_IMAGE**), using **McolDraw()** or **McolMatch()**.

Normalization is typically used to improve the display and understanding of the distance image values, by remapping them to the dynamic range of the buffer in which they are drawn. To set the normalization factor, use **McolControl()** with **M_DISTANCE_IMAGE_NORMALIZE**. This can be used to:

- Normalize distances by the maximum distance calculated by the distance image (**M_MAX_NORMALIZE**).

M_MAX_NORMALIZE corresponds to the result **M_MAX_DISTANCE(McolGetResult())**.

- Normalize distances by a specific normalization factor.
- Not normalize distances (**M_NO_NORMALIZE** or **M_DEFAULT**).

After normalization, distances are saturated according to the distance buffer type. For *integer* buffers, distances are remapped according to the buffer's possible range of values. For *floating-point* buffers, the normalized values are left between 0 and 1.

Color space encoding

Color space encoding defines how color is transformed from the range represented in an image buffer to its native (theoretical) range, which is device independent. For example, RGB color space data is represented in an image buffer as values between 0 and 255 (8-bit); these values are then mapped to their native data range, which consists of all real numbers between 0 and 1.

You must set the color space encoding according to the actual dynamic range of your color data, using **McolControl()** with **M_ENCODING**. By default, the Color Analysis module assumes that you are using an 8-bit color space encoding, regardless of the depth of your buffers. If this default is not appropriate, you should modify it with **M_ENCODING** accordingly. For example, if your color data was acquired with a 16-bit camera, you should set **M_ENCODING** to **M_16BIT**.

A number of predefined color space encoding settings are provided (**M_nBIT**). These are typically sufficient for most applications. You can even use image buffers that exceed the actual dynamic range of your color data, provided that their content respects this range. For example, if you set **M_ENCODING** to **M_8BIT**, you can use 16-bit image buffers that contain values between 0 and 255 (8-bit); however, if they contain values outside this range, results will be incoherent.

If required, you can explicitly set, for each band, specific offset (**M_OFFSET_BAND_n**) and scale (**M_SCALE_BAND_n**) values that specify how color is transformed from the range represented in an image buffer to its native (theoretical) range. In this case, you must set **M_ENCODING** to **M_USER_DEFINED**.

Performing the encoding

To actually encode the data, the following native range is used for the color spaces:

RGB	Color space band	Native Min	Native max
	R	0.0	1.0
	G	0.0	1.0
	B	0.0	1.0
LAB	Color space band	Native Min	Native max
	L	0.0	100.0
	A	-128.0	127.0
	B	-128.0	127.0
HSL	Color space band	Native Min	Native max
	H	0.0	1.0 (normalized angle)
	S	0.0	1.0
	L	0.0	1.0

If you use **M_nBIT**, the color analysis module takes the color space's native range, and the range represented in the image buffer, and uses it to internally apply the following offset and scale, to perform the encoding:

Offset	Color Space	M_ENCODING	M_OFFSET_BAND_0	M_OFFSET_BAND_1	M_OFFSET_BAND_2
	RGB	M_nBIT	0	0	0
	CIELAB	M_8BIT	0	128	128
		M_nBIT	0	$128 \times (2^{n-1}) / 255$	$128 \times (2^{n-1}) / 255$
	HSL	M_nBIT	0	0	0
Scale	Color Space	M_ENCODING	M_SCALE_BAND_0	M_SCALE_BAND_1	M_SCALE_BAND_2
	RGB	M_nBIT	$1.0 / (2^{n-1})$	$1.0 / (2^{n-1})$	$1.0 / (2^{n-1})$
	CIELAB	M_nBIT	$100.0 / (2^{n-1})$	$255 / (2^{n-1})$	$255 / (2^{n-1})$
	HSL	M_nBIT	$1.0 / (2^n)$	$1.0 / (2^{n-1})$	$1.0 / (2^{n-1})$
In this table, <i>n</i> refers to the buffer's depth. Note that the H component in HSL is represented by a normalized angle, which is transformed according to the MIL angle convention. For example, in an 8-bit image buffer, 360° is mapped to 256. Since 0° and 360° are the same angle, both use the same numerical value, which is 0.					

If you use **M_USER_DEFINED**, the following encoding is applied using the specified offset (**M_OFFSET_BAND_n**) and scale (**M_SCALE_BAND_n**), for each band:

- 1. The offset (**M_OFFSET_BAND_n**) is subtracted from the color value represented by the image buffer.
- 2. The resulting color value, as modified by **M_OFFSET_BAND_n**, is then multiplied by **M_SCALE_BAND_n**.

To determine the scale and offset values you should use with **M_USER_DEFINED**, you can perform the following calculations:

Let [n1, n2] be the native range of the color space. Let [e1, e2] be the numerical range of the image buffer.	$Scale = (n2 - n1) / (e2 - e1)$ $Offset = -n2 / Scale + e2$
--	---

Color separation

You can use **McolProject()** with **M_COLOR_SEPARATION** to remove a color from an image. Eliminating a color can be very useful, since it allows you to isolate only the parts of the image that you are interested in. To perform color separation, you must identify the background, selected, and rejected colors.

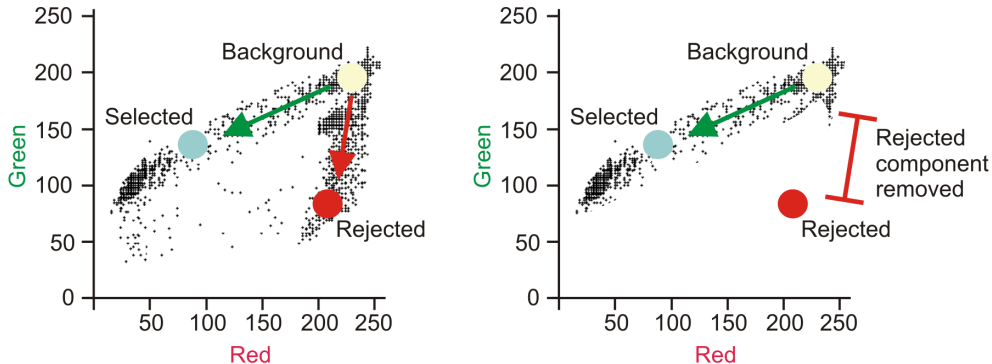


Source image

Background, selected, and rejected colors

Stamp removed

By properly specifying the colors, the projection operation is able to identify the unwanted color information (the stamp), and create a new version of the image that only contains the wanted color information (the background and the signature). The following two-dimensional graphs illustrate the color distribution of the signature/stamp image, and how that distribution is modified to remove the stamp and keep the signature.



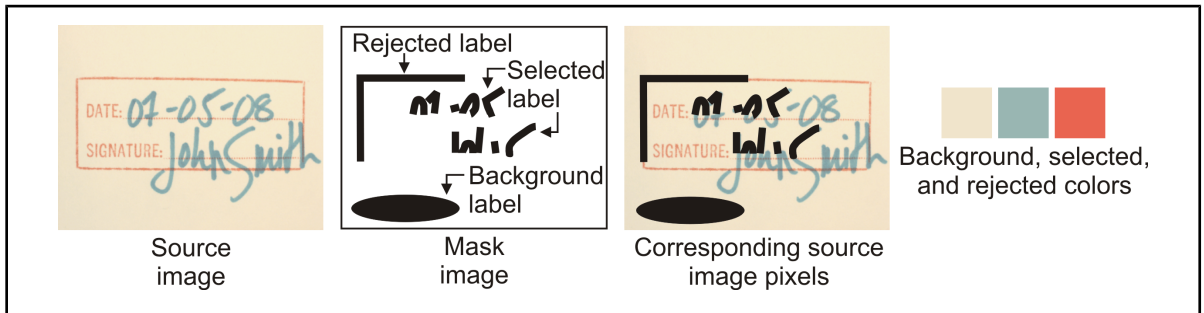
Note that these graphs represent a two-dimensional projection of three-dimensional data; this has been done to better illustrate the example.

Separation operation

The color separation operation can return either an image with the separated colors, or the transformation matrix with which to perform the color separation. This depends on whether you store the result in an image buffer or a MIL array buffer (**DestId** parameter). If you retrieve the matrix, you can then use it with **MimConvert()** to separate colors. When doing this, make sure that the color distribution of all the images that use this matrix is similar, otherwise unpredictable results can occur.

The source on which to perform the **M_COLOR_SEPARATION** operation is typically an image (**SrcId** parameter). In this case, you will either be separating the colors or calculating the matrix, depending on the **DestId** parameter. However, you can also specify **M_NULL** as the source. In this case, **McolProject()** must return the transformation matrix to perform a color separation, which is based on the provided separation colors (background, selected, and rejected).

The separation colors can be provided to **McolProject()** as either three explicit color values, or as colors taken from the source image. To specify explicit color values, you must use a 3 x 3 MIL array buffer, where each row defines, in the following order, the background, selected, and rejected colors. To specify colors taken from the source image, you must specify a type of mask (specified with the **ColorsId** parameter), which identifies the corresponding source image pixels to use as the background (**M_BACKGROUND_LABEL**), selected (**M_SELECTED_LABEL**), and rejected (**M_REJECTED_LABEL**) colors.



You must assign at least one pixel, in the mask image, to each of these labels. Pixels in the source that do not correspond to one of these labels in the mask are ignored. If the mask is smaller than the source, the outer pixels are ignored.

Color statistics

You can use **McolProject()** to perform analytical tasks, such as calculating an image's covariance matrix (**M_COVARIANCE**), or the principal components of an image's color information (**M_PRINCIPAL_COMPONENTS**).

Covariance

If you use **M_COVARIANCE**, **McolProject()** will calculate the following covariance matrix, which indicates the variation of color in the source image:

<i>var (Band0)</i>	<i>cov (Band0, Band1)</i>	<i>cov (Band0, Band2)</i>
<i>cov (Band1, Band0)</i>	<i>var (Band1)</i>	<i>cov (Band1, Band2)</i>
<i>cov (Band2, Band0)</i>	<i>cov (Band2, Band1)</i>	<i>var (Band2)</i>

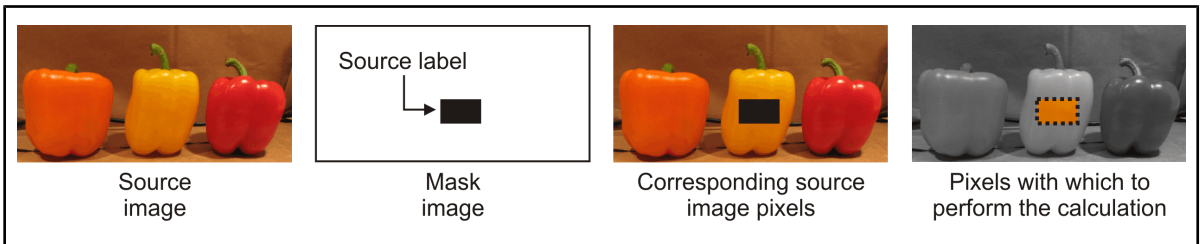
var = variance, *cov* = covariance;
covariance matrix is symmetrical, so: *CovMatrix(l, j) = CovMatrix(j, l)*

This matrix is returned if you provide a 3 x 3 MIL array as the destination buffer. You can also receive the mean color, for each band, by providing a 4 x 3 MIL array:

<i>var (Band0)</i>	<i>cov (Band0, Band1)</i>	<i>cov (Band0, Band2)</i>	<i>Band0MeanValue</i>
<i>cov (Band1, Band0)</i>	<i>var (Band1)</i>	<i>cov (Band1, Band2)</i>	<i>Band1MeanValue</i>
<i>cov (Band2, Band0)</i>	<i>cov (Band2, Band1)</i>	<i>var (Band2)</i>	<i>Band2MeanValue</i>

Source label

The covariance can either be calculated on the entire source image provided, or on a specific set of source image pixels. In this case, you must provide a type of mask image indicating the corresponding source pixels (**M_SOURCE_LABEL**) to use to compute the matrix. Identifying specific source pixels allows you to isolate only those parts of the image that contain the color you are interested in.



Principal components

If you use **M_PRINCIPAL_COMPONENTS**, **McolProject()** will calculate the three eigenvectors of the source image's color information. These values are also known as an image's principal components. In addition, you can also retrieve the image's average color values. The information that will be provided depends on the size of the MIL array that you specify as the destination buffer:

- 3 x 3.

The 3 eigenvectors are returned column-wise.

- 4 x 3.

In addition to the 3 x 3 array results, the 3 eigenvalues are also returned in the fourth column.

- 5 x 3.

In addition to the 4 x 3 array results, the 3 average color values are returned in the fifth column.

Eigenvectors and their eigenvalues are returned in decreasing order of importance.

The principal components can either be calculated on the entire source image provided, or on a specific set of source image pixels (**M_SOURCE_LABEL**). For more information on specifying a specific set of source pixels, see the *Source label* subsection in this section.

Color processing and analysis example

For an example on how to use the MIL Color Analysis module, see *mcol.cpp*.

```

/*****
/*
* File name: MCol.cpp
*
* Synopsis: This program contains 3 examples of the color module usage:
*
*           The first example performs color segmentation of an image
*           by classifying each pixel with one out of 7 color samples.
*           The ratio of each color in the image is then calculated.
*
*           The second example performs color matching of circular regions
*           in objects located with model finder.
*
*           The third example performs color separation in order to
*           separate 2 types of ink on a piece of paper.
*/
#include <mil.h>

/* Display image margin */
#define DISPLAY_CENTER_MARGIN_X 5

/* Color patch sizes */
#define COLOR_PATCH_SIZEX 30
#define COLOR_PATCH_SIZEY 40

/* Example functions declarations. */
void ColorSegmentationExample(MIL_ID MilSystem, MIL_ID MilDisplay);
void ColorMatchingExample(MIL_ID MilSystem, MIL_ID MilDisplay);
void ColorSeparationExample(MIL_ID MilSystem, MIL_ID MilDisplay);

/* Utility function */
void DrawSampleColors(MIL_ID DestImage,
                     const MIL_INT pSamplesColors[][3],
                     MIL_CONST_TEXT_PTR *pSampleNames,
                     MIL_INT NumSamples,
                     MIL_INT XSpacing,

```

```

        MIL_INT YOffset);

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

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

    /* Run the color segmentation example. */
    ColorSegmentationExample(MilSystem, MilDisplay);

    /* Run the color matching example. */
    ColorMatchingExample(MilSystem, MilDisplay);

    /* Run the color projection example. */
    ColorSeparationExample(MilSystem, MilDisplay);

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

    return 0;
}

/*****
Color Segmentation using color samples.
*****/

/* Image filenames */
#define CANDY_SAMPLE_IMAGE_FILE    M_IMAGE_PATH MIL_TEXT("CandySamples.mim")
#define CANDY_TARGET_IMAGE_FILE    M_IMAGE_PATH MIL_TEXT("Candy.mim")

/* Number of samples */
#define NUM_SAMPLES                6

/* Draw spacing and offset */
#define CANDY_SAMPLES_XSPACING     35
#define CANDY_SAMPLES_YOFFSET     145

/* Match parameters */
#define MATCH_MODE                 M_MIN_DIST_VOTE /* Minimal distance vote mode. */
#define DISTANCE_TYPE              M_MAHALANOBIS  /* Mahalanobis distance.      */
#define TOLERANCE_MODE             M_SAMPLE_STDDEV /* Standard deviation tolerance mode. */
#define TOLERANCE_VALUE           6.0             /* Mahalanobis tolerance value. */
#define RED_TOLERANCE_VALUE       6.0
#define YELLOW_TOLERANCE_VALUE    12.0
#define PINK_TOLERANCE_VALUE      5.0

void ColorSegmentationExample(MIL_ID MilSystem, MIL_ID MilDisplay)

```

```

{
MIL_ID      SourceChild      = M_NULL,    /* Source image buffer identifier. */
            DestChild        = M_NULL,    /* Dest image buffer identifier. */
            MatchContext     = M_NULL,    /* Color matching context identifier. */
            MatchResult      = M_NULL,    /* Color matching result identifier. */
            DisplayImage     = M_NULL;    /* Display image buffer identifier. */

MIL_INT      SourceSizeX, SourceSizeY,      /* Source image sizes */
            SampleIndex, SpacesIndex;      /* Indices */

MIL_DOUBLE MatchScore;                      /* Color matching score. */

/* Blank spaces to align the samples names evenly. */
MIL_CONST_TEXT_PTR Spaces[4] = {MT(""), MT(" "), MT("  "), MT("   ")};

/* Color samples names. */
MIL_CONST_TEXT_PTR SampleNames[NUM_SAMPLES] = {MT("Green"),
                                                MT("Red"),
                                                MT("Yellow"),
                                                MT("Purple"),
                                                MT("Blue"),
                                                MT("Pink")};

/* Color samples position: {OffsetX, OffsetY} */
const MIL_DOUBLE SamplesROI[NUM_SAMPLES][2] = {{ 58, 143},
                                                {136, 148},
                                                {217, 144},
                                                {295, 142},
                                                {367, 143},
                                                {442, 147}};

/* Color samples size. */
const MIL_DOUBLE SampleSizeX = 36, SampleSizeY = 32;

/* Array for match sample colors. */
MIL_INT SampleMatchColor[NUM_SAMPLES][3];

MosPrintf(MIL_TEXT("\nCOLOR SEGMENTATION:\n"));
MosPrintf( MIL_TEXT("-----\n"));

/* Allocate the parent display image. */
MbufDiskInquire(CANDY_SAMPLE_IMAGE_FILE, M_SIZE_X, &SourceSizeX);
MbufDiskInquire(CANDY_SAMPLE_IMAGE_FILE, M_SIZE_Y, &SourceSizeY);
MbufAllocColor(MilSystem, 3, 2*SourceSizeX + DISPLAY_CENTER_MARGIN_X, SourceSizeY,
               8+M_UNSIGNED, M_IMAGE+M_DISP+M_PROC, &DisplayImage);
MbufClear(DisplayImage, M_COLOR_BLACK);

/* Create a source and dest child in the display image. */
MbufChild2d(DisplayImage, 0, 0, SourceSizeX, SourceSizeY, &SourceChild);
MbufChild2d(DisplayImage, SourceSizeX + DISPLAY_CENTER_MARGIN_X, 0, \
            SourceSizeX, SourceSizeY, &DestChild);

/* Load the source image into the source child. */

```



```

MbufLoad(CANDY_SAMPLE_IMAGE_FILE, SourceChild);

/* Allocate a color matching context. */
McolAlloc(MilSystem, M_COLOR_MATCHING, M_RGB, M_DEFAULT, M_DEFAULT, &MatchContext);

/* Define each color sample in the context. */
for(MIL_INT i=0; i<NUM_SAMPLES; i++)
{
    McolDefine(MatchContext, SourceChild, M_SAMPLE_LABEL(i+1), M_IMAGE,
               SamplesROI[i][0], SamplesROI[i][1], SampleSizeX, SampleSizeY);
}

/* Set the color matching parameters. */
McolSetMethod(MatchContext, MATCH_MODE, DISTANCE_TYPE, M_DEFAULT, M_DEFAULT);
McolControl(MatchContext, M_CONTEXT, M_DISTANCE_TOLERANCE_MODE, TOLERANCE_MODE);
McolControl(MatchContext, M_ALL, M_DISTANCE_TOLERANCE, TOLERANCE_VALUE);

/* Adjust tolerances for the red, yellow and pink samples. */
McolControl(MatchContext, M_SAMPLE_INDEX(1), M_DISTANCE_TOLERANCE,
             RED_TOLERANCE_VALUE);
McolControl(MatchContext, M_SAMPLE_INDEX(2), M_DISTANCE_TOLERANCE,
             YELLOW_TOLERANCE_VALUE);
McolControl(MatchContext, M_SAMPLE_INDEX(5), M_DISTANCE_TOLERANCE,
             PINK_TOLERANCE_VALUE);

/* Preprocess the context. */
McolPreprocess(MatchContext, M_DEFAULT);

/* Fill the samples colors array. */
for(MIL_INT i=0; i<NUM_SAMPLES; i++)
{
    McolInquire(MatchContext, M_SAMPLE_LABEL(i+1),
                M_MATCH_SAMPLE_COLOR_BAND_0 + M_TYPE_MIL_INT, &SampleMatchColor[i][0]);
    McolInquire(MatchContext, M_SAMPLE_LABEL(i+1),
                M_MATCH_SAMPLE_COLOR_BAND_1 + M_TYPE_MIL_INT, &SampleMatchColor[i][1]);
    McolInquire(MatchContext, M_SAMPLE_LABEL(i+1),
                M_MATCH_SAMPLE_COLOR_BAND_2 + M_TYPE_MIL_INT, &SampleMatchColor[i][2]);
}

/* Draw the samples. */
DrawSampleColors(DestChild, SampleMatchColor, SampleNames, NUM_SAMPLES,
                 CANDY_SAMPLES_XSPACING, CANDY_SAMPLES_YOFFSET);

/* Select the image buffer for display. */
MdispSelect(MilDisplay, DisplayImage);

/* Pause to show the original image. */
MosPrintf(MIL_TEXT("Color samples are defined for each possible candy color.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to do color matching.\n\n"));
MosGetch();

/* Load the target image.*/
MbufClear(DisplayImage, M_COLOR_BLACK);

```

```

MbufLoad(CANDY_TARGET_IMAGE_FILE, SourceChild);

/* Allocate a color matching result buffer. */
McolAllocResult(MilSystem, M_COLOR_MATCHING_RESULT, M_DEFAULT, &MatchResult);

/* Enable controls to draw the labeled color image. */
McolControl(MatchContext, M_CONTEXT, M_GENERATE_LABEL_PIXEL_IMAGE, M_ENABLE);
McolControl(MatchContext, M_CONTEXT, M_GENERATE_SAMPLE_COLOR_LUT, M_ENABLE);

/* Match with target image. */
McolMatch(MatchContext, SourceChild, M_DEFAULT, M_NULL, MatchResult, M_DEFAULT);

/* Retrieve and display results. */
MosPrintf(MIL_TEXT("Each pixel of the mixture is matched ")
           MIL_TEXT("with one of the color samples.\n"));
MosPrintf(MIL_TEXT("\nColor segmentation results:\n"));
MosPrintf(MIL_TEXT("-----\n"));

for(SampleIndex=0; SampleIndex<NUM_SAMPLES; SampleIndex++)
{
    McolGetResult(MatchResult, M_DEFAULT, M_SAMPLE_INDEX(SampleIndex),
                 M_SCORE, &MatchScore);
    SpacesIndex = 6 - MosStrlen(SampleNames[SampleIndex]);
    MosPrintf(MIL_TEXT("Ratio of %s%s sample = %5.2f%%\n"), SampleNames[SampleIndex],
              Spaces[SpacesIndex], MatchScore);
}
MosPrintf(MIL_TEXT("\nResults reveal the low proportion of Blue candy.\n"));

/* Draw the colored label image in the destination child. */
McolDraw(M_DEFAULT, MatchResult, DestChild, M_DRAW_COLORED_LABEL_PIXEL_IMAGE,
          M_ALL, M_ALL, M_DEFAULT);

/* Pause to show the result image. */
MosPrintf(MIL_TEXT("\nPress <Enter> to end.\n\n"));
MosGetch();

/* Free all allocations. */
MbufFree(DestChild);
MbufFree(SourceChild);
MbufFree(DisplayImage);
McolFree(MatchResult);
McolFree(MatchContext);
}

/*****
Color matching in labeled regions.
*****/
/* Image filenames */
#define FUSE_SAMPLES_IMAGE      M_IMAGE_PATH MT("FuseSamples.mim")
#define FUSE_TARGET_IMAGE      M_IMAGE_PATH MT("Fuse.mim")

/* Model Finder context filename */
#define FINDER_CONTEXT          M_IMAGE_PATH MT("FuseModel.mmf")

```

```

/* Number of fuse sample objects */
#define NUM_FUSES          4

/* Draw spacing and offset */
#define FUSE_SAMPLES_XSPACING 40
#define FUSE_SAMPLES_YOFFSET 145

void ColorMatchingExample(MIL_ID MilSystem, MIL_ID MilDisplay)
{
    MIL_ID    DisplayImage      = M_NULL, /* Display image buffer identifier. */
    SourceChild = M_NULL, /* Source image buffer identifier. */
    DestChild   = M_NULL, /* Dest image buffer identifier. */
    ColMatchContext = M_NULL, /* Color matching context identifier. */
    ColMatchResult = M_NULL, /* Color matching result identifier. */
    ModelImage   = M_NULL, /* Model image buffer identifier. */
    AreaImage    = M_NULL, /* Area image buffer identifier. */
    OverlayID    = M_NULL, /* Overlay image buffer identifier. */
    OverlaySourceChild = M_NULL, /* Overlay source child identifier. */
    OverlayDestChild = M_NULL, /* Overlay dest child identifier. */
    FuseFinderCtx = M_NULL, /* Model finder context identifier. */
    FuseFinderRes = M_NULL; /* Model finder result identifier. */

    /* Image sizes */
    MIL_INT SizeX, SizeY;

    /* Color sample names */
    MIL_CONST_TEXT_PTR SampleNames[NUM_FUSES] =
        {MT("Green"), MT(" Blue"), MT(" Red"), MT("Yellow")};

    /* Sample ROIs coordinates: OffsetX, OffsetY, SizeX, SizeY */
    const MIL_INT SampleROIs[NUM_FUSES][4] = {{ 54, 139, 28, 14},
        {172, 137, 30, 23},
        {296, 135, 31, 23},
        {417, 134, 27, 22}};

    /* Array of match sample colors. */
    MIL_INT SampleMatchColor[NUM_FUSES][3];

    MosPrintf(MIL_TEXT("\nCOLOR IDENTIFICATION:\n"));
    MosPrintf( MIL_TEXT("-----\n"));

    /* Allocate the parent display image. */
    MbufDiskInquire(FUSE_TARGET_IMAGE, M_SIZE_X, &SizeX);
    MbufDiskInquire(FUSE_TARGET_IMAGE, M_SIZE_Y, &SizeY);
    MbufAllocColor(MilSystem,
        3,
        2*SizeX + DISPLAY_CENTER_MARGIN_X,
        SizeY, 8+M_UNSIGNED,
        M_IMAGE+M_DISP+M_PROC,
        &DisplayImage);
    MbufClear(DisplayImage, M_COLOR_BLACK);

```

```

/* Allocate the model, area and label images. */
MbufAlloc2d(MilSystem, SizeX, SizeY, 8+M_UNSIGNED, M_IMAGE+M_PROC+M_DISP, &ModelImage);
MbufAlloc2d(MilSystem, SizeX, SizeY, 8+M_UNSIGNED, M_IMAGE+M_PROC+M_DISP, &AreaImage);

/* Create a source and destination child in the display image. */
MbufChild2d(DisplayImage, 0, 0, SizeX, SizeY, &SourceChild);
MbufChild2d(DisplayImage, SizeX + DISPLAY_CENTER_MARGIN_X, 0, SizeX, SizeY, &DestChild);

/* Load the sample source image. */
MbufLoad(FUSE_SAMPLES_IMAGE, SourceChild);

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

/* Prepare the overlay. */
MdispControl(MilDisplay, M_OVERLAY, M_ENABLE);
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);
MdispInquire(MilDisplay, M_OVERLAY_ID, &OverlayID);
MbufChild2d(OverlayID, 0, 0, SizeX, SizeY, &OverlaySourceChild);
MbufChild2d(OverlayID, SizeX + DISPLAY_CENTER_MARGIN_X, 0,
            SizeX, SizeY, &OverlayDestChild);

/* Prepare the model finder context and result. */
MmodRestore(FINDER_CONTEXT, MilSystem, M_DEFAULT, &FuseFinderCtx);
MmodPreprocess(FuseFinderCtx, M_DEFAULT);
MmodAllocResult(MilSystem, M_DEFAULT, &FuseFinderRes);

/* Allocate a color match context and result. */
McolAlloc(MilSystem, M_COLOR_MATCHING, M_RGB, M_DEFAULT, M_DEFAULT, &ColMatchContext);
McolAllocResult(MilSystem, M_COLOR_MATCHING_RESULT, M_DEFAULT, &ColMatchResult);

/* Define the color samples in the context. */
for(MIL_INT i=0; i<NUM_FUSES; i++)
{
    McolDefine(ColMatchContext, SourceChild, M_SAMPLE_LABEL(i+1), M_IMAGE,
               (MIL_DOUBLE)SampleROIs[i][0],
               (MIL_DOUBLE)SampleROIs[i][1],
               (MIL_DOUBLE)SampleROIs[i][2],
               (MIL_DOUBLE)SampleROIs[i][3]);
}

/* Preprocess the context. */
McolPreprocess(ColMatchContext, M_DEFAULT);

/* Fill the samples colors array. */
for(MIL_INT i=0; i<NUM_FUSES; i++)
{
    McolInquire(ColMatchContext, M_SAMPLE_LABEL(i+1),
                M_MATCH_SAMPLE_COLOR_BAND_0 + M_TYPE_MIL_INT, &SampleMatchColor[i][0]);

    McolInquire(ColMatchContext, M_SAMPLE_LABEL(i+1),
                M_MATCH_SAMPLE_COLOR_BAND_1 + M_TYPE_MIL_INT, &SampleMatchColor[i][1]);
}

```

```

    McolInquire(ColMatchContext, M_SAMPLE_LABEL(i+1),
                M_MATCH_SAMPLE_COLOR_BAND_2 + M_TYPE_MIL_INT, &SampleMatchColor[i][2]);

}

/* Draw the color samples. */
DrawSampleColors(DestChild, SampleMatchColor, SampleNames,
                 NUM_FUSES, FUSE_SAMPLES_XSPACING, FUSE_SAMPLES_YOFFSET);

/* Draw the sample ROIs in the source image overlay. */
MgraColor(M_DEFAULT, M_COLOR_RED);
for(MIL_INT SampleIndex = 0; SampleIndex < NUM_FUSES; SampleIndex++)
{
    MIL_INT XEnd = SampleROIs[SampleIndex][0] + SampleROIs[SampleIndex][2] - 1;
    MIL_INT YEnd = SampleROIs[SampleIndex][1] + SampleROIs[SampleIndex][3] - 1;
    MgraRect(M_DEFAULT, OverlaySourceChild, SampleROIs[SampleIndex][0],
              SampleROIs[SampleIndex][1],
              XEnd, YEnd);
}

/* Pause to show the source image. */
MosPrintf(MT("Colors are defined using one color sample region per fuse.\n"));
MosPrintf(MT("Press <Enter> to process the target image.\n"));
MosGetch();

/* Clear the overlay. */
MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

/* Load the target image into the source child. */
MbufLoad(FUSE_TARGET_IMAGE, SourceChild);

/* Get the grayscale model image and copy it into the display dest child. */
MimConvert(SourceChild, ModelImage, M_RGB_TO_L);
MbufCopy(ModelImage, DestChild);

/* Find the Model. */
MmodFind(FuseFinderCtx, ModelImage, FuseFinderRes);

/* Draw the blob image: labeled circular areas centered at each found fuse occurrence. */
MIL_INT Number;
MmodGetResult(FuseFinderRes, M_DEFAULT, M_NUMBER+M_TYPE_MIL_INT, &Number);
MbufClear(AreaImage, 0);
for(MIL_INT ii=0; ii<Number; ii++)
{
    MIL_DOUBLE X, Y;
    /* Get the position */
    MmodGetResult(FuseFinderRes, ii, M_POSITION_X, &X);
    MmodGetResult(FuseFinderRes, ii, M_POSITION_Y, &Y);
    /* Set the label color */
    MgraColor(M_DEFAULT, (MIL_DOUBLE) ii+1);
    /* Draw the filled circle */
    MgraArcFill(M_DEFAULT, AreaImage, X, Y, 20, 20, 0, 360);
}

```

```

    }

    /* Enable controls to draw the labeled color image. */
    McolControl(ColMatchContext, M_CONTEXT, M_SAVE_AREA_IMAGE, M_ENABLE);
    McolControl(ColMatchContext, M_CONTEXT, M_GENERATE_SAMPLE_COLOR_LUT, M_ENABLE);

    /* Perform the color matching. */
    McolMatch(ColMatchContext, SourceChild, M_DEFAULT, AreaImage, ColMatchResult, M_DEFAULT);

    /* Draw the label image into the overlay child. */
    McolDraw(M_DEFAULT, ColMatchResult, OverlayDestChild,
             M_DRAW_COLORED_LABEL_AREA_IMAGE, M_ALL, M_ALL, M_DEFAULT);

    /* Draw the model position over the colored areas. */
    MgraColor(M_DEFAULT, M_COLOR_BLUE);
    MmodDraw(M_DEFAULT, FuseFinderRes, OverlayDestChild, M_DRAW_BOX+M_DRAW_POSITION,
             M_ALL, M_DEFAULT);

    /* Enable the display update. */
    MdispControl(MilDisplay, M_UPDATE, M_ENABLE);

    /* Pause to show the resulting image. */
    MosPrintf(MT("\nFuses are located using the Model Finder tool.\n"));
    MosPrintf(MT("The color of each target area is identified.\n"));
    MosPrintf(MT("Press <Enter> to end.\n"));
    MosGetch();

    /* Free all allocations. */
    MmodFree(FuseFinderRes);
    MmodFree(FuseFinderCtx);
    MbufFree(AreaImage);
    MbufFree(ModelImage);
    MbufFree(SourceChild);
    MbufFree(DestChild);
    MbufFree(OverlaySourceChild);
    MbufFree(OverlayDestChild);
    MbufFree(DisplayImage);
    McolFree(ColMatchContext);
    McolFree(ColMatchResult);
}

/*****
Perform color separation of colored inks on a piece of paper.
*****/
/* Source image */
#define WRITING_IMAGE_FILE    M_IMAGE_PATH MIL_TEXT("stamp.mim")

/* Color triplets */
#define BACKGROUND_COLOR    {245, 234, 206}
#define WRITING_COLOR        {141, 174, 174}
#define STAMP_COLOR          {226, 150, 118}

/* Drawing spacing */

```

```

#define PATCHES_XSPACING    70

void ColorSeparationExample(MIL_ID MilSystem, MIL_ID MilDisplay)
{
    MIL_ID DisplayImage      = M_NULL,          /* Display image buffer identifier. */
    SourceChild              = M_NULL,          /* Source image buffer identifier. */
    DestChild                = M_NULL,          /* Destination image buffer identifier. */
    Child                    = M_NULL,          /* Child buffer identifier. */
    ColorsArray               = M_NULL;         /* Array buffer identifier. */

    /* Source image sizes. */
    MIL_INT SourceSizeX, SourceSizeY;

    /* Color samples' names */
    MIL_CONST_TEXT_PTR ColorNames[3] = {MT("BACKGROUND"), MT("WRITING"), MT("STAMP")};

    /* Array with color patches to draw. */
    MIL_INT Colors[3][3] = {BACKGROUND_COLOR, WRITING_COLOR, STAMP_COLOR};

    /* Samples' color coordinates */
    MIL_UINT8 BackgroundColor[3] = BACKGROUND_COLOR;
    MIL_UINT8 SelectedColor[3]   = WRITING_COLOR;
    MIL_UINT8 RejectedColor[3]   = STAMP_COLOR;

    MosPrintf(MIL_TEXT("\nCOLOR SEPARATION:\n"));
    MosPrintf( MIL_TEXT("-----\n"));

    /* Allocate an array buffer and fill it with the color coordinates. */
    MbufAlloc2d(MilSystem, 3, 3, 8+M_UNSIGNED, M_ARRAY, &ColorsArray);
    MbufPut2d(ColorsArray, 0, 0, 3, 1, BackgroundColor);
    MbufPut2d(ColorsArray, 0, 1, 3, 1, SelectedColor);
    MbufPut2d(ColorsArray, 0, 2, 3, 1, RejectedColor);

    /* Allocate the parent display image. */
    MbufDiskInquire(WRITING_IMAGE_FILE, M_SIZE_X, &SourceSizeX);
    MbufDiskInquire(WRITING_IMAGE_FILE, M_SIZE_Y, &SourceSizeY);
    MbufAllocColor(MilSystem,
        3,
        2*SourceSizeX + DISPLAY_CENTER_MARGIN_X,
        SourceSizeY,
        8+M_UNSIGNED,
        M_IMAGE+M_DISP,
        &DisplayImage);
    MbufClear(DisplayImage, M_COLOR_BLACK);

    /* Clear the overlay. */
    MdispControl(MilDisplay, M_OVERLAY_CLEAR, M_DEFAULT);

    /* Create a source and dest child in the display image */
    MbufChild2d(DisplayImage, 0, 0, SourceSizeX, SourceSizeY, &SourceChild);
    MbufChild2d(DisplayImage, SourceSizeX + DISPLAY_CENTER_MARGIN_X,
        0, SourceSizeX, SourceSizeY, &DestChild);

```

```

/* Load the source image into the display image source child. */
MbufLoad(WRITING_IMAGE_FILE, SourceChild);

/* Draw the color patches. */
DrawSampleColors(DestChild, Colors, ColorNames, 3, PATCHES_XSPACING, -1);

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

/* Pause to show the source image and color patches. */
MosPrintf(MIL_TEXT("The writing will be separated from the ")
          MIL_TEXT("stamp using the following triplets:\n"));
MosPrintf(MIL_TEXT("the background color: beige [%d, %d, %d],\n"),
          BackgroundColor[0], BackgroundColor[1], BackgroundColor[2]);
MosPrintf(MIL_TEXT("the writing color : green [%d, %d, %d],\n"),
          SelectedColor[0], SelectedColor[1], SelectedColor[2]);
MosPrintf(MIL_TEXT("the stamp color : red [%d, %d, %d].\n\n"),
          RejectedColor[0], RejectedColor[1], RejectedColor[2]);
MosPrintf(MIL_TEXT("Press <Enter> to extract the writing.\n\n"));
MosGetch();

/* Perform the color projection. */
McolProject(SourceChild, ColorsArray, DestChild, M_NULL,
            M_COLOR_SEPARATION, M_DEFAULT, M_NULL);

/* Wait for a key. */
MosPrintf(MIL_TEXT("Press <Enter> to extract the stamp.\n\n"));
MosGetch();

/* Switch the order of the selected vs rejected colors in the color array. */
MbufPut2d(ColorsArray, 0, 2, 3, 1, SelectedColor);
MbufPut2d(ColorsArray, 0, 1, 3, 1, RejectedColor);

/* Perform the color projection. */
McolProject(SourceChild, ColorsArray, DestChild, M_NULL,
            M_COLOR_SEPARATION, M_DEFAULT, M_NULL);

/* Wait for a key. */
MosPrintf(MIL_TEXT("Press <Enter> to end.\n\n"));
MosGetch();

/* Free all allocations. */
MbufFree(ColorsArray);
MbufFree(SourceChild);
MbufFree(DestChild);
MbufFree(DisplayImage);
};
/*****
/* Draw the samples as color patches. */
void DrawSampleColors(MIL_ID DestImage,
                     const MIL_INT pSamplesColors[][3],
                     MIL_CONST_TEXT_PTR *pSampleNames,
                     MIL_INT NumSamples,

```



```

        MIL_INT XSpacing,
        MIL_INT YOffset)
{
    MIL_INT DestSizeX = MbufInquire(DestImage, M_SIZE_X, M_NULL);
    MIL_INT DestSizeY = MbufInquire(DestImage, M_SIZE_Y, M_NULL);
    MIL_DOUBLE OffsetX = (DestSizeX - (NumSamples * COLOR_PATCH_SIZEX) -
        ((NumSamples - 1) * XSpacing)) / 2.0;
    MIL_DOUBLE OffsetY = YOffset > 0 ? YOffset : (DestSizeY - COLOR_PATCH_SIZEY) / 2.0;
    MIL_DOUBLE TextOffsetX;
    MgraFont(M_DEFAULT, M_FONT_DEFAULT_SMALL);

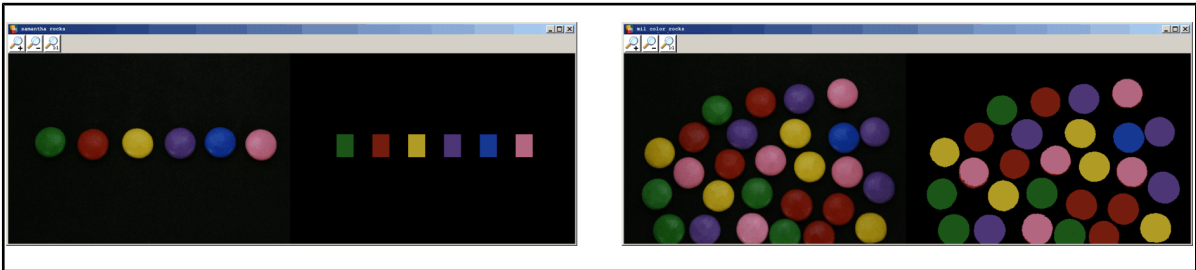
    for(MIL_INT SampleIndex = 0; SampleIndex < NumSamples; SampleIndex++)
    {
        MgraColor(M_DEFAULT, M_RGB888(pSamplesColors[SampleIndex][0],
            pSamplesColors[SampleIndex][1], pSamplesColors[SampleIndex][2]));
        MgraRectFill(M_DEFAULT, DestImage, OffsetX, OffsetY, OffsetX + COLOR_PATCH_SIZEX,
            OffsetY + COLOR_PATCH_SIZEY);

        MgraColor(M_DEFAULT, M_COLOR_YELLOW);
        TextOffsetX = OffsetX + COLOR_PATCH_SIZEX / 2.0 - 4.0 *
            MosStrlen(pSampleNames[SampleIndex]) + 0.5;
        MgraText(M_DEFAULT, DestImage, TextOffsetX, OffsetY-20, pSampleNames[SampleIndex]);
        OffsetX += (COLOR_PATCH_SIZEX + XSpacing);
    }
}

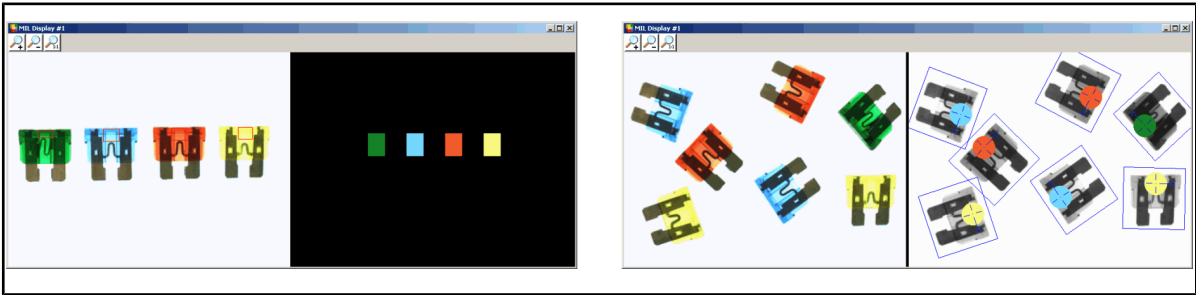
```

This example shows you how to perform:

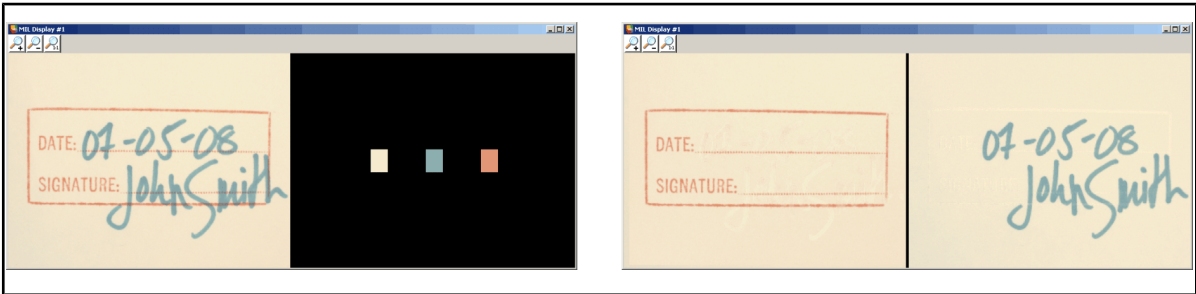
1. Color segmentation of an image by classifying each pixel with 1 out of 6 color-samples. The ratio of each color in the image is then calculated.



2. Color identification of circular regions in objects located with the MIL Model Finder module.



3. Color separation of a signature and a stamp.



Chapter

18

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

In this help file, 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

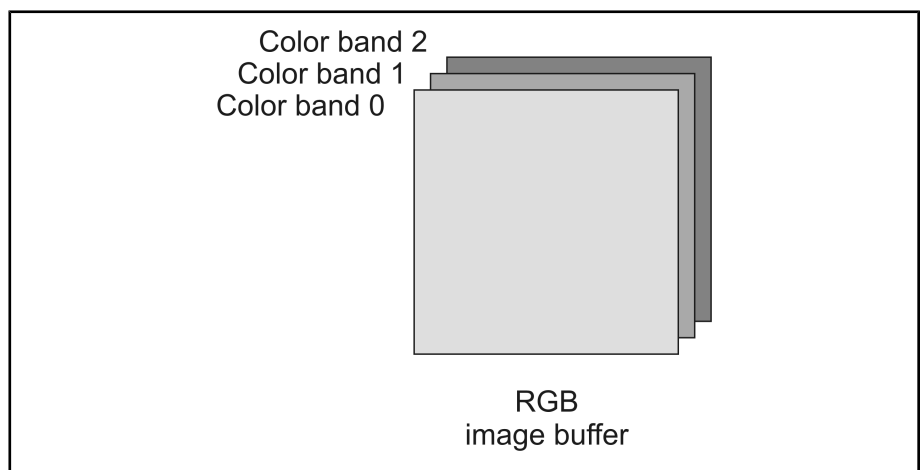
A data buffer is allocated on the specified system. If the **M_DEFAULT_HOST** system is specified, the default Host system of the current MIL application will be used.

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 **MbuffFree()**.



Certain MIL functions support manipulating multi-band image buffers. For details on handling color image buffers, see the *Dealing with color* section in *Chapter 2: Building an application*.

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 function's description, otherwise the function can be used with any combination of data buffers.

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 Reference.

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

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 buffers are used to store grabbed images.
- **M_LUT**. Lookup table buffers are used to store lookup table data.
- **M_KERNEL**. Kernel buffers define the filters used by convolution functions.
- **M_STRUCT_ELEMENT**. Structuring element buffers are used to store the structuring element data for morphology functions.
- **M_ARRAY**. Array buffers store array type data.

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+M_GRAB.

- ❖ The **M_KERNEL** and **M_STRUCT_ELEMENT** attributes are not required to work under MIL-Lite.

Displayable buffers	<p>When a displayable buffer is allocated and selected for display (MbufAlloc... with M_DISP, and then MdispSelect()), multiple buffers are maintained: one in Host memory for processing purposes, and other internal buffers in display or non-paged memory (maintained directly or using the normal Windows mechanisms) for display purposes (not necessarily the same size). When the Host buffer is modified, its associated internal buffers in display/non-paged memory are automatically updated. When displaying a buffer, both the buffer and the display should be allocated on the same system.</p>
Grab buffers	<p>Buffers with the M_GRAB attribute are allocated in MIL non-paged (DMA) memory, which is physically contiguous. An advantage of non-paged memory is that a bus mastering device can write to it without the help of the Host CPU. If a system does not support grab buffers (for example, M_SYSTEM_HOST), you could still allocate a buffer on such a system in physically contiguous and non-paged by giving it an M_NON_PAGED attribute instead.</p> <p>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 will typically bypass the specified buffer, and grab into an intermediate temporary display buffer (in display or Host non-paged memory) to minimize CPU usage and improve performance. Only the last frame grabbed is written into the selected buffer.</p>
Storage format specifiers	<p>It is 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.</p>
Inappropriate data buffer usage	<p>If you try to use a data buffer in a situation that is not appropriate for its allocated attributes, 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.</p>

Memory locations

When allocating a buffer, the default memory storage locations are determined based on whether your Matrox Imaging board has an on-board processor (CPU, PA, or JPEG2000 accelerator). In most cases using the default memory storage locations will produce the best results. The following memory storage locations are available:

- **M_HOST_MEMORY.** Stores the buffer in Host memory.
- **M_ON_BOARD.** Stores the buffer in memory located on your Matrox Imaging board.
- **M_OFF_BOARD.** Stores the buffer in memory located off your Matrox Imaging board.
- **M_MAPPABLE.** Stores the buffer in non-paged memory that is not mapped to Host memory. The Host memory mapping can be controlled using **MbufControl()** with **M_MAP**.
- **M_VIDEO_MEMORY.** Stores the buffer in display memory.

Settings for Matrox Imaging boards with an integrated CPU

For boards with an integrated CPU, specifying **M_OFF_BOARD** assumes that your MIL application has access to a source of memory located off the main board (for example, a Matrox Imaging board installed in a computer).

Using on-board memory can improve performance when all buffers are allocated and processed on-board. This is beneficial for Matrox Imaging boards (with either an integrated CPU or a PA) that can accelerate the processing functions (for example, Matrox Odyssey). This is also beneficial when using a GPU of your display board for processing (GPU system).

There are three types of on-board memory available for boards with an integrated PA:

- **M_FAST_MEMORY.** Available only on a limited number of boards. In the case of the Matrox Odyssey, **M_FAST_MEMORY** forces the buffer in the G4 PowerPC fast-processing memory. This memory is L3 external cache.
- **M_SHARED.** Available in all boards with a PA, this forces the buffer in memory that is mapped on the PCI bus. Note that shared memory is only available on-board.
- **Unshared memory.** This is the default memory location for all boards with a PA. The buffer is stored in unmapped memory.

Settings for Matrox Imaging boards without an on-board CPU

By default, all Matrox Imaging boards without an on-board CPU allocate buffers in shared Host memory instead of on-board memory. On these boards, on-board memory is typically limited in size and Host memory can be accessed much faster than on-board memory. Changing the default location for your buffer's allocation is typically done only when dealing with large buffers or limited amounts of memory.

For all Matrox Imaging boards without an on-board processor **M_HOST_MEMORY** is the same as **M_OFF_BOARD**.

Settings for all Matrox Imaging boards

The transfer from an off-board memory location to the on-board processor can be CPU intensive. For more information, refer to the Board Specific Notes for your Matrox Imaging board.

When dealing with many very large buffers that cannot all be mapped in Host memory at the same time, allocate the buffers as **M_MAPPABLE** so that they can be mapped and unmapped as needed. A mappable buffer is allocated in non-paged Host memory. By default the buffer will have a physical address, but not a Host address. Use **MbufControl()** with **M_MAP** to enable or disable the mapping.

There are two types of memory locations available to all Matrox Imaging boards:

- **M_PAGED.** Forces the buffer into paged memory. Note that this value cannot be used with **M_MAPPABLE**.
- **M_NON_PAGED.** Forces the buffer into non-paged memory.

Paging memory

Paging memory at allocation divides the available computer memory into small portions where the page is the smallest building block. Paging memory helps reduce the amount of external fragmentation and thus little memory is wasted. Use MilConfig to set the amount of Non-paged memory available to your system. The rest of the memory is available for paging memory. Whenever possible, use the default settings for paged and non-paged memory.

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.

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.

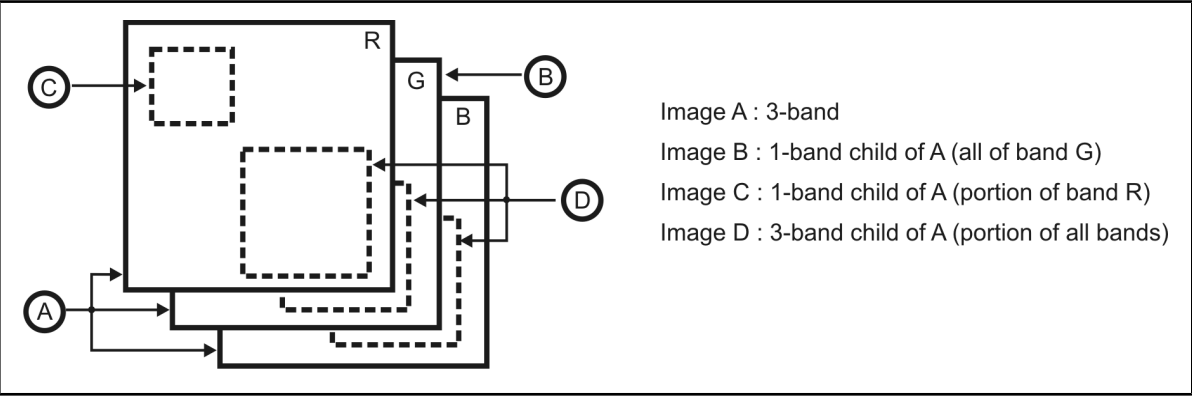
Child buffers

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 memory space as the parent buffer, changes made to the child buffer affect the parent buffer and vice versa.

A child buffer is considered a data buffer in its own right, and can be used in the same context as its parent buffer. A child buffer takes on the same attributes and type as the parent buffer. In addition, any coordinates specified or returned when using a child buffer are relative to the child buffer's top-left corner.

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 and 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.

When the parent buffer is multi-band, the child buffer can occupy one or all bands. The following example shows some child buffers that can be created from a multi-band image parent buffer.



Just as its parent buffer, a child buffer must be allocated so that it can be associated with an identifier and recognized as an entity by MIL. You can use one of the MIL functions from the table below to allocate the child buffer.

MIL function	Monochrome parent buffer	Multi-band parent buffer
MbufChild1d()	1D region of first row of band.	1D region of first row of all bands.
MbufChild2d()	2D region of band.	2D region of all bands.
MbufChildColor()	-	Entire area of band chosen.
MbufChildColor2d()	-	2D region of band chosen or 2D region of all bands.

Allocate a child buffer by specifying its size and offset with respect to each of the parent buffer dimensions. Note that, as a subset of the parent buffer, a child buffer cannot exceed the bounds of its parent in any dimension. For example, a color child buffer cannot be created from a monochrome parent buffer.

Once you have finished using a child buffer, you must delete it using **MbufFree()**, before freeing the parent buffer.

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 entire image buffer to another buffer, using **MbufCopy()**.
 - 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 condition buffer, using **MbufCopyCond()**. Source buffer data is copied to the destination buffer if corresponding data in the specified condition 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.
- ❖ Note that only **MbufCopy()** copies the entire buffer into another buffer, while the other functions copy only portions of a buffer.

If the source and destination buffers have different depth and size, MIL converts data according to the following general rules:

Case	Result
Source depth > destination depth	The most significant bits are truncated when the data is copied into the destination.
Destination depth > source depth	The source data is zero or sign-extended (depending on the type of the source) when copied into the destination.
Destination size > source size	Exceeding areas of the destination buffer are unaffected.

If the source and destination buffers have the same number of band(s), all band(s) will be copied correspondingly. Otherwise, the following rules apply:

When copying from	Result
3-band to 1-band	Only the first band of the three is copied (for example, the Y band of a YUV buffer and the R band of a RGB buffer).
1-band to 3-band	All the three bands of the destination buffer are filled with the same data. Note that if the source buffer is associated with a LUT buffer, it will be first mapped through the LUT.

MIL automatically handles data type and data format conversions. When copying from a floating-point buffer to an integer buffer, the values are truncated. When converting an M_RGB15 buffer into an M_BGR24 buffer, the least-significant bits of each band are set to 0.

MIL also automatically handles different compression types.

When copying from	Result
M_COMPRESS to uncompressed	The data will be automatically decompressed.
Uncompressed to M_COMPRESS	The data will be automatically compressed.
M_COMPRESS to M_COMPRESS (with different compression types)	The data will be re-compressed according to the settings in the destination buffer.

- ❖ Note that when copying from a non-binary buffer to a binary buffer, all non-zero pixels in the source buffer are represented as ones (1) in the binary buffer. When copying a binary buffer to a buffer of a different depth, each bit is copied into the least-significant bit of a different destination pixel. The remaining bits of the destination pixel are set to 0; to propagate the bit value to all bits, use `MimBinarize()`.

Processing a non-rectangular region of interest

To process a non-rectangular region of interest, you can apply a mask to the source image. For example:

1. Allocate an image buffer, using **MbufAlloc...** The buffer should be at least the same size as the region of interest.
2. Draw a filled shape in the mask buffer corresponding to the pixels to be modified, using the graphics functions (**Mgra...**) or the drawing functions (**M...Draw**). You can also annotate the image with Windows GDI annotations. For more information, see the *Using GDI annotations* subsection in the *Annotating the displayed image nondestructively* section in *Chapter 20: Displaying an image*.
3. Allocate a temporary destination buffer, using **MbufAlloc...** The temporary destination buffer should be at least the same size as the region of interest.
4. Perform the operation from the source buffer to the temporary destination buffer.
5. Copy the temporary destination buffer to the source buffer, using **MbufCopyCond()**, with the mask buffer as the condition 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 functions 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 functions, 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 functions 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 functions 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, explicitly specify the required internal storage format when calling **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 display memory 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 **MbufAllocColor()** reference to determine which formats are supported on your board.

RGB buffers

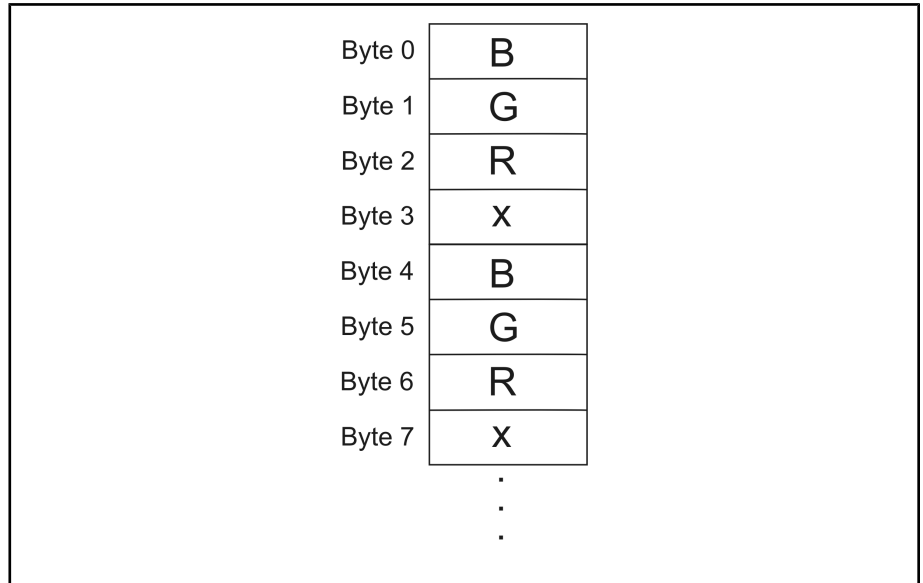
By default, MIL allocates color image buffers in an RGB color format. The pixels are internally stored in little-endian order, that is, they are stored in memory from their least-significant to the most significant bytes. The definitions of the RGB formats that are available are shown here. The corresponding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

RGB data formats

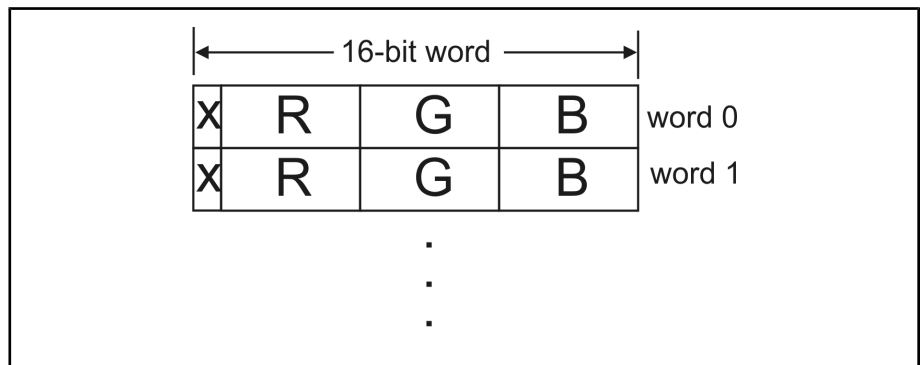
BGR24 packed (**M_BGR24+M_PACKED**) is a format whereby each pixel is internally stored as three consecutive bytes in little-endian order, that is:

Byte 0	B
Byte 1	G
Byte 2	R
Byte 3	B
Byte 4	G
Byte 5	R
	.
	.
	.

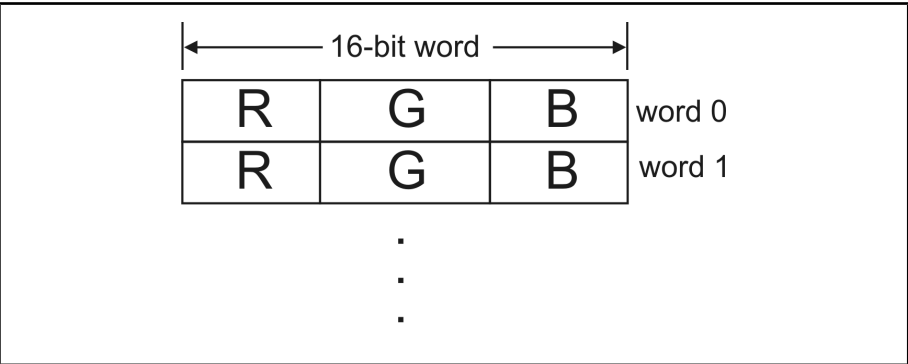
BGR32 packed (**M_BGR32+M_PACKED**) is a format whereby each pixel is internally stored as four consecutive bytes, in little-endian order. The most-significant byte is a *"don't care"* byte, as shown below:



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 left-most pixel of an image is the least significant bit that is stored in memory.

YUV buffers

YUV is a 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 **MbufAllocColor()** reference 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.
- YUV24 Planar.

YUV16 Packed

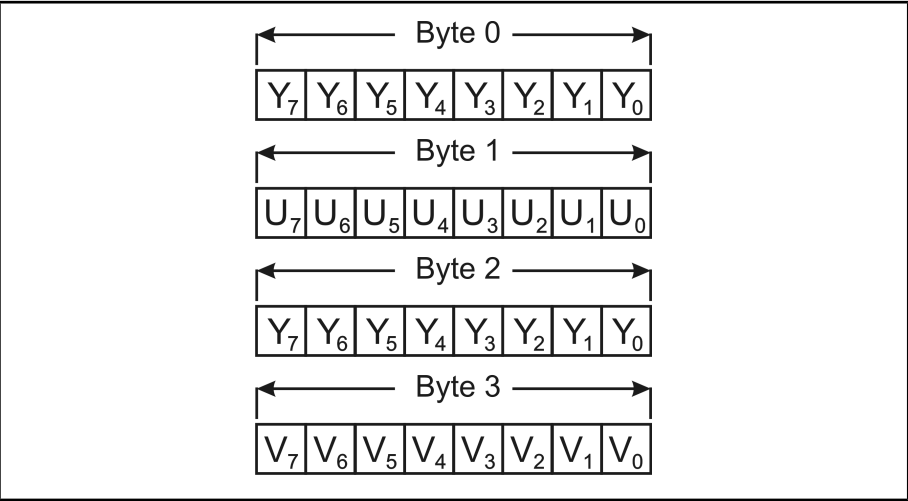
YUV16 Packed or YUV 4:2:2 (**M_YUV16+M_PACKED**) is an interleaved data format. Although each pixel has a corresponding one byte Y (luminance component), each pair of pixels share the same one byte U (chrominance U) and the same one byte V (chrominance V). Since a pair (two pixels) is represented by 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.

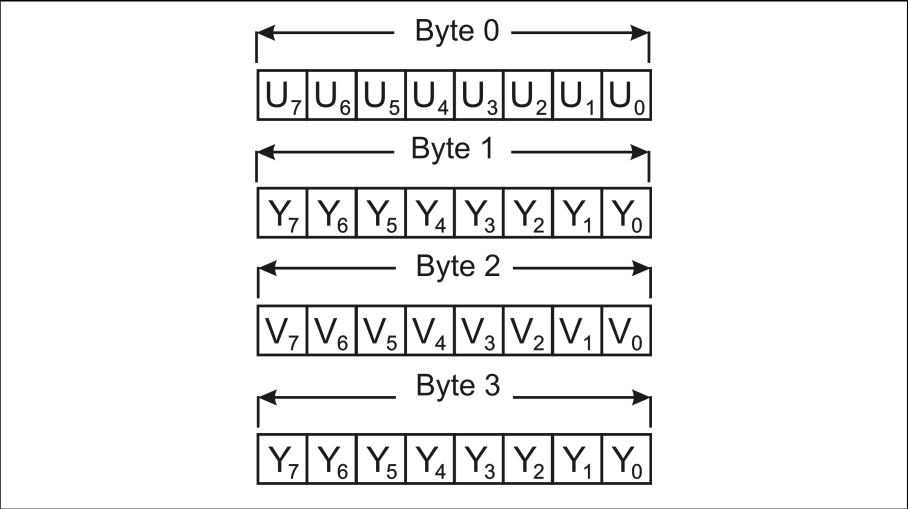
When you allocate an **M_YUV16+M_PACKED** buffer, MIL allocates the buffer in the format that is most suitable for the selected system and the specified buffer attributes. You can, however, force a format using the **M_YUV16_YUYV** or **M_YUV16_UYVY** control types. When the buffer has an **M_GRAB** 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 **MbufAllocColor()** reference for supported data formats.

The following image is an example of a YUYV buffer:

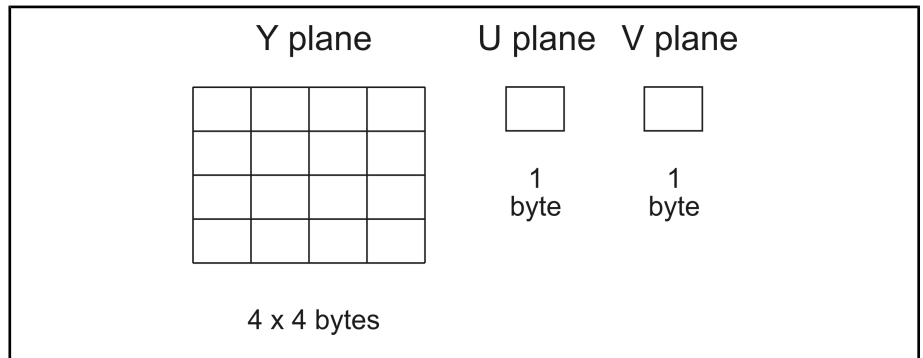


The following image is an example of a UYVY buffer:

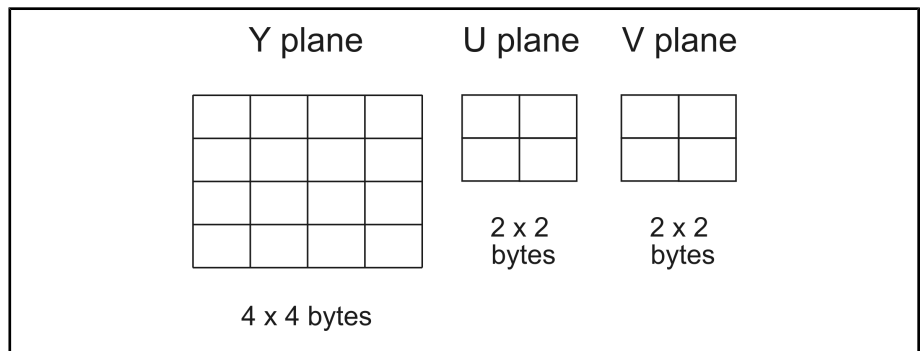


YUV9 Planar

YUV9 Planar (**M_YUV9+M_PLANAR**) is a planar format whose components have a depth of one byte but are not of the same size. Although each pixel has a corresponding 1 byte Y (luminance) component, each block of 16 pixels share the same one byte of U (chrominance U) and the same one byte of V (chrominance V). Since the 16 pixels are represented by 18 bytes, each pixel has an average 9 bits. For example, a block of 16 pixels has the following: 16 bytes Y and 1 byte each of U and V.

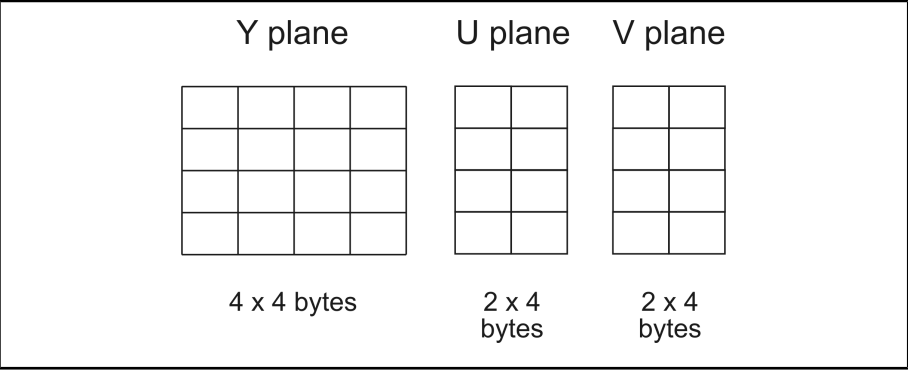
**YUV12 Planar**

YUV12 Planar (**M_YUV12+M_PLANAR**) is a planar format whose components have a depth of one byte but are not of the same size. Although each pixel has a corresponding 1 byte Y (luminance) component, each block of 4 pixels share the same one byte of U (chrominance U) and the same one byte of V (chrominance V). Since the 16 pixels are represented by 24 bytes, each pixel has an average of 12 bits. For example, a block of 16 pixels has the following: 16 bytes Y and 4 bytes each of U and V.



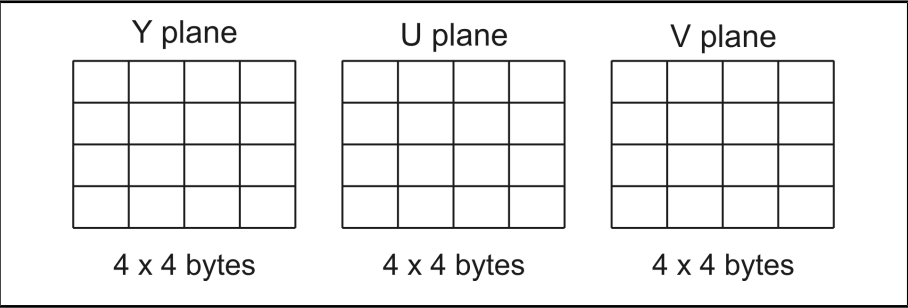
YUV16 Planar

YUV16 Planar (**M_YUV16+M_PLANAR**) is a planar format whose components have a depth of one byte but are not of the same size. Although each pixel has a corresponding 1 byte Y (luminance) component, each block of 2 pixels share the same 1 byte of U (chrominance U) and the same 1 byte of V (chrominance V). Since the 16 pixels are represented by 32 bytes, each pixel has an average 16 bits. For example, a block of 16 pixels has the following: 16 bytes Y and 8 bytes each of U and V.



YUV24 Planar

YUV24 Planar (**M_YUV24+M_PLANAR**) is an uncompressed planar format whose components have a depth of one byte and are of equal size. Each pixel has a corresponding 1 byte Y (luminance) component, 1 byte U component (chrominance U), and 1 byte V component (chrominance V). Since the 16 pixels are represented by 48 bytes, each pixel has an average 24 bits. For example, a block of 16 pixels has the following: 16 bytes Y and 16 bytes each of U and V.



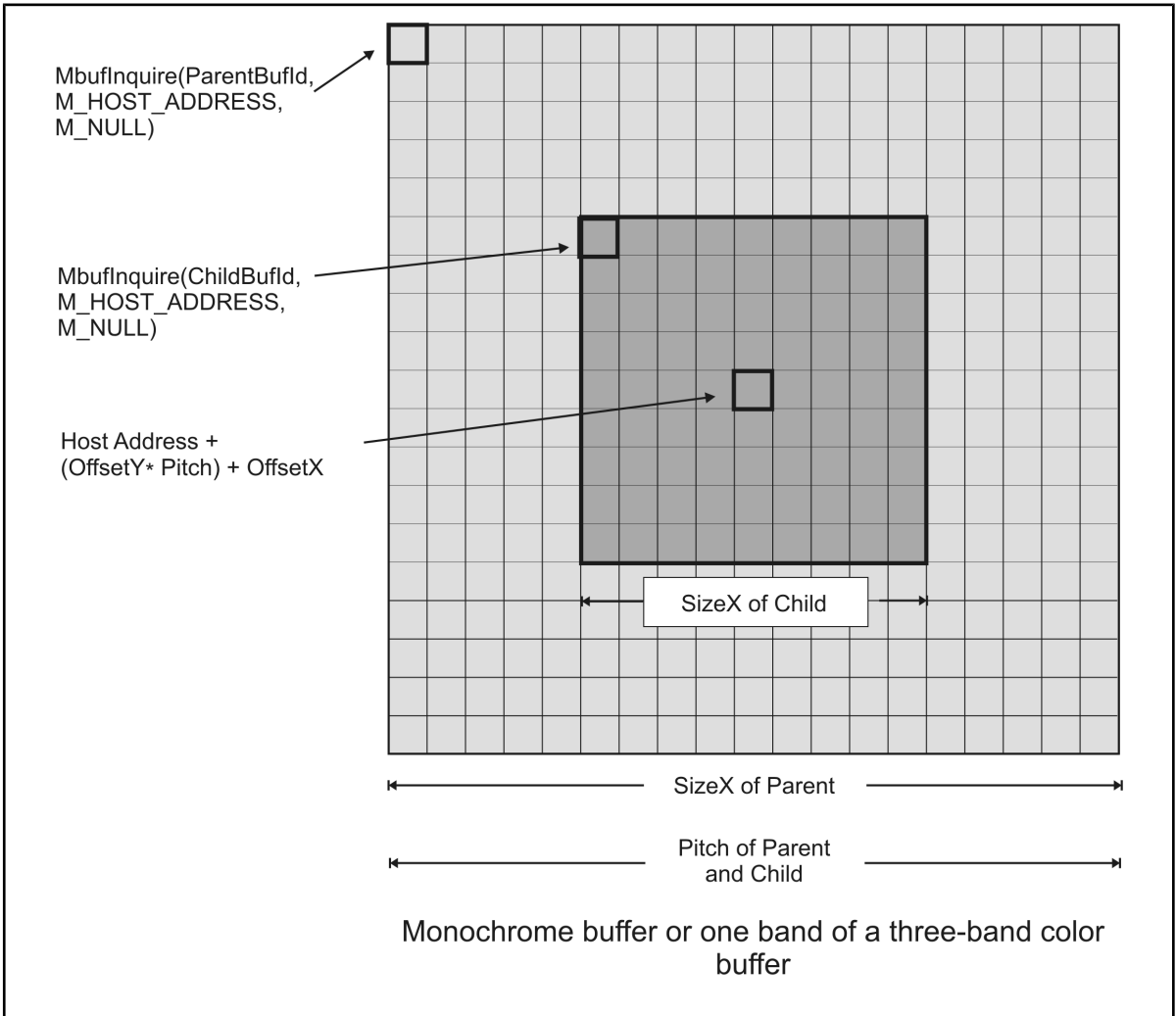
Child YUV buffers

You can create child buffers from YUV buffers in the same way as RGB child buffers. When creating YUV child buffers, MIL will keep the proportions of the U and V bands with respect to the Y band. For example, if your YUV9 Planar Y band is a size of 256 x 256 pixels, the U and V bands will be 1/4 the size of the Y band in each dimension (width and height): 64 x 64 pixels, which is 1/16 the size of the Y band. If a child buffer is 16 x 16 pixels, then the U and V bands will be 4 x 4 pixels. In other words, the 4 x 4 U and V bands (16 pixels) is 1/16 the size of the Y band (256 pixels).

Accessing a MIL buffer directly

If needed, a MIL buffer's contents can be accessed directly. For instance, if you want to calculate the average value of the pixels of your image, you could create a custom algorithm. The algorithm could be applied directly to the buffer without having to copy the contents of the MIL buffer into a user-allocated array (**MbufAlloc...**) by using **MbufGet()** and **MbufPut()**. To do so would be more efficient and might improve the performance of the custom algorithm.

In order to access the MIL buffer directly, the buffer's address and pitch must be known. Once you know this, you will be able to access them directly for optimum performance.



Address

The address of a parent or child buffer can be returned using **MbufInquire()**. Selecting **M_HOST_ADDRESS** will return a logical address, while **M_PHYSICAL_ADDRESS** will return a physical address. In either case, the first address of the buffer you are specifying will be the top left-most pixel in the image. Knowing the pitch and the depth of the buffer will tell you the address of the following row.

Pitch

The pitch of a buffer is the number of units between the beginnings of any two adjacent lines of the buffer's data and can be measured in pixels or bytes. Note that in some instances, the pitch in bytes will be more accurate than in pixels. If the last pixel falls outside of a 32-bit boundary (required by Windows), the start of the next row will be located at the beginning of the next 32-bit boundary; this process is called internal padding. When measuring the pitch in pixels, the padding can be counted as "extra" pixels, depending on the depth of the pixels. This will result in an inaccurate pitch.

Mapping a data buffer to user-allocated memory

Instead of allocating new memory to a buffer using **MbufAlloc...**, you can create a buffer from the memory at a specified location, using **MbufCreate2d()** to create a monochrome data buffer and **MbufCreateColor()** to create a color data buffer. In these cases, MIL does not allocate any memory; it uses the memory that you provide.

When creating a buffer with **MbufCreateColor()**, you can pass an array of data pointers. For packed color buffers, you can pass an array of one pointer; for planar buffers, you can pass an array with the same number of pointers as the number of bands in the buffer. When creating a buffer with **MbufCreate2d()**, you can 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 access to any of your computer's memory mapped devices; when using logical addresses, memory protection errors could result.

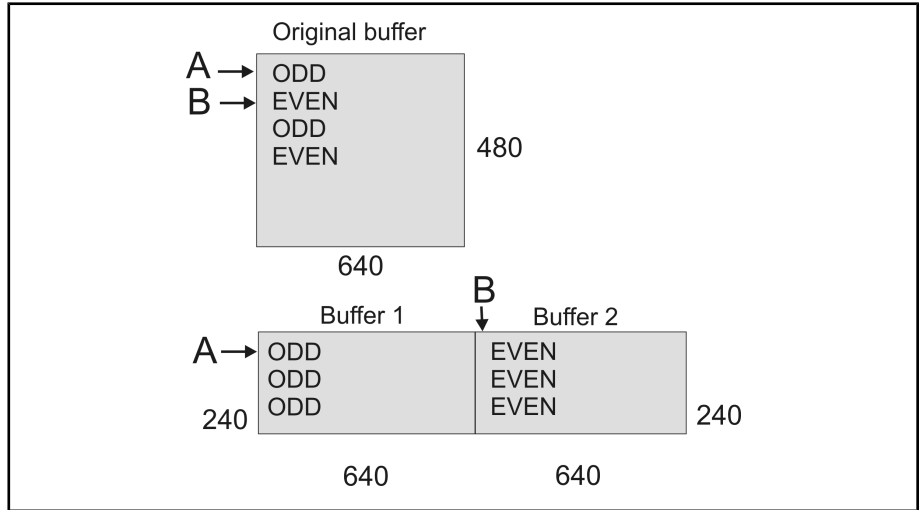
You can use **MbufInquire()** with the **M_HOST_ADDRESS** or **M_PHYSICAL_ADDRESS** inquire type to determine the Host's logical address or the physical address of a buffer's data, respectively. Note that the physical address is not necessarily an address in Host memory. It could be an address in on-board 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.

Both **MbufCreate2d()** and **MbufCreateColor()** can also be used to create a buffer that maps to an already existing buffer. When mapping to an existing buffer, pass the MIL identifier of the buffer to **MbufCreate2d()**, or a pointer to the MIL identifier of the buffer to **MbufCreateColor()**. Note that on some Matrox frame grabbers, on-board buffers are allocated in unshared memory by default. In this case, **MbufCreate2d()** and **MbufCreateColor()** are not supported, unless creating the buffer on the same system.

There are several instances when memory mapping is useful. A particularly useful instance is when processing and displaying an interlaced grab in a time critical application. In this case, you could use a displayable buffer to store and display the grabbed data. Then, to process each field as it is grabbed, you could use a buffer that is mapped to the odd field of the displayable buffer (Buffer 1) and a buffer that is mapped to the even field (Buffer 2).

- ❖ Note that the following example does not work with a created buffer with an **M_DIB** storage format because, if the pitch is in pixels (**MbufCreate...** with **M_PITCH**), the pitch must be equal to the width. If the pitch is in bytes (**MbufCreate...** with **M_PITCH_BYTE**), the pitch value must be the next multiple of 4 that is larger or equal to ($\text{SizeX} * \text{bytes per pixels}$).

Create Buffers 1 and 2 as follows:



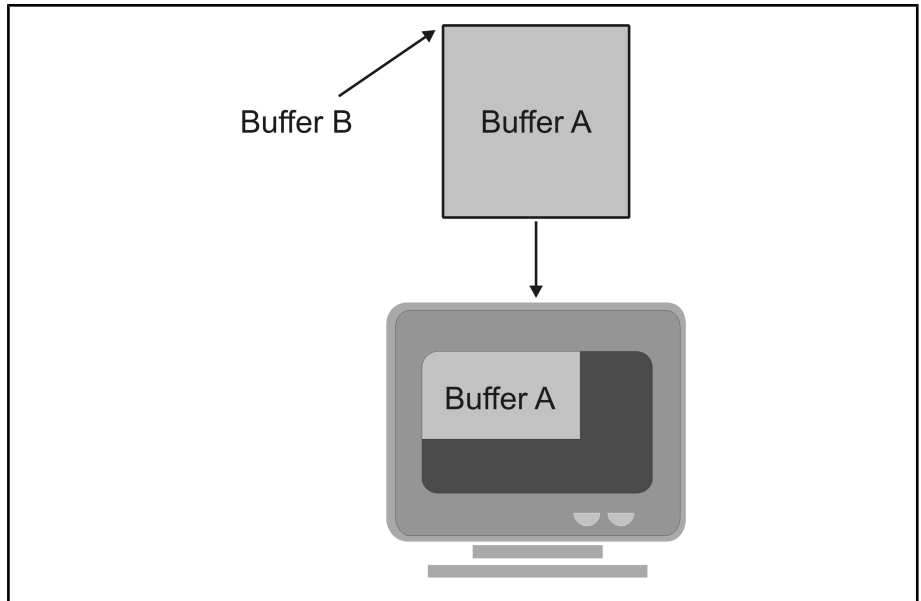
Buffer 1 (Odd field):

- 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 the **M_MODIFIED** setting in **MbufControl()**.



For another instance where creating buffers is useful, see the *Multiple systems* section in *Chapter 2: Building an application*.

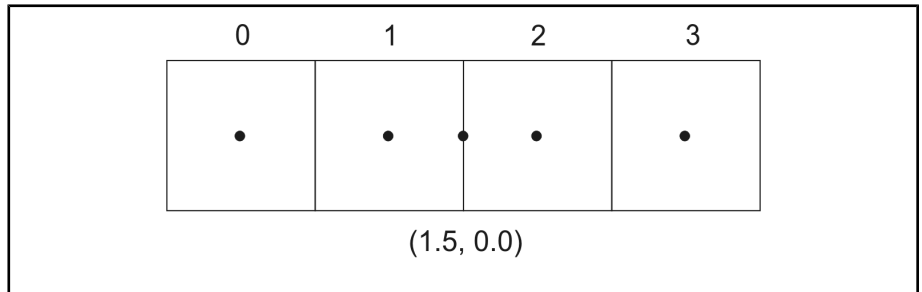
Pixel conventions

The center of a pixel is important for all MIL functions which return positional results with subpixel accuracy. The reference position of a pixel is its center, and the resulting subpixel coordinates are with respect to the pixel's center.

With this in mind, the coordinates of the center of an image can always be found using the following formula:

$$\left(\frac{\text{Width}-1}{2}, \frac{\text{Height}-1}{2} \right)$$

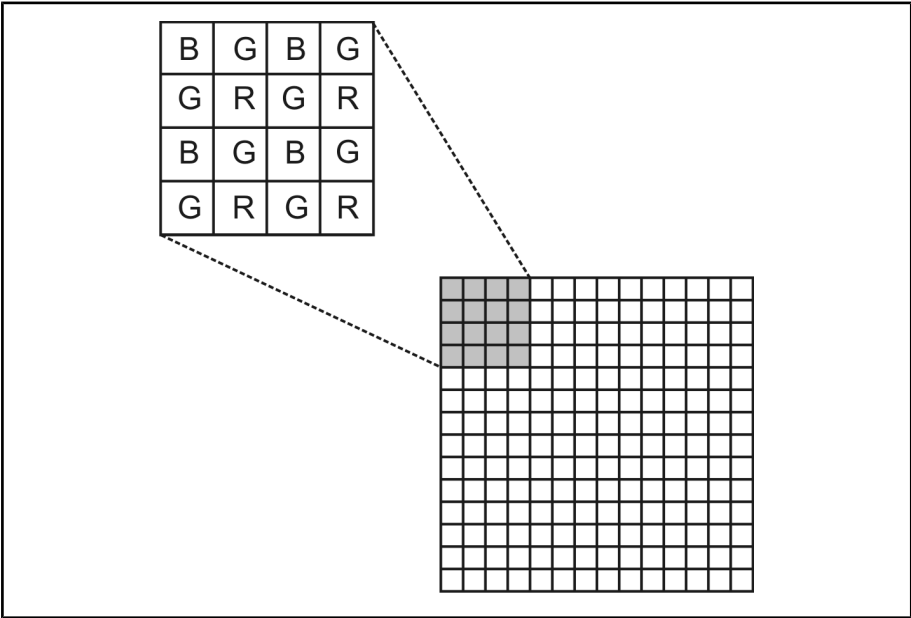
For example, the following image contains 4 pixels. If the formula is applied, the center of the image is found at (1.5, 0).



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 you can convert 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 perform color correction, white balance correction, and gamma correction on the Bayer image during conversion. Color correction adjusts an image for color artifacts that are introduced during the demosaicing operation and that appear on the transition edges of the image. White balance correction adjusts an image for color variations that were introduced by the lighting conditions when the image was grabbed; **MbufBayer()** converts pixels that represent white so they appear as close to white as possible, and adjusts other pixels accordingly. Gamma correction changes the overall brightness and color saturation of an image so that the midtones appear more realistic on your display device; gamma correction effectively adjusts an image to compensate for the non-linear relationship between a pixel's value and its displayed intensity on your display device. Color correction, white balance, and gamma correction are discussed in greater detail later in this section.

Note that if the scale of the image is changed (using **MdigControl()** with **M_GRAB_SCALE...**) to a value other than 1 prior to grabbing a Bayer image, the Bayer image will not be converted properly; some of the Bayer pattern is lost during the scaling process, rendering color recovery impossible.

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 correction coefficients (optional). For information on how to calculate the coefficients, see the *Correcting the white balance of your Bayer images* subsection in this section.
2. Determine the gamma correction exponents (optional). For information on how to calculate the exponents, see the *Performing gamma correction* subsection in this section.
3. Grab or load a Bayer image into your source buffer.
4. Apply the MIL Bayer filter on the image using **MbufBayer()**, including the white balance correction coefficients and/or gamma correction exponents, if they are being used. To use the adaptive demosaicing algorithm, you must also add **M_ADAPTIVE** to the **ControlFlag** parameter.

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.

```

/*****
/*
 * File name: MBufBayerColorFilter.cpp
 *
 * Synopsis: This program shows how to perform Bayer-to-Color conversion.
 */

#include "mil.h"
#include "conio.h"
void MosMain(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. */
    MIL_INT ConversionType = M_BAYER_GR;
    /* Buffer characteristics. */
    MIL_INT XSize;
    MIL_INT YSize;

    /* Allocate an application. */
    MappAllocDefault(M_SETUP, &MilApplication, &MilSystem, &MilDisplay,
    &MilDigitizer, M_NULL);
    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. */
    MbufAllocId(MilSystem, 3, 32+M_FLOAT, M_ARRAY, &MilWBCoefficients);

    /* Display the image. */
    MbufClear(MilImageDisp, M_RGB888(0, 0, 0));
    MdispSelect(MilDisplay, MilImageDisp);
    /* Ask the user for a white image for white balance. */
    MosPrintf(MIL_TEXT("Place a white paper in front of the camera \
and press <ENTER> when ready.\n"));

```

```

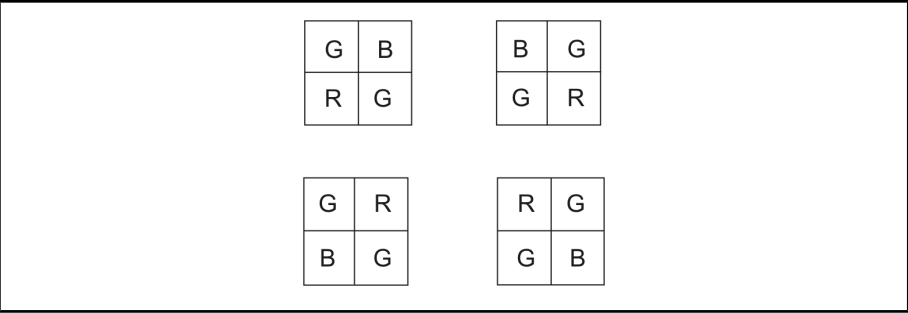
do
{
/* Grab a white Bayer image. */
MdigGrab(MilDigitizer, MilImageGrab);
/* Convert the white Bayer image to color without white balance. */
MbufBayer(MilImageGrab, MilImageDisp, M_DEFAULT, ConversionType);
}
while (!MosKbhit());
MosGetch();
/* Determine the white balance coefficients. */
MbufBayer(MilImageGrab, MilImageDisp, MilWBCoefficients,
ConversionType+M_WHITE_BALANCE_CALCULATE);
/* Print the computed coefficients. */
MbufGet(MilWBCoefficients, (void *) &WBCoefficients[0]);
MosPrintf(MIL_TEXT("\nWhite balance correction coefficients : %f, %f, %f\n\n"),
WBCoefficients[0], WBCoefficients[1], WBCoefficients[2]);
/* Grab a new Bayer image with white balance correction. */
MosPrintf(MIL_TEXT("Press <ENTER> to grab an image\n"));
MosGetchar();
do
{
/* Grab a Bayer image. */
MdigGrab(MilDigitizer, MilImageGrab);
/* Convert the Bayer image to color. */
MbufBayer(MilImageGrab, MilImageDisp, MilWBCoefficients, ConversionType);
}
while (!MosKbhit());
MosGetch();
MosPrintf(MIL_TEXT("Press <ENTER> to end\n"));
MosGetchar();

/* 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 pixels. 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 the **MbufBayer()** function's supported patterns to find the correct one.

Once you have specified the Bayer pattern of your camera, you must consider which demosaicing algorithm to use to convert a single-band Bayer color-encoded image to a multi-band color image. There are currently two from which you can choose from: the bilinear interpolation demosaicing algorithm and the adaptive demosaicing algorithm.

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.

Regardless of the destination buffer format, MIL converts the Bayer image first to RGB, and then to the format of the destination buffer.

- ❖ This is the default demosaicing algorithm used by **MbufBayer()**.

Adaptive algorithm

The bilinear demosaicing algorithm is simple, fast, and does an effective job of demosaicing the Bayer images. However, since it uses only the neighborhood averages to determine the pixel values, details are sometimes lost and sporadic artifacts are introduced. To minimize image artifacts, such as the zipper effect and false colors, you can use the adaptive demosaicing algorithm to convert your images.

The adaptive demosaicing algorithm analyzes the surrounding neighborhood of the source pixel and assigns weights to its neighbors. MIL then uses these weights in the calculation of the destination pixel value, depending on the neighbor's relevancy to the source pixel.

To use the adaptive demosaicing algorithm, add **M_ADAPTIVE** to the **ControlFlag** parameter.

Average algorithm

The average algorithm is the fastest demosaicing algorithm. The average algorithm uses a 2x2 neighborhood kernel to perform the demosaicing. When using this algorithm the zipper effect is minimal, but the image is shifted by half a pixel to the left and by half a pixel upwards and some false colors can be introduced.

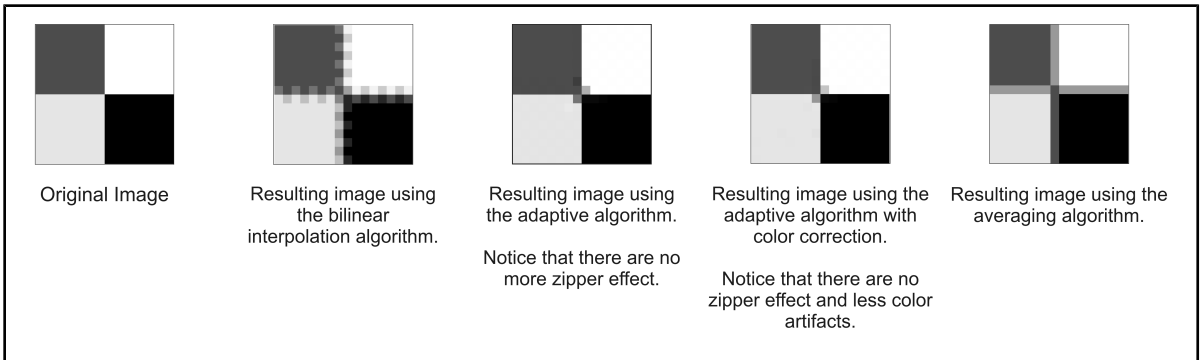
To use the average demosaicing algorithm, add **M_AVERAGE_2X2** to the **ControlFlag** parameter.

Performing color correction

Sometimes grabbed images appear with a color fringe on transition edges. To minimize these color artifacts on the edges, you can perform a color correction on the image during the Bayer conversion.

Color correction can only be used in conjunction with the adaptive demosaicing algorithm and is designed to produce very good looking vertical and horizontal edges. For best visual results, you should combine **M_ADAPTIVE** and **M_COLOR_CORRECTION** together.

The following compares the results of the two demosaicing algorithms and the effect of color correction.



Correcting the white balance of 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 performing a white balance correction on the image.

On the premise that white pixels should contain no chrominance, white balance correction applies a coefficient to each band of the image so that "white" pixels contain no chrominance. For RGB images, this means that a white pixel's value for all 3 bands is equal. After white balance correction, 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 should be a uniform shade of gray. To do so, hold a flat, smooth, non-reflective object that is a uniform shade of gray in front of a camera so that the object occupies the camera's entire field of view. The shade of gray should not be too close to white (too saturated) because if one or more color bands is actually being clipped to maximum saturation, the unbalance between the bands cannot be observed.

Ensure that the image is grabbed under the same lighting conditions as subsequent source images. Note that it is unlikely that you will be able to grab an image whose pixels have the same red, green, and blue value.

2. Allocate a 3 x 1 (or 6 x 1 if gamma is used) MIL array of type **M_FLOAT** using **MbufAlloc1d()** or **MbufAlloc2d()**.
3. Call **MbufBayer()**, using the gray image as the source image, and adding **M_WHITE_BALANCE_CALCULATE** to the **ControlFlag** parameter. This call calculates the three coefficients required to white balance the specified source image and writes them to the array.

Alternatively, you could fill an array with custom values, and then load these values (in the order described below) into the MIL array using **MbufPut1d()** or **MbufPut2d()**.

4. Grab the image required for the Bayer conversion.
5. Call **MbufBayer()**, using the new source image and the MIL array with the white balance coefficients.

RGB images

When **MbufBayer()** calculates the coefficients, the three white balance correction coefficients, a , b , and c , are calculated and written 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 gray surface in that lighting:

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

where \bar{R} , \bar{G} , and \bar{B} with macrons ($\bar{}$) are the average values of the intermediate red, green, and blue color components, respectively. When source images are converted, the intermediate color components of each pixel are multiplied by their corresponding coefficient. Regardless of the destination buffer format, white balance correction is performed on the intermediate RGB values recovered by the demosaicing process (before being converted into the proper destination format).

Performing gamma correction

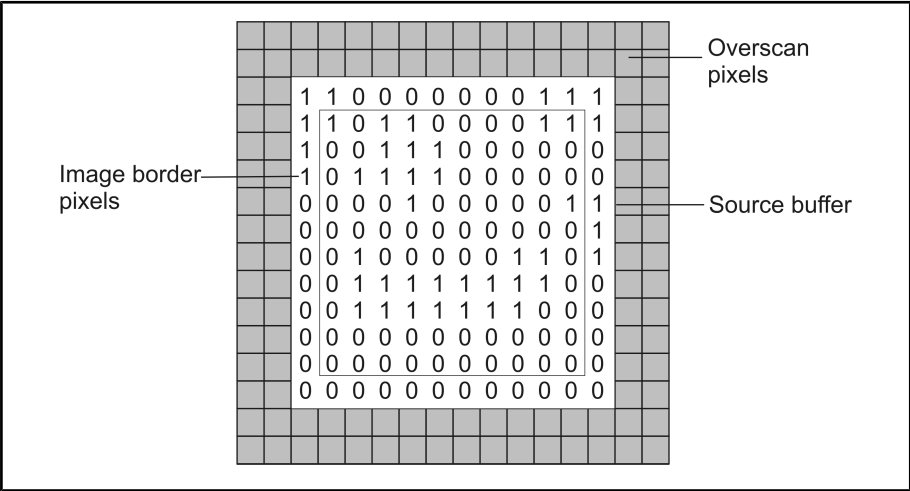
Sometimes the colors of grabbed images can appear to be over-saturated or too dark due to the display device used. Most display devices receive your image as an analog video signal. The voltage range of the active portion of the video signal is divided linearly among the possible pixel value range. However, the intensity to voltage response ratio of the display device is roughly a power function (intensity = voltage^{gamma}), whereby the exponent is rarely 1 (for many display devices, it is 2.5). This means that there is typically a non-linear relationship between a pixel's value and its displayed intensity on your display device. Therefore, images can look either bleached out or too dark.

The exponent of the power function of each display device is referred to as the gamma of the display device. You can compensate for the gamma of your display device by performing gamma correction on your images. That is, you can apply a power function that uses a $1/\text{gamma}$ exponent, on each band of your images. The exponents must be positive, are typically in the range of 0.0 to 1.0, and are most commonly $1/2.5 = 0.4$.

If you pass the required exponents to the **MbufBayer()** function, the function can perform gamma correction on the intermediate RGB values recovered by the demosaicing process (just after performing the white balance correction operation and before converting the image into the proper destination format). Pass the exponents in the forth, fifth, and sixth elements of the MIL array buffer.

Buffer overscan region

Buffer overscan pixels are pixels added around an image buffer. MIL neighborhood operations, such as **MimConvolve()**, use these pixels to perform operations that go beyond the limits of the buffer. For example, in the image below, if **MimConvolve()** is called with the **M_HORIZ_EDGE** FIR filter, which operates on 3 by 3 neighborhoods, the neighborhoods of the image border pixels would be incomplete. In such cases, overscan pixels might be used. MIL recursive neighborhood operations, such as **MimConvolve()** called with the **M_SHEN_FILTER** IIR filter, ignore the values of overscan pixels.



To set the values of overscan pixels, use the **MbufControlNeighborhood()** function. However, some MIL neighborhood operations with predefined kernels or structuring elements do not require that you set the values of overscan pixels. For example, if you call **MimErode()**, the overscan pixels are automatically set to the highest possible buffer value, which will produce the most accurate possible results for the image border pixels.

The number of overscan pixels added around an image buffer during allocation is set using **MsysControl()**. The set size represents the number of 1-pixel high/wide rows/columns added to each side of the image buffer. For example, in the image above, the size of the overscan is 2. Each buffer allocated on a particular system will have the same overscan size. You can inquire the size of the overscan using either the **MbufInquire()** or **MsysInquire()** function.

When the results of operations performed on image border pixels are not important, and you want to speed up your application, you can use the **MbufControlNeighborhood()** function to disable the use of overscan pixels. Also, both **MimConvolve()** and **MimEdgeDetect()** allow you to disable the use of overscan pixels for predefined kernels.

Chapter

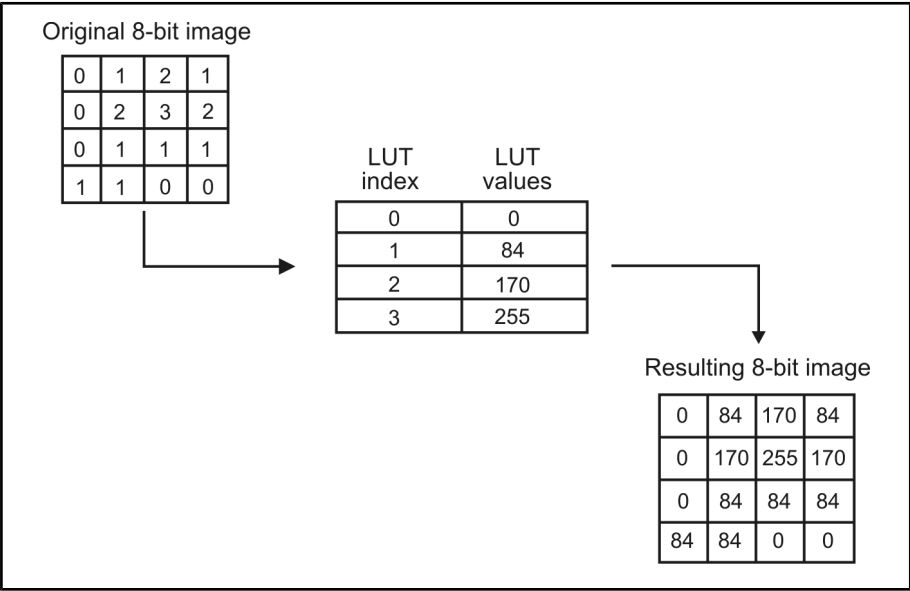
19

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 in general

Lookup tables (LUTs) are collections of memory locations that are used to map data to precalculated 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

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()**.

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()** function 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 function generates an inverse ramp. If the increment is equal to zero, it loads the entire LUT range with the given start value.

The **MgenLutFunction()** function 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()** function 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 precalculating 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 functions other than **MgenLutRamp()** (for example, using the **MimArith()** function 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 function 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).

For details, refer to the chapters that discuss image processing, displays, and acquiring data with your digitizer.

Chapter

20

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 display effects that can be achieved.

Overview

MIL can display images. It will use the most appropriate graphics controller for display purposes. MIL will typically display the image on the computer running the main MIL application (master computer); however, MIL can also display an image on a remote computer on your network. If your imaging board has a display section, and it is available and appropriate, MIL will typically use it for display purposes.

Displayable image buffers

To display an image buffer, you must allocate the buffer with a displayable attribute (**M_DISP**). In addition, you must allocate a display, using **MdispAlloc()** or **MappAllocDefault()**, on the same system as the buffer. Once both are allocated, use **MdispSelect()** to select the image buffer to display.

The buffer can be displayed, on the master or remote computer, in a window on the desktop or without a window on a dedicated (auxiliary) screen. For some systems (for example, Matrox Iris), MIL also supports publishing the display on the network so that the selected image buffer can be viewed at a remote computer in, for example, a web browser.

Besides other display effects, you can pan and zoom the displayed image, as well as overlay annotations on the displayed image non-destructively. If you use MIL's overlay-display mechanism, you can actually overlay any image on the displayed image and select the transparency color; when you pan and zoom the displayed image, the overlay data is also panned and zoomed. When displaying in a window under Windows, you can also access the display's window DC and draw annotations that are not affected by panning and zooming.

You can interactively select a region of interest in a display, inquire the location of this region, and then act upon it.

- ❖ An image buffer, or any of its child buffers, can be selected on more than one display.

Frame buffer

MIL documentation uses the term **display memory** to refer to physical display (graphics controller) memory.

Supported hardware acceleration modes

Under Windows, MIL uses one of three hardware-acceleration display modes to render images on the screen.

- **Standard hardware acceleration mode.** This mode uses the Windows graphics device interface (GDI) functions of the Windows API to render images on the screen. This mode does not directly use graphics acceleration hardware; therefore, it works the same under all Windows implementations.
- **Direct3D hardware acceleration mode.** This mode allows MIL to use Direct3D (Microsoft DirectX version 9.0) to manipulate internal display buffers, when possible. In addition, the final blit to screen is done using Direct3D if you disable the possibility of drawing directly into the display's window (**MdispControl()** with **M_WINDOW_ANNOTATIONS**); otherwise, the final blit is done using GDI functions.
- **DirectDraw 7 hardware acceleration mode.** This mode uses DirectDraw 7 (Microsoft DirectX version 7.0). This was the only mode used prior to MIL 9.0.

Each mode has its own advantages and disadvantages. By default, MIL selects the best hardware-acceleration display mode based on the current operating system and hardware available. You can change the default mode using **MILConfig** or you force a specific mode using **MappAlloc()** with **M_DX_VERSION**. It is recommended to use DirectDraw 7 if using a Matrox display board or a Matrox imaging board with a display section.

	Hardware acceleration mode				
Features	Standard	DirectDraw 7 (Matrox display board)	DirectDraw 7 (non-Matrox display board)	Direct3D 9 (final blit done using GDI functions)	Direct3D 9 (final blit done using Direct3D)
Requires Direct3D-capable display board, with at least Vertex Shader version 3.0	No	No	No	Yes	Yes
Supports applications that also use native Direct3D calls (but not DirectDraw calls)	No	No	No	Yes	Yes
Supports applications that also use native DirectDraw calls (but not Direct3D calls)	No	Yes	Yes	No	No
Supports drawing directly in the display's window (using the window's DC)	Yes	Yes	Yes	Yes	No
Supports auxiliary displays	No	Yes*	No	Yes**	Yes**
Compatible with no-tearing	No	Yes	Yes***	No	Yes
Supports grabbing directly in display memory	Yes	Yes	Yes***	No	No
Recommended when using remote desktop (or other 3rd party remote tools)	Yes	No	No	No	No
Compatible with Vista Aero Glass mode	Yes	No	No	Yes	Yes
Compatible with GPU systems	No	No	No	Yes	Yes
*You must enable this feature using MILConfig before you can use it in MIL. **Available only for Matrox imaging boards with a display section. ***For non-Matrox display boards, support depends on the board. Therefore, you must enable this feature using MILConfig before you can use it in MIL.					

Each of these features is described in more detail in the remaining sections of this chapter. For more information on these acceleration mode, see the *Advanced display concepts* section later in this chapter.

Types of displays

You can allocate a display so that an image buffer selected to this display is:

- Displayed with a windowed border. This is called a **windowed display** (**M_WINDOWED**).
- Displayed without a windowed border on a dedicated screen. This is called an **auxiliary display** (**M_AUXILIARY**).
- Published on the network. This is called a **network display** (**M_NETWORK**).

You must specify the required type of display upon allocating the display, with **MdispAlloc()**.

Windowed display

An image buffer selected to a windowed display is presented with a windowed border on the Windows desktop screen(s). To choose a windowed display, set the **InitFlag** parameter of **MdispAlloc()** to **M_WINDOWED**.

Extended desktop

Windowed displays can be presented on a desktop that is displayed using one screen or multiple screens. We refer to these screens, either one or many, as the **Windows desktop screen(s)**. Your desktop can be extended over screens at different resolutions.

All windowed displays are displayed in their own MIL default window (or, as discussed 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.

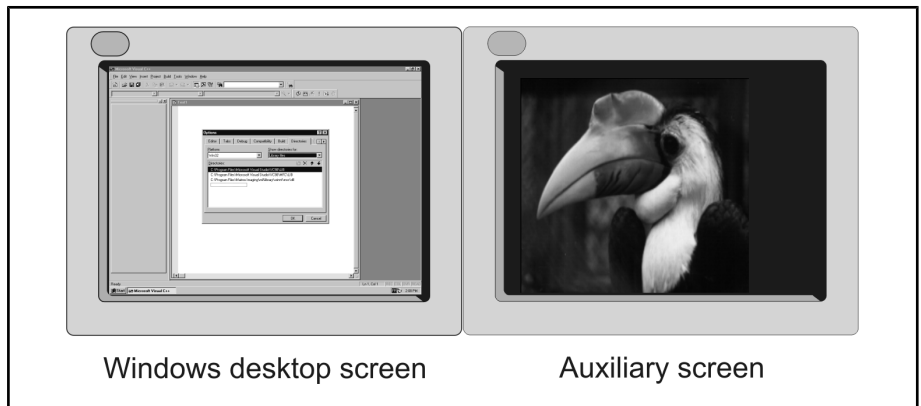
Multiple windowed displays can be allocated and selected for display; the display's device number should always be set to **M_DEFAULT**.

For windowed displays, MIL does not typically communicate directly with the graphics controller, but uses the normal Windows mechanisms to display images. Upon selecting a windowed display, MIL uses internal image buffers to store the selected image buffer and overlay information; it then uses either Windows GDI functions or a Windows DirectDraw/Direct3D surface for the final blit to screen.

When allocating a windowed display for a Distributed MIL application, you can set the **SystemId** parameter to a remote system. Image buffers selected to such a display are presented on the master computer by default. However, if required, you can allocate the display so that image buffers are presented on the remote computer. For more information on allocating such a display, see *Chapter 24: Distributed MIL*.

Auxiliary display

An image selected to an auxiliary display is presented 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 only supported under one of the following circumstances:

- You have two graphics controllers: one to drive your Windows desktop screen and one to drive your auxiliary screen. The one driving your auxiliary screen must be on a Matrox imaging board (for example, Matrox Corona-II) or it must be a Matrox G-Series or P-Series graphics controller.
- You have a Matrox graphics controller that integrates two video output controllers: one to drive your Windows desktop screen and one to drive your auxiliary screen. The graphics controller must be on your Matrox imaging board (for example, Matrox Corona-II) or it must be a Matrox G-Series or P-Series graphics controller.

To output the auxiliary display in an encoded format (for example, for a TV or VCR), you can only use a graphics controller on a Matrox imaging board.

Unless otherwise documented for a specific Matrox imaging board (for example, Matrox Vio), auxiliary displays are only supported if using DirectDraw 7. You can specify the version of DirectX to use for display using **MappAlloc()** with **M_DX_VERSION**.

Before trying to allocate an auxiliary display, you must enable detection of the graphics controller using MILConfig. Once enabled, you can allocate the display using **MdispAlloc()** with its **InitFlag** parameter set to **M_AUXILIARY**. You can only allocate one auxiliary display at a time on a given auxiliary screen.

The video output format of the auxiliary screen is set with the **DispFormat** parameter of **MdispAlloc()**. In general, you must set the display format to a supported high resolution format (for example, 1024x768x32@70hz). To output using the encoder in the on-board display section of a Matrox imaging board, set the display format to an encoded video format (for example, NTSC/PAL). For example:

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

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

To use the second video output controller of a Matrox G-Series or P-Series graphics controller for a 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 video output controller, and the auxiliary display will not work.

For auxiliary displays, the display's device number should always be set to **M_DEFAULT**.

When allocating an auxiliary display for a Distributed MIL application, you can set the **SystemId** parameter to a remote system. Image buffers selected to such a display are presented on the master computer by default. However, if required, you can allocate the display so that image buffers are presented on the remote computer. For more information on allocating such a display, see *Chapter 24: Distributed MIL*.

Network display

An image selected to a network display is published on the network so that image buffers selected to the display can be viewed from any remote computer on your network in, for example, a web browser. In this case, there are no special requirements for the remote computer; you do not need to install MIL, a MIL license, nor Distributed MIL server on it. An application interface (for example, a web page) must be created to view the display.

- ❖ Note that network displays are only supported for some systems (for example, Matrox Iris).

To choose a network display, set the **InitFlag** parameter of **MdispAlloc()** to **M_NETWORK**. In addition, select an appropriate format for the display. For more information, see **MdispAlloc()**.

Display size and depth

For windowed displays, the display format of the on-screen portion of display memory is set using the selected Windows display resolution. In this case, the display format parameter of **MdispAlloc()** is ignored. 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 display, and the surrounding area will be blacked out.

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

Displaying buffers of different data depths

Displayable image buffers usually have a depth of 8 bits (or 8 bits per band, in the case of color images). You can also display images of other depths (for example, 1-bit or 16-bit images). Using **MdispControl()** with **M_VIEW_MODE**, you can control the way such buffers are actually displayed.

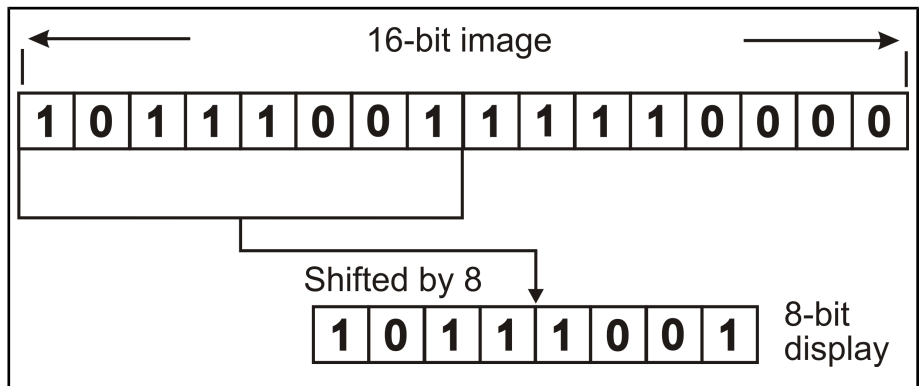
The **M_VIEW_MODE** control type provides different modes of displaying non 8-bit images:

- **M_BIT_SHIFT.**
- **M_AUTO_SCALE.**
- **M_MULTI_BYTES.**
- **M_TRANSPARENT.**
- **M_DEFAULT.**

Note that **M_VIEW_MODE** is not available for network displays.

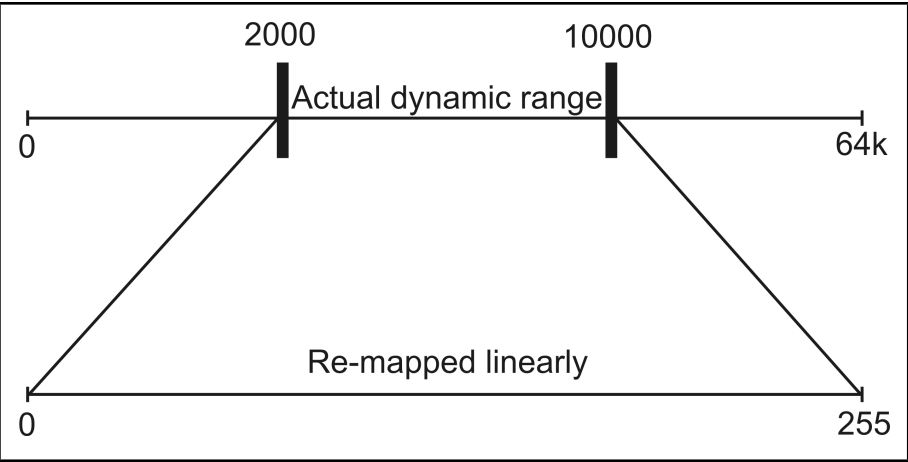
M_BIT_SHIFT

The **M_BIT_SHIFT** setting will bit-shift the pixel values of the image by the specified number of bits upon updating the display. Specify the number of bits by which to shift using the **M_VIEW_BIT_SHIFT** control type.

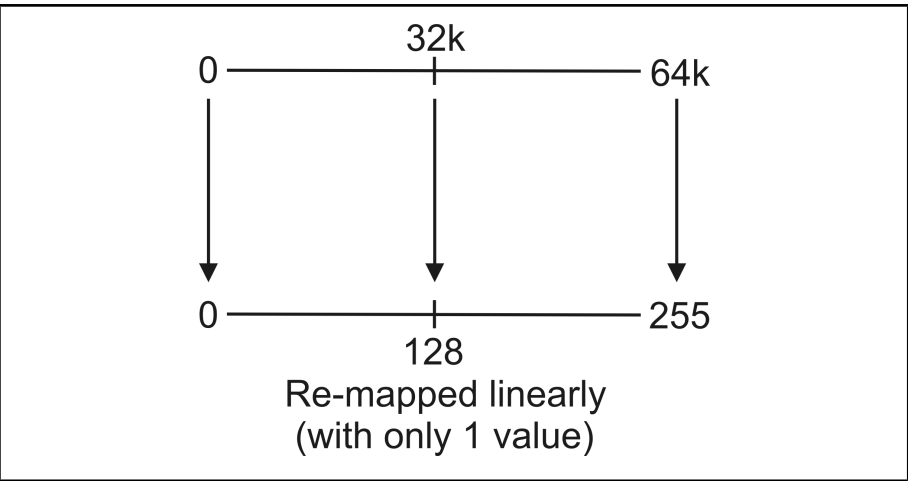


M_AUTO_SCALE

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. **M_AUTO_SCALE** is the default setting of **M_VIEW_MODE** when displaying a 1-bit buffer.



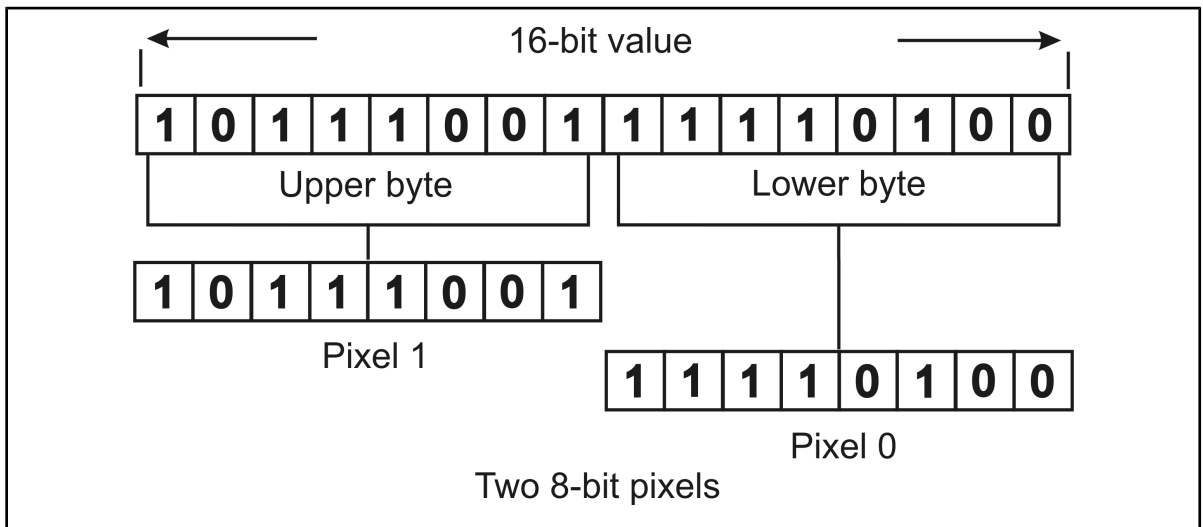
If the image buffer contains a single value, its corresponding displayed value is determined by linearly re-mapping the full range of the buffer (for example, (0 to 64K) to (0 to 255)).



- ❖ MIL-Lite does not support displaying a 32-bit image buffer in a display that has its **M_VIEW_MODE** control type set to **M_AUTO_SCALE**, unless you have purchased the MIL Image Analysis license.

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. This mode is only supported for 16-bit and 32-bit 1-band images.



M_TRANSPARENT

The **M_TRANSPARENT** setting will display only the 8 least-significant bits of the image. No pixel remapping is performed. This is the default setting unless a 1-bit buffer is used.

Displaying an image in a user-defined window

Images selected to a windowed display using **MdispSelect()** are presented in a default window created by MIL. 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 type setting associated with the display using an **Mdisp...** function.

Selecting a buffer into a specific display window

Alternatively, for windowed displays, you can choose to display image buffers 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 **MdispSelectWindow()** with **M_NULL** to remove 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()**.

To select an image to a user-defined window, the display cannot be allocated on a remote computer (**MdispAlloc()** with **M_REMOTE_DISPLAY**).

The following 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: MDispWindow.cpp
*
* Synopsis: This program displays a welcoming message in a user-defined
*           window and grabs (if supported) in it. It uses
*           the MIL system and the MdispSelectWindow() function
*           to display the MIL buffer in a user-created client window.
*/
#include <windows.h> /* windows.h must precede mil.h. */
#include <mil.h>

/* Window title. */
#define MIL_APPLICATION_NAME      MT("MIL Application")
#define MIL_APPLICATION_NAME_SIZE 128

/* Default image dimensions. */
#define DEFAULT_IMAGE_SIZE_X      640
#define DEFAULT_IMAGE_SIZE_Y      480
#define DEFAULT_IMAGE_SIZE_BAND   1

/* Background color of the window client area. */
#define BACKCOLORRED               160
#define BACKCOLORGREEN            160
#define BACKCOLORBLUE             164

/* Defines for menu. */
#define IDM_START_MAIN             100

/* Function prototypes. */
void MilApplication(HWND UserWindowHandle);
void MilApplicationPaint(HWND UserWindowHandle);

*****/
/*
* Name:      MilApplication()
*
* Synopsis:  This function is the core of the MIL application that
*           is executed when the "Start" menu item of this
*           Windows program is selected. See WinMain() below
*           for the program entry point.
*
*           It uses MIL to display a welcoming message in the
*           specified user window and to grab in it (if it is supported)
*           using the target system.
*/

```

```

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. */

    MIL_INT BufSizeX;
    MIL_INT BufSizeY;
    MIL_INT BufSizeBand;

    /* Allocate a MIL application. */
    MappAlloc(M_DEFAULT, &MilApplication);

    /* Allocate a MIL system. */
    MsysAlloc(MT("M_DEFAULT"), M_DEFAULT, M_DEFAULT, &MilSystem);

    /* Allocate a MIL display. */
    MdispAlloc(MilSystem, M_DEFAULT, MT("M_DEFAULT"), M_WINDOWED, &MilDisplay);

    /* Allocate a MIL digitizer, if supported, and set the target image size. */
    if (MsysInquire(MilSystem, M_DIGITIZER_NUM, M_NULL) > 0)
    {
        MdigAlloc(MilSystem, M_DEFAULT, MT("M_DEFAULT"), 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 = DEFAULT_IMAGE_SIZE_X;
        BufSizeY = DEFAULT_IMAGE_SIZE_Y;
        BufSizeBand = DEFAULT_IMAGE_SIZE_BAND;
    }

    /* Do not allow example to run in auxiliary mode. */
    if (MdispInquire(MilDisplay, M_DISPLAY_MODE, M_NULL) != M_WINDOWED)
    {
        MessageBox(0, MT("This example does no run in auxiliary mode."),
            MT("MIL application example"),
            MB_APPLMODAL | MB_ICONEXCLAMATION );
        goto end;
    }

    /* 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: When a MIL buffer is modified using a MIL command, the display
   automatically updates the window passed to MdispSelectWindow().
*/
MgraFont(M_DEFAULT, M_FONT_DEFAULT_LARGE);
MgraText(M_DEFAULT, MilImage, (BufSizeX/8)*2, BufSizeY/2,
        MIL_TEXT(" Welcome to MIL !!! "));
MgraRect(M_DEFAULT, MilImage, ((BufSizeX/8)*2)-60, (BufSizeY/2)-80,
        ((BufSizeX/8)*2)+370, (BufSizeY/2)+100);
MgraRect(M_DEFAULT, MilImage, ((BufSizeX/8)*2)-40, (BufSizeY/2)-60,
        ((BufSizeX/8)*2)+350, (BufSizeY/2)+80);
MgraRect(M_DEFAULT, MilImage, ((BufSizeX/8)*2)-20, (BufSizeY/2)-40,
        ((BufSizeX/8)*2)+330, (BufSizeY/2)+60);

/* Open a message box to wait for a key press. */
MessageBox(0,MIL_TEXT("\\"Welcome to MIL !!!\\" was printed"),
        MIL_TEXT("MIL application example"),
        MB_APPLMODAL | MB_ICONEXCLAMATION);

/* Grab in the user window if supported. */
if (MilDigitizer)
{
    /* Grab continuously. */
    MdigGrabContinuous(MilDigitizer, MilImage);

    /* Open a message box to wait for a key press. */
    MessageBox(0,MIL_TEXT("Continuous grab in progress"),
        MIL_TEXT("MIL application example"),
        MB_APPLMODAL | MB_ICONEXCLAMATION );

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

/* Remove the MIL buffer from the display. */
MdispSelect(MilDisplay, M_NULL);

/* Free allocated objects. */
MbufFree(MilImage);

```

end:

```

if (MilDigitizer)
    MdigFree(MilDigitizer);
MdispFree(MilDisplay);
MsysFree(MilSystem);
MappFree(MilApplication);

```

}

```

/*****/
*/
* Name:      MilApplicationPaint()
*
* synopsis:  This function can be used to complete the work done by
*            the MIL paint manager when the client window receives
*            a paint message.
*
*            This is called every time WindowProc receives a
*            WM_PAINT message
*/

void MilApplicationPaint(HWND UserWindowHandle)
{
    /* Nothing is done here because nothing gets drawn on top of
     * or around our displayed MIL buffer in the window.
     */
    UserWindowHandle = UserWindowHandle; /* line to remove warning */
}

/*****/
*/
* Synopsis:  The next functions lets the user to quickly create a Windows
*            program which will call the MilApplication() function each
*            time the Start menu item is selected. The user may also
*            use the MilApplicationPaint() function that is called
*            each time the window receives a paint message to
*            complete the work already done by the MIL paint manager.
*/

/* Prototypes for Windows application */
int WINAPI MosWinMain(HINSTANCE, HINSTANCE, LPTSTR, int);
long WINAPI MainWndProc(HWND, UINT, WPARAM, LPARAM);
BOOL InitApplication(HINSTANCE, int);

/*****/
*/
* Name:      WinMain()
*
* Synopsis:  Calls initialization function, processes message loop.
*/

int WINAPI MosWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                     LPTSTR lpCmdLine, int nCmdShow)
{
    MSG msg;

    /* Avoid warning for the unused parameter. */
    lpCmdLine = lpCmdLine;

    /* Allow only one instance of program. */

```

```

if (hPrevInstance)
{
    MessageBox(0, MIL_TEXT("Sample MIL Host Windows Application already active.\n")
        MIL_TEXT("This version sample application does not allow ")
        MIL_TEXT("multiple instances."),
        MIL_TEXT("Sample MIL Host Application - Error"),
        MB_OK | MB_ICONSTOP);
    return (FALSE);
}

/* Exit if unable to initialize. */
if (!InitApplication(hInstance,nCmdShow))
    return (FALSE);

/* Get and dispatch messages until a WM_QUIT message is received. */
while (GetMessage(&msg,(HWND)NULL,(UINT)NULL,(UINT)NULL))
{
    TranslateMessage(&msg); /* Translates virtual key codes. */
    DispatchMessage(&msg); /* Dispatches message to window. */
}

return ((MIL_INT32)msg.wParam); /* Returns the value from PostQuitMessage. */
}

/*****
/*
*   Name:      MainWndProc()
*
*   Synopsis:  Processes messages:
*               WM_COMMAND      - application menu.
*               WM_ENDSESSION   - destroy window.
*               WM_CLOSE        - destroy window.
*               WM_DESTROY      - destroy window.
*/
long WINAPI MainWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    HMENU hAppMenu;

    switch (message)
    {
        case WM_COMMAND: /* message: command from application menu. */
            switch(wParam)
            {
                case IDM_START_MAIN:
                    hAppMenu = GetMenu(hWnd);
                    EnableMenuItem(hAppMenu,MF_BYCOMMAND | IDM_START_MAIN,MF_GRAYED);
                    DrawMenuBar(hWnd);
                    MilApplication(hWnd);
                    if (!GetWindowLong(hWnd,sizeof(LPVOID)))
                    {
                        PostQuitMessage(0);

```

```

        }
        else
        {
            EnableMenuItem(hAppMenu,MF_BYCOMMAND | IDM_START_MAIN,MF_ENABLED);
            DrawMenuBar(hWnd);
        }
        break;
    default:
        return ((MIL_INT32)DefWindowProc(hWnd, message, wParam, lParam));
    }
    break;
case WM_PAINT:
    MilApplicationPaint(hWnd);
    return ((MIL_INT32)DefWindowProc(hWnd, message, wParam, lParam));
case WM_ENDSESSION:
case WM_CLOSE:
    PostMessage(hWnd, WM_DESTROY, (WORD)0, (LONG)0);
    break;
case WM_DESTROY:
    /* Free the reserved extra window LPSTR to szAppName. */
    if (GetWindowLongPtr(hWnd,sizeof(LPVOID)))
    {
        free((char *)GetWindowLongPtr(hWnd,sizeof(LPVOID)));
        SetWindowLongPtr(hWnd,sizeof(LPVOID),0L);
    }
    PostQuitMessage(0);

    default:
        return ((MIL_INT32)DefWindowProc(hWnd, message, wParam, lParam));
    }
    return(0L);
}

/*****
/*
*   Name:      InitApplication()
*
*   Synopsis:  Initializes window data, registers window class and
*               creates main window.
*/

BOOL InitApplication(HINSTANCE hInstance, int nCmdShow)
{
    WNDCLASS      wc;
    HWND          hwnd;
    MIL_TEXT_CHAR TmpName[MIL_APPLICATION_NAME_SIZE];
    LPTSTR        szAppName;

    /* Allocation and storage of application name. */
    szAppName = (LPTSTR)malloc((MIL_APPLICATION_NAME_SIZE+40) * sizeof(MIL_TEXT_CHAR));
    MosStrcpy((MIL_TEXT_PTR)szAppName, MIL_APPLICATION_NAME_SIZE+40, MIL_APPLICATION_NAME);

```

```

wc.style           = CS_VREDRAW | CS_HREDRAW;
wc.lpfWndProc      = (WNDPROC)MainWndProc;
wc.cbClsExtra      = 0;
wc.cbWndExtra      = sizeof(LPVOID)+sizeof(LPTSTR);
wc.hInstance       = hInstance;
wc.hIcon           = LoadIcon(0,IDI_APPLICATION);
wc.hCursor         = LoadCursor(0, IDC_ARROW);
wc.hbrBackground   = CreateSolidBrush(RGB(BACKCOLORRED,
                                           BACKCOLORGREEN,
                                           BACKCOLORBLUE));

wc.lpszMenuName     = MT("MILAPPLMENU");
wc.lpszClassName    = szAppName;

/* Register the class. */
if (!RegisterClass(&wc))
    return (FALSE);

/* Create the window. */
hwnd = CreateWindow (szAppName,
                    szAppName,
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    0, 0, hInstance, 0) ;

if (!hwnd)
    return (FALSE);

lstrcat(szAppName,MT(" - "));
GetModuleFileName(hInstance,TmpName,MIL_APPLICATION_NAME_SIZE);
lstrcat(szAppName,TmpName);
SetWindowText(hwnd,szAppName);

/* Set the reserved extra window LPVOID to NULL. */
SetWindowLongPtr(hwnd,0,0L);

/* Set the reserved extra window LPSTR to szAppName. */
SetWindowLongPtr(hwnd,sizeof(LPVOID),(LONG_PTR)szAppName);

/* Display the window. */
ShowWindow (hwnd, nCmdShow) ;
UpdateWindow (hwnd) ;

return (TRUE);
}

```

Removing a buffer from the display

To remove an image buffer from the display, you can use **MdispSelect()** or **MdispSelectWindow()** with **M_NULL**, depending on which function was used to select the buffer to the display. For MIL windowed displays, this closes the associated window; whereas for user-defined windowed displays, this leaves the associated window open but leaves it blank. For auxiliary displays, this leaves the display blank.

To display a different image buffer, you are not required to remove the current buffer from the display; selecting another buffer for display automatically updates the display with the new buffer.

You can only remove the entire image buffer from the display. Therefore, when displaying a parent buffer, you cannot remove one of its child buffers from the display.

Once you have finished using a display, you must free it using **MdispFree()**.

Freeing the display, or freeing the buffer currently selected to the display, produces the same visual effect as when using **MdispSelect()** or **MdispSelectWindow()** with **M_NULL**.

Displaying multiple buffers

You can view one image buffer at a time in a display, but you can view multiple image buffers using multiple displays. Select the image buffers to different displays, using **MdispSelect()**.

Using multiple windowed displays, you can view more than one buffer at the same time on the Windows desktop screen(s).

For auxiliary displays, you can have as many auxiliary displays as you have auxiliary screens; however, you can have only one auxiliary display at a time on a given auxiliary screen. To view more than one image at a time in one display, use child buffers. For example, you can display the source and destination images of an operation, using the following steps:

1. Allocate a large displayable buffer (big enough to contain the source and destination images) 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 example shows how to display multiple buffers in a single display. The source bird image, *Bird.mim*, is loaded into a child of a displayable buffer and then used as the source of an image processing operation (increasing the image luminance). The result is stored in another child of the same displayable buffer.

```

/*****
*/
* File name: MBufColor.cpp
*
* Synopsis: This program demonstrates color buffer manipulations. It allocates
* a displayable color image buffer, displays it, and loads a color
* image into the left part. After that, color annotations are done
* in each band of the color buffer. Finally, to increase the image
* luminance, the image is converted to Hue, Saturation and Luminance
* (HSL), a certain offset is added to the luminance component and
* the image is converted back to Red, Green, Blue (RGB) into the
* right part to display the result.
*
* The example also demonstrates how to display multiple images
* in a single display using child buffers.
*/
#include <mil.h>

/* Source MIL image file specifications. */
#define IMAGE_FILE M_IMAGE_PATH MIL_TEXT("Bird.mim")

/* Luminance offset to add to the image. */
#define IMAGE_LUMINANCE_OFFSET 40L

/* Main function. */
int MosMain(void)
{
    MIL_ID MilApplication, /* Application identifier. */
    MilSystem, /* System identifier. */
    MilDisplay, /* Display identifier. */
    MilImage, /* Image buffer identifier. */
    MilLeftSubImage, /* Sub-image buffer identifier for original image. */
    MilRightSubImage, /* Sub-image buffer identifier for processed image. */
    MilLumSubImage=0, /* Sub-image buffer identifier for luminance. */
    MilRedBandSubImage, /* Sub-image buffer identifier for red component. */
    MilGreenBandSubImage, /* Sub-image buffer identifier for green component. */
    MilBlueBandSubImage; /* Sub-image buffer identifier for blue component. */
    MIL_INT SizeX, SizeY, SizeBand, Type;

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

    /* Allocate a color display buffer twice the size of the source image and display it. */
    MbufAllocColor(MilSystem,
        MbufDiskInquire(IMAGE_FILE, M_SIZE_BAND, &SizeBand),

```



```

        MbufDiskInquire(IMAGE_FILE, M_SIZE_X, &SizeX) * 2,
        MbufDiskInquire(IMAGE_FILE, M_SIZE_Y, &SizeY),
        MbufDiskInquire(IMAGE_FILE, M_TYPE, &Type),
        M_IMAGE+M_DISP+M_PROC, &MilImage);
MbufClear(MilImage, 0L);
MdispSelect(MilDisplay, MilImage);

/* Define 2 child buffers that maps to the left and right part of the display
   buffer, to put the source and destination color images.
*/
MbufChild2d(MilImage, 0L, 0L, SizeX, SizeY, &MilLeftSubImage);
MbufChild2d(MilImage, SizeX, 0L, SizeX, SizeY, &MilRightSubImage);

/* Load the color source image on the left. */
MbufLoad(IMAGE_FILE, MilLeftSubImage);

/* Define child buffers that map to the red, green and blue components
   of the source image.
*/
MbufChildColor(MilLeftSubImage, M_RED, &MilRedBandSubImage);
MbufChildColor(MilLeftSubImage, M_GREEN, &MilGreenBandSubImage);
MbufChildColor(MilLeftSubImage, M_BLUE, &MilBlueBandSubImage);

/* Write color text annotations to show access in each individual band of the image.

   Note that this is typically done more simply by using:
   MgraColor(M_DEFAULT, M_RGB(0xFF,0x90,0x00));
   MgraText(M_DEFAULT, MilLeftSubImage, ...);
*/
MgraColor(M_DEFAULT, 0xFF);
MgraText(M_DEFAULT, MilRedBandSubImage, SizeX/16, SizeY/8, MIL_TEXT(" TOUCAN "));
MgraColor(M_DEFAULT, 0x90);
MgraText(M_DEFAULT, MilGreenBandSubImage, SizeX/16, SizeY/8, MIL_TEXT(" TOUCAN "));
MgraColor(M_DEFAULT, 0x00);
MgraText(M_DEFAULT, MilBlueBandSubImage, SizeX/16, SizeY/8, MIL_TEXT(" TOUCAN "));

/* Print a message. */
MosPrintf(MIL_TEXT("\nCOLOR OPERATIONS:\n"));
MosPrintf(MIL_TEXT("-----\n\n"));
MosPrintf(MIL_TEXT("A color source image was loaded on the left and color text\n"));
MosPrintf(MIL_TEXT("annotations were written in it.\n"));
MosPrintf(MIL_TEXT("Press <Enter> to continue.\n\n"));
MosGetch();

/* Convert image to Hue, Saturation, Luminance color space (HSL). */
MimConvert(MilLeftSubImage, MilRightSubImage, M_RGB_TO_HSL);

/* Create a child buffer that maps to the luminance component. */
MbufChildColor(MilRightSubImage, M_LUMINANCE, &MilLumSubImage);

/* Add an offset to the luminance component. */
MimArith(MilLumSubImage, IMAGE_LUMINANCE_OFFSET, MilLumSubImage,
        M_ADD_CONST+M_SATURATION);

```

```

/* Convert image back to Red, Green, Blue color space (RGB) for display. */
MimConvert(MilRightSubImage, MilRightSubImage, M_HSL_TO_RGB);

/* Print a message. */
MosPrintf(MIL_TEXT("Luminance was increased using color image processing.\n"));

/* Print a message. */
MosPrintf(MIL_TEXT("Press <Enter> to end.\n"));
MosGetch();

/* Release sub-images and color image buffer. */
MbufFree(MilLumSubImage);
MbufFree(MilRedBandSubImage);
MbufFree(MilGreenBandSubImage);
MbufFree(MilBlueBandSubImage);
MbufFree(MilRightSubImage);
MbufFree(MilLeftSubImage);
MbufFree(MilImage);

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

return 0;
}

```

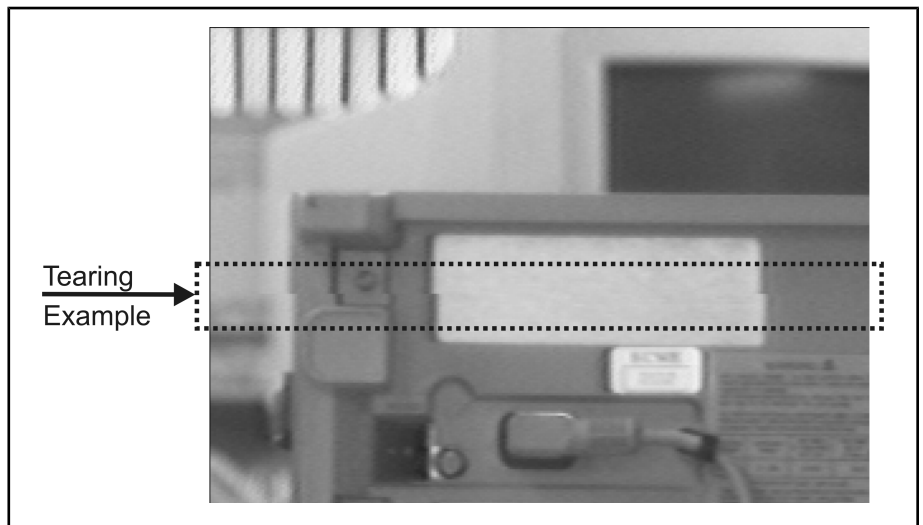
- ❖ For MIL-Lite, the luminance operation is not performed; the image is merely copied from the left child buffer to the right.

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. For auxiliary displays, if your imaging board has a display section and it is available, MIL will typically use it for display purposes.

Screen tearing

Screen tearing occurs when display memory and the screen are not updated synchronously. The updates to display memory and the screen might not be synchronized because the rate at which display memory is updated is either slower or faster than the rate at which the screen is refreshed. Alternatively, the updates might occur at the same rate but are not in phase with each other. The non-synchronous update causes two images to temporarily appear simultaneously. For example:



MIL supports displays with no tearing. Use **MdispControl()** with **M_NO_TEARING** to enable no-tearing. When using DirectDraw 7 (DirectX version 7.0), no-tearing requires special hardware, such as a Matrox Millennium G550 graphics controller; when using a non-Matrox display board, you must enable this feature using MILConfig before you can use it in MIL. If the appropriate hardware is not available and you try to enable no-tearing, MIL will permit tearing instead of generating an error. When using Direct3D (DirectX version 9.0), no special hardware is required; this technology handles no-tearing directly. However, Direct3D enables no-tearing for the entire screen; therefore, enabling no-tearing for one MIL display enables it for all displays at the same time. Standard hardware-acceleration display mode does not support no-tearing and an error is returned if you try to enable it.

To determine if the graphics controller is currently implementing a specified display with no-tearing, use **MdispInquire()** with **M_NO_TEARING_ACTIVE**.

Before you enable no-tearing, you must disable support for direct window annotations (**MdispControl()** with **M_WINDOW_ANNOTATIONS**); otherwise, an error is generated.

Note that no-tearing is not available for network displays.

No-tearing modes

When enabling no-tearing, you can set **M_NO_TEARING** to **M_ENABLE**, in which case MIL will choose the no-tearing mode which is optimal for the current situation (varies depending on such factors as the board used and CPU availability). Alternatively, you can explicitly set **M_NO_TEARING** to one of the following no-tearing modes:

- **M_BASIC**.
- **M_GRAB_CONTINUOUS_ONLY**.
- **M_ADVANCED**.

The **M_BASIC** no-tearing mode uses a basic no-tearing mechanism for all updates to the display. This mechanism compensates only when the rate at which display memory is updated is equal to or slower than the rate at which the screen is refreshed. Therefore, screen tearing might still occur when **MdispInquire()** with **M_NO_TEARING_ACTIVE** returns **M_YES**.

The **M_GRAB_CONTINUOUS_ONLY** no-tearing mode uses **M_BASIC** no-tearing mode when displaying a continuous grab. For all other display updates (that is, updates that are necessary due to modifications made to the display's overlay buffer or to the buffer selected to the display), the graphics controller does not use a no-tearing mechanism. Therefore, **MdispInquire()** with **M_NO_TEARING_ACTIVE**

returns **M_YES** only during a continuous grab, indicating that the graphics controller is updating the screen with a no-tearing mechanism. After **MdigHalt()** is called, **MdispInquire()** with **M_NO_TEARING_ACTIVE** returns **M_NO**. **M_GRAB_CONTINUOUS_ONLY** is not available when using Direct3D.

The **M_ADVANCED** no-tearing mode is similar to **M_BASIC** no-tearing mode in that all updates made to the display are done with a no-tearing mechanism. However, **M_ADVANCED** ensures that screen tearing does not occur even when display memory is updated faster than the rate at which the screen is refreshed. In this case, when more than one update to display memory occurs between two vertical synchronization pulses of the screen, **M_ADVANCED** uses multiple buffers to queue the updates. Note that during a vertical synchronization pulse, only one update is copied to the screen. By default, every update in the queue is copied to the screen; this might result in a lag in displaying updates.

When using DirectDraw 7, you can avoid the lag using the skip versions of the **M_ADVANCED** no-tearing mode. By adding **M_SKIP_OLDEST** to **M_ADVANCED**, MIL skips the oldest updates in the queue and copies only the newest update to the screen during each vertical synchronization pulse. Alternatively, by adding **M_SKIP_NEWEST** to **M_ADVANCED**, MIL skips the newest updates in the queue and copies the oldest update to the screen during each vertical synchronization pulse.

By default, no-tearing is typically disabled (**M_DISABLE**). For auxiliary displays using an encoder and displaying a continuous grab and using DirectDraw 7, no-tearing is enabled (**M_ENABLE**) by default.

Region of interest in a windowed display

It is often useful to highlight a region in a displayed image, for example, to demonstrate a calculated region or a user-selected region. With windowed displays, not only can you non-destructively outline an established region of interest (ROI) with a selected color, you can also allow a user to interactively select an ROI from the display; you can then retrieve its coordinates and act on this region (for example, to save it or create a child buffer from it). In each windowed display, you can outline/interactively select one ROI, without enabling the overlay mechanism and drawing a rectangle in the overlay buffer around the required region. The ROI is drawn directly in the display's window (using its DC); you must enable the **M_WINDOW_ANNOTATIONS** control type before you can use this feature. The ROI respects panning and zooming.

Defining an ROI in a display interactively

To interactively define an ROI in a display, you must:

1. Select the required image buffer for display, using **MdispSelect()**.
2. Enable drawing in the display's window, using **MdispControl()** with **M_WINDOW_ANNOTATIONS**.
3. Enable ROI defining mode. To do so, use **MdispControl()** with **M_ROI_DEFINE** set to **M_START**. Alternatively, if the display is presented in a default MIL window, you can click on the **DefineROI** button in the window's toolbar. Note that by default, the toolbar does not include this button and other ROI buttons; to add them, use **MdispControl()** with **M_WINDOW_ROI_BUTTONS**.
4. Position the cursor at a corner of the required ROI, and then drag the cursor to the required opposite corner; a rectangle with sizing handles appears as you drag. After releasing the mouse button, you can resize the ROI using the handles, or reposition it by clicking in its center and dragging it to the required location.

5. Disable ROI defining mode, by clicking outside of the defined region. You can also use **MdispControl()** with **M_ROI_DEFINE** set to **M_STOP**, or you can click on the **DefineROI** button again.

Once ROI defining mode is disabled, a rectangle without sizing handles appears around the selected ROI. You can hide the rectangle using **MdispControl()** with **M_ROI_SHOW** or by clicking on the **ShowROI** button in the default MIL window's toolbar.

By default, if you re-enter ROI defining mode using **M_START**, a rectangle with sizing handles appears around the last ROI defined in the display, allowing you to reposition and resize it. To start defining the ROI from scratch, add **M_RESET** to **M_START**.

In ROI defining mode, MIL intercepts and handles all Windows mouse events occurring in the default MIL window or a user-defined window of the display. Therefore, before entering ROI defining mode, you can hook a function to the ROI change event or change end event, using **MdispHookFunction()** with **M_ROI_CHANGE** or **M_ROI_CHANGE_END**, respectively.

Outlining an established region

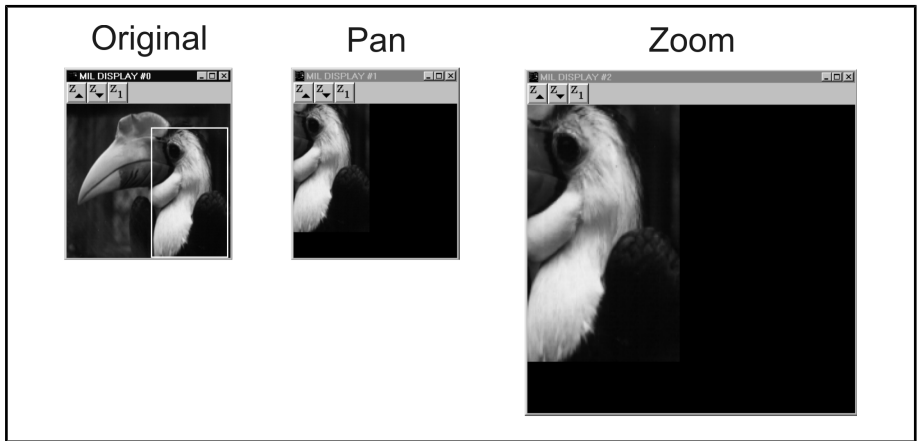
In a display, after enabling drawing in the display's window (**MdispControl()** with **M_WINDOW_ANNOTATIONS**), you can outline a region established using some calculation or analysis, using **MdispControl()** with **M_ROI_BUFFER_...** or **M_ROI_DISPLAY_...**. The **M_ROI_BUFFER_...** control types allow you to specify the position and size of the ROI in image buffer coordinates. The **M_ROI_DISPLAY_...** control types allow you to specify them in display coordinates. Note that the ROI is restricted to the image buffer area.

Note that as the image is zoomed and panned, the ROI increases in size and moves accordingly, ensuring that the area encompassed by the ROI does not change. The offset will change value in display coordinates, but will remain the same in buffer coordinates.

When using these control types, the region cannot be moved or resized interactively.

Panning 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. You can associate panning and zooming effects with the display to view specific parts of the image.



Panning and scrolling

Panning displaces an image horizontally and/or vertically on the display, from the top-left corner of the window (for windowed displays) or the screen (for auxiliary displays). Panning is also known as scrolling. To pan displayed images, use **MdispPan()** and specify the required X- and Y-panning offsets in image pixels. To center images in the display, use **MdispControl()** with **M_CENTER_DISPLAY** instead.

Zooming

Zooming replicates or subsamples the pixels of an image upon display by a specified factor. To zoom displayed images in the horizontal and/or vertical direction with floating point precision, use **MdispZoom()** with the required X- and Y-zoom factors; to reduce the size of the displayed images, specify a zoom factor less than 1. Zooming uses the interpolation mode set using **MdispControl()** with **M_INTERPOLATION_MODE**. By default, zooming uses the fastest interpolation mode.

Zoom factors

The specified zoom factors also affect the panning offsets since these offsets are specified in image pixels. For example, if the X-zoom factor is set to 4, panning by an X-offset of one image pixel results in panning by 4 pixels in the horizontal direction on the display.

Instead of explicitly specifying a zoom factor, you can automatically scale the displayed images to fit the display, using `MdispControl()` with `M_SCALE_DISPLAY` or `M_FILL_DISPLAY`. `M_SCALE_DISPLAY` maintains the image aspect ratio, whereas `M_FILL_DISPLAY` doesn't. Both these control types override the explicitly specified zoom factors (`MdispZoom()`), disable the window's zoom buttons (`M_WINDOW_ZOOM`), and override pan settings (`MdispPan()`).

Note that `MdispPan()` and `MdispZoom()` are not available for network displays.

Zooming example

For an example of zooming, refer to *MImConvolve.cpp*, where the display is zoomed by a factor of 2 to better demonstrate the result of an image processing operation (spatial filtering operation).

Annotating the displayed image non-destructively

Annotating images using the overlay-display mechanism

For all types of 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_OVERLAY, 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**. The overlay buffer can be used to annotate the underlying image with an effect called transparency, or keying. For more information, see Transparency (Keying).

3. To access the display's overlay buffer, use **MdispInquire()** with **M_OVERLAY_ID** to determine the MIL identifier of the buffer:

```
MdispInquire(DisplayID, M_OVERLAY_ID, &OverlayBufferID);
```

4. Annotate the display's overlay buffer as required. For example, to write text in the overlay buffer, you can use **MgraText()**. Note that since this temporary overlay buffer is a real image buffer, any function (except grabbing) can be used.

You can also annotate the displayed image buffer or the overlay buffer with Windows GDI annotations. For information, see the *Using GDI annotations* subsection in the *Annotating the displayed image nondestructively* section in *Chapter 20: Displaying an image*.

The overlay buffer will have the same size as its associated image (not the size of the display). In addition, it will have the same number of bands and depth as the desktop for windowed displays or as the VCF for auxiliary displays.

Overlay buffer behavior

When an image is selected to a display that has an associated overlay buffer, and you select another image to that display, one of the following occurs:

- If the new image has the same format and size 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()**.
- If the new image has a different format or size than the image currently selected to that display, the current overlay buffer is freed, and another overlay buffer is created. The new overlay buffer has the same size as the image selected to the display, and the annotations of the old overlay buffer are copied into the new one.

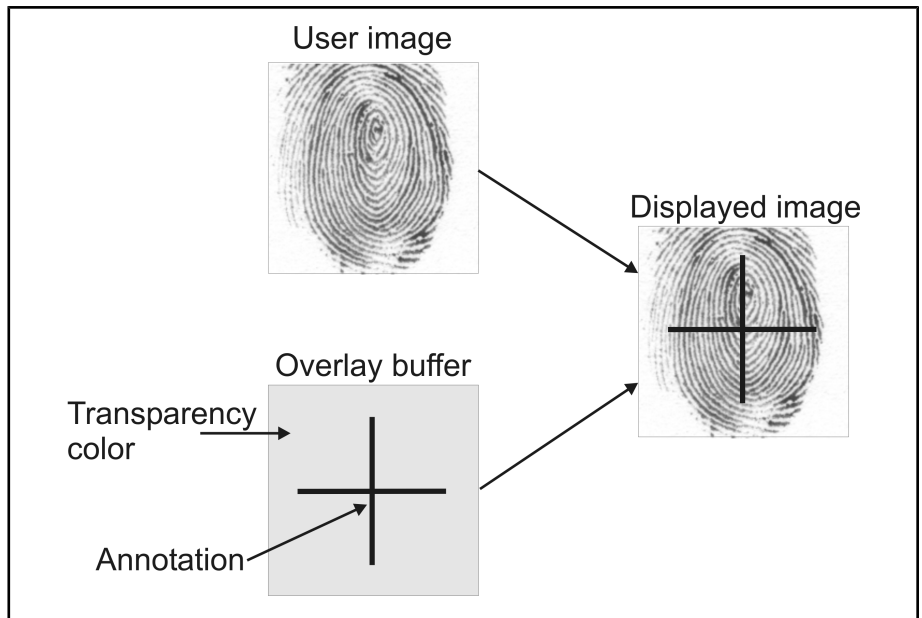
The identifier of the old overlay buffer is now obsolete. To inquire the identifier of this newly created overlay buffer, use **MdispInquire()** with the **M_OVERLAY_ID** inquire type.

CPU-assisted overlay

The ability to annotate the displayed image non-destructively using MIL's overlay-display mechanism is always available. Depending on the hardware-acceleration display mode, MIL might produce the overlay-effect using your hardware (that is, your graphics controller) or using the CPU. In the latter case, the display is said to be CPU-assisted.

Transparency (Keying)

The display's overlay buffer annotates the image selected to the display using transparency. Transparency, also known as keying, is a mechanism that replaces pixels of an image that are of the specified transparency (keying) color with the underlying areas of another image. Using transparency, annotations made to the overlay buffer in a color other than the transparency color will annotate the image selected to the display.



When allocating a display (**MdispAlloc()**), the transparency color is automatically set to a default color, which is generally appropriate. Use **MdispInquire()** with **M_TRANSPARENT_COLOR** to determine the transparency color. If required, select another transparency color using **MdispControl()** with **M_TRANSPARENT_COLOR**.

If you are using an 8-bit display resolution (256 colors), set the transparency 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:

```
MdispControl(DisplayID, M_TRANSPARENT_COLOR, M_RGB888(20,32,24));
```

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

Using GDI annotations

Besides using MIL functions (for example, `MgraText`) to annotate your display, you can use GDI annotations. There are two ways to do so.

The first technique allows you to draw GDI annotations in the buffer selected to the display or in the display's overlay buffer. To perform this technique:

1. Allocate a device context (DC) for either the image buffer or the overlay buffer of the display using **MbufControl()** with **M_DC_ALLOC**. Windows GDI annotation functions require a DC to draw. You cannot allocate a DC for an image buffer that is not associated with a display. If you draw using the DC of the image buffer selected to the display, drawing will be destructive (that is, the data of the image buffer is actually changed).
 - ❖ The buffer in which you draw must be GDI compatible; otherwise, an error is reported. To force the buffer selected to the display to be GDI compatible, allocate it using **MbufAlloc...** with **M_GDI**; to force the overlay buffer to be GDI compatible, allocate the display using **MdispAlloc()** with **M_GDI_OVERLAY**. You cannot force the overlay buffer of a network display to be GDI compatible.
2. Inquire the identifier of the device context (DC) created in the previous step using **MbufInquire()** with the **M_DC_HANDLE** inquire type.
3. Paint your annotations using Windows GDI annotation functions with the DC.

4. Free the device context (DC), using **MbufControl()** with **M_DC_FREE**.
5. Notify MIL that the display needs to be updated with the GDI annotations, using **MbufControl()** with **M_MODIFIED**.

When drawing GDI annotations using this technique, the main advantages are that you draw using the buffer's coordinates, and the annotations will follow the underlying image when the display is zoomed or panned. The drawback is that the size of the overlay buffer is limited to the size of the selected image buffer. It is not possible to draw outside those limits.

The following portion of MIL code shows the creation of the device context for the overlay buffer, the inquiring of the device context, and the drawing and writing in the overlay buffer (see also, *MDispOverlay.cpp*).

```
HDC hCustomDC;
HGDIOBJ hpen, hpenOld;
MIL_TEXT_CHAR chText[80];

/* Create a device context for drawing. */
MbufControl(MilOverlayImage, M_DC_ALLOC, M_DEFAULT);
/* Inquire the handle of the device context. */
hCustomDC = ((HDC)MbufInquire(MilOverlayImage, M_DC_HANDLE, 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. */
    MosStrcpy(chText, 80, MIL_TEXT("GDI Overlay Text "));
    SetTextColor(hCustomDC,RGB(0, 0, 255));
    TextOut(hCustomDC,ImageWidth*3/18,ImageHeight*4/6, chText, (int)MosStrlen(chText));
    SetTextColor(hCustomDC,RGB(255, 0, 0));
    TextOut(hCustomDC,ImageWidth*12/18,ImageHeight*4/6, chText, (int)MosStrlen(chText));
}
```

```

/* Deselect and destroy the blue pen. */
SelectObject(hCustomDC,hpenOld);
DeleteObject(hpen);
}

/* Free the created device context. */
MbufControl(MilOverlayImage, M_DC_FREE, M_DEFAULT);

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

```

The second technique, which is only supported for windowed displays in user-defined windows (**MdispSelectWindow()**), allows you to draw GDI annotations directly in the display's window. To perform this technique:

1. Enable drawing directly into the display's window using **MdispControl()** with **M_WINDOW_ANNOTATIONS**.
2. In the window procedure (**WindowProc** function) of your user-defined window:
 - a. Pass the window handle to the Windows **GetDC** function to get a Windows display device context (DC).
 - b. Then, paint the annotations with GDI functions each time the window receives a paint message (for example, **WM_PAINT**).
 - c. Release the DC using the Windows **ReleaseDC** function.

When drawing GDI annotations using this technique, you draw directly in display's window, so you use window coordinates. You can draw over the image, as well as outside the image area. MIL is unaware of these annotations, so if you zoom or pan the display, the annotations are not affected. Note that when a repaint of the window is needed, MIL will first clear and then redraw the image area; then, MIL will call your **WindowProc** function to redraw the window's annotations. This might result in visible flickering.

Mapping 1-band images through a LUT upon display

Upon display, you can map 1-band images through a specified LUT to control their gray levels or to display them in color; the LUT maps the image pixels to precalculated values on display. The following are some situations for which you can use a LUT to produce required display effects:

- 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.

Selecting the LUT to use for display

Upon display, you can map all 1-band image buffers selected to a specified display through a specified LUT, or you can map only a specified 1-band image buffer through a LUT. To do so:

1. Allocate an 8-bit LUT buffer using **MbufAlloc1d()** or **MbufAllocColor()** with **M_LUT**. The LUT buffer must have $2^{\text{depth of image buffer(s) to map}}$ number of entries (for example, to map 8-bit images, the LUT buffer should have 256 entries).
2. Generate the data into the buffer using **MgenLutRamp()**, or load the data into it, using **MbufPut()**.
3. Associate the LUT buffer with the required display using **MdispLut()**, or to a particular image buffer using **MbufControl()** with **M_ASSOCIATED_LUT**. In the former case, all image buffers selected to the specified display are mapped through the LUT. In the latter case, only the specified image buffer is mapped and when the image buffer is saved, the LUT data is saved with it.

Depending on the required display effect, associate either a 1-band or 3-band LUT buffer with the display or image buffer. For example, to invert the values of an image on display, use a 1-band **M_LUT** buffer that maps each pixel to the maximum pixel value minus the current pixel value; you can use **MgenLutRamp()**

to generate the LUT data for such a mapping. To view a 1-band image buffer with each gray intensity in a different color, use a 3-band **M_LUT** buffer; to achieve this effect, you can also use the **MdispLut()** predefined 3-band pseudo-color LUT buffer, **M_PSEUDO**, instead of allocating your own LUT buffer.

If both the display and the selected image buffer have an associated LUT buffer, the one associated with the display is respected.

To disassociate a LUT buffer from a display or image buffer, use **MdispLut()** with **M_DEFAULT**, or **MbufControl()** with **M_ASSOCIATED_LUT** and **M_DEFAULT**, respectively.

Restrictions when displaying using LUT

When displaying using a LUT, note the following:

- If the content of a LUT buffer changes while the image is selected on the display, the changes will not take effect until you call **MdispLut()** again.
- You cannot associate a 3-band image buffer with a LUT buffer, nor can you select a 3-band image buffer to a display that is associated with a LUT buffer. In addition, the image buffer must be an 8- or 16-bit buffer. If these conditions are not respected, an error is generated.
- The view mode of the display cannot be set to **M_AUTO_SCALE** or **M_MULTI_BYTES**.
- 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 map an 8-bit image buffer (that is, an image that can have 256 intensities), your LUT buffer must also have 256 entries.
- When using **MdispLut()** with a network display, only **M_PSEUDO** and **M_DEFAULT** are supported.

Advanced display concepts

MIL uses different mechanisms to overcome any hardware limitations and produce an artifact-free display at the fastest possible rate. Although in general you do not need to know how the display is achieved, an understanding can sometimes prove useful.

Display architectures

To display images, MIL uses on-screen and off-screen display memory and/or Host memory, depending on the MIL display architecture used. MIL automatically switches between four different display architectures, depending on which is most appropriate. Each architecture imposes its own restrictions and advantages. The display architectures are presented in the table below in order of performance from left to right:

Feature	Display architecture			
	Underlay surface live	Underlay surface non-live	GPU display	CPU display
Standard hardware acceleration mode	No	No	No	Yes
DirectDraw 7 (DirectX version 7)	Yes	Yes	Yes	Yes
Direct3D (DirectX version 9)	No	No	Yes	Yes
Supports non-Matrox display board	No	Depends	Yes	Yes
Supports screen capture	No	No	Yes	Yes
Supports remote desktop utilities	No	No	No	Yes
Can span display across multiple screens	No	No	No	Yes
Supports displaying image buffers of all sizes and color depths	No	No	No	Yes

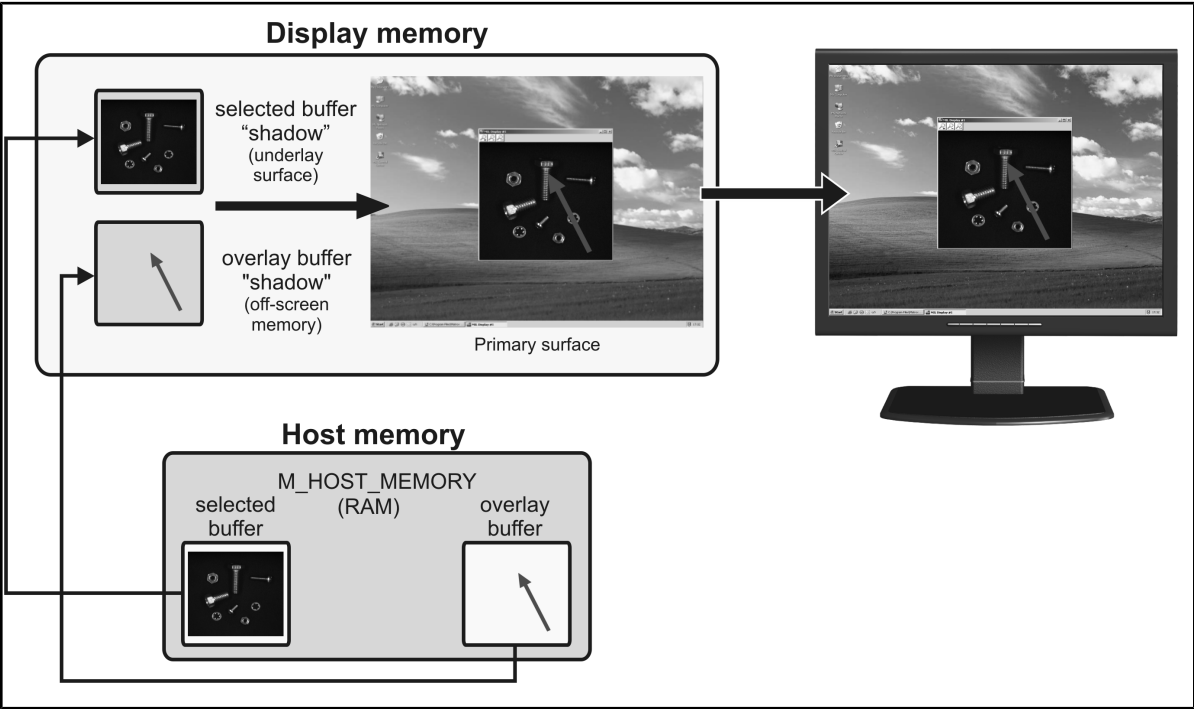
A visible difference between the display architectures is very rare.

Underlay surface live architecture

The fastest MIL display architecture is the underlay surface live architecture, available for use with DirectDraw 7. This architecture makes use of the hardware ability of all Matrox display boards and many third-party display boards to have two on-screen surfaces per video output controller: the primary surface and an underlay surface. The primary surface is the surface from which the desktop is displayed. The underlay surface is a surface that DirectDraw can allocate, link to

a region on the primary surface (for example, a window), and make visible in areas of the region that are set to the transparency color. When using this architecture, MIL makes use of special Matrox display hardware that automatically triggers a screen refresh when the underlay surface is modified.

When using this architecture, MIL fills the region of the display in the primary surface with a copy (shadow) of the overlay buffer (if enabled) or with the transparency color; it fills the underlay surface with a copy of the buffer selected to the display. When performing a continuous grab, the data is grabbed into the underlay surface without Host intervention. DirectDraw merges the displayed image with the display's overlay buffer. The Host processor is not used.



This architecture is only available on Matrox display boards. Depending on the selected display resolution, an underlay surface might not be supported, in which case MIL cannot use this architecture. Depending on the supported underlay surface sizes and depths, there might be restrictions on the size and color depth of the buffer that can be selected to the display. There might also be some restrictions on the zoom factors and pan offsets supported. In addition, only one display per screen can use this architecture and the display cannot span across multiple screens. Finally, screen capture and remote desktop utilities are not supported.

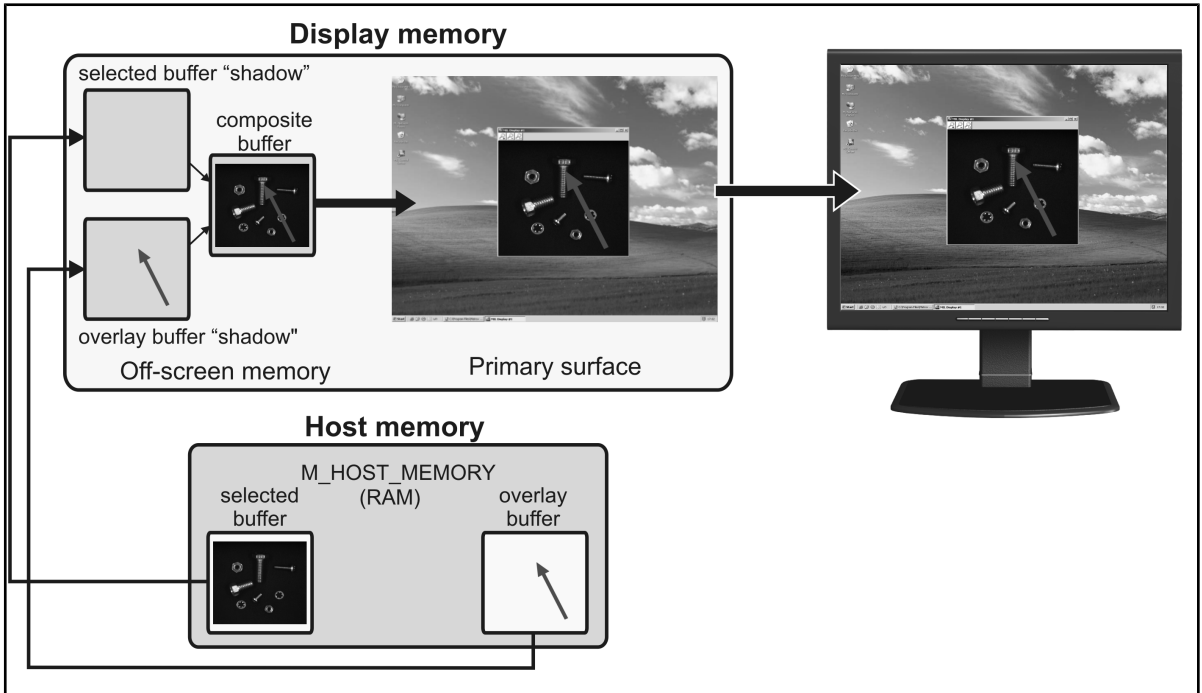
Underlay surface non-live architecture

The underlay surface non-live architecture is almost identical to the underlay surface live architecture in implementation, performance, and restrictions, and is available for use with DirectDraw 7. The only difference is that it can also be used with third-party display boards that support an underlay surface. To support these boards, the non-live architecture uses the Host to issue a screen refresh request after display updates; this involves very little Host processing power.

For third-party display boards, you must use MILConfig to enable the architecture before MIL can use it. Once enabled, the buffer selected for display is presented from the underlay surface and automatically merged with the display's overlay using DirectDraw. For these boards, you must also explicitly enable, using MILConfig, the ability to grab continuously into the underlay surface; if this second option is not enabled, MIL will grab in an intermediate temporary buffer in Host non-paged memory and then copy the grabbed image into the underlay surface. You should leave this second option disabled if the display memory used to implement the underlay surface is not contiguous or its addresses are not constant. If unsure of the underlay surface implementation, you can try enabling the option and see if there are any problems with the display when grabbing continuously (it could remain black or even crash your application).

GPU display architecture

The third high-performance architecture available is the graphical processing unit (GPU) video architecture, available for use with DirectDraw 7 and Direct3D. When using this architecture, MIL maintains in off-screen display memory a copy (shadow) of both the buffer selected for display and the overlay buffer. The GPU is used for the composition and final blit of the displayed image. Very little Host processing power is used.

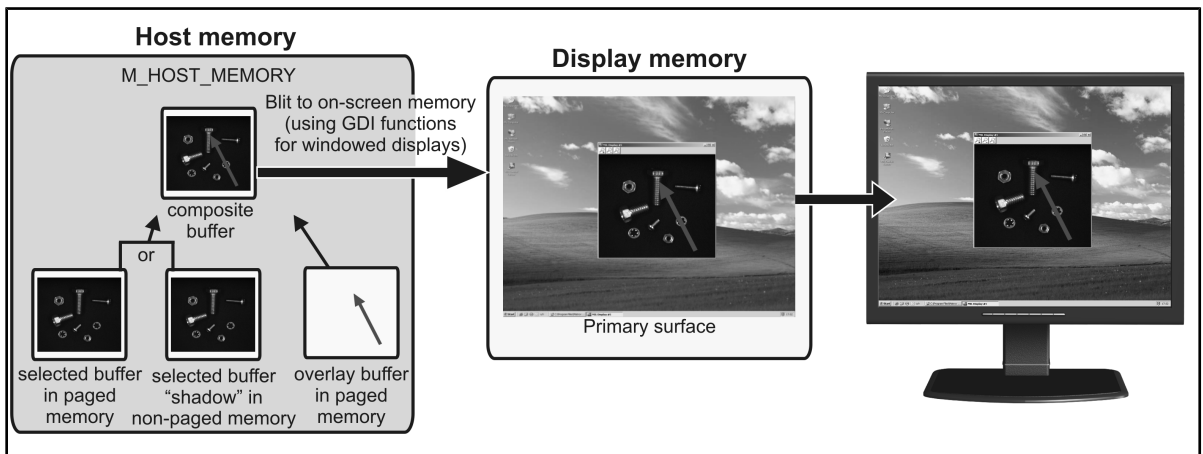


In DirectDraw 7, when performing a continuous grab and using a Matrox display board, the data is grabbed into off-screen memory without Host intervention (if the display memory is physically accessible from the acquisition board). If you want to grab continuously into off-screen memory when using a third-party display board, you must explicitly enable this ability using MILConfig. If this option is not enabled, MIL will grab in an intermediate temporary buffer in Host non-paged memory and then copy the grabbed image into off-screen memory. You should leave this option disabled if off-screen display memory is not contiguous or its addresses are not constant. If not sure, you can try enabling the option and see if there are any problems with the display when grabbing continuously (it could remain black or even crash your application).

To be able to use the GPU display architecture, the size and color depth of the buffer selected to the display might have to meet certain restrictions, based on the GPU and the amount of display memory available. Also, the display cannot span across multiple screens. Since the underlay surface is not used, MIL can use this architecture when screen capture functions are used. However, when remote desktop utilities are used, MIL cannot use this architecture because it accesses and uses display memory directly.

CPU display architecture

The CPU display architecture is the most flexible. The performance of the CPU display architecture is dependent on the speed of the Host CPU, and is available for use with DirectDraw 7, Direct3D, and standard hardware acceleration display mode. The CPU display architecture uses the Host CPU and memory for all display operations, using Windows GDI functions. When performing a continuous grab, the data is grabbed in an intermediate temporary buffer in Host non-paged memory. There are no restrictions on the size and color depth of the buffer that can be selected to the display. Moreover, screen capture and remote desktop utilities are fully supported. Owing to its flexibility, MIL often uses this display architecture, especially when other architectures are not available.



On older computers, when the overlay-display mechanism is enabled and using this architecture, you might notice overlay flickering and a slight lag in the display of a continuous grab. 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.

CPU-assisted displays always use the CPU display architecture.

Selecting an architecture

When displaying an image, MIL transparently selects the best available architecture. MIL dynamically changes architecture at each occurrence of a selection-influencing event. The following are factors that could influence the architecture selection:

- Hardware acceleration mode.
- Buffer format and size.
- Presence of a LUT.
- View mode.
- Overlay usage.
- Hardware limitations.
- Continuous grab settings.
- Screen-tearing settings.
- Number of displays allocated and their uses.
- Pan and zoom values.
- Display position in the desktop (windowed display).
- Running a full-screen application, such as a screen saver.

- Selecting a new buffer on a display.
- Moving or resizing a window.

If the automatically selected architecture is causing display artifacts, you can force the use of the CPU display architecture under required circumstances. To do so, select a specific hardware-acceleration mode using MILConfig. If you select the standard hardware-acceleration mode, MIL will only use the CPU display architecture. If you select the DirectDraw 7 or Direct3D hardware acceleration mode, you can use the presented slider to disable the acceleration of certain features in hardware; in this case, these features will always be implemented using the CPU display architecture. Note that disabling the acceleration of certain features might also disable the acceleration of other features.

For older computers experiencing display artifacts when using the CPU display architecture, try the following:

- Under Windows, switch to DirectDraw 7.
- Lower your screen resolution or refresh rate.
- Free all video memory buffers or displays that are no longer being used.
- Avoid forcing a specific internal format when allocating the image buffer, unless it is required.

Video output controllers

Except when using the CPU display architecture, MIL allocates and accesses on-screen (primary and underlay) and off-screen display memory directly, using the DirectDraw 7 or Direct3D API. All surfaces in display memory (on-screen and off screen surfaces) are tightly linked to a specific video output controller of the graphics controller. Unfortunately, it is possible to lose the connection to the video output controller when, for example, the desktop resolution is changed, a monitor is added to or removed from the desktop, a screen-saver or another full-screen mode program is started, or a remote connection is initiated. When the connection is lost, all buffers allocated in display memory are lost, the underlay surface becomes unusable, and MIL must switch to the CPU display architecture. Once the connection is re-established, MIL will switch back to the best available architecture.

Remote desktop

Remote desktop utilities allow you to connect remotely to a computer and control it as if you were seated in front of it. When you use a remote desktop utility to access a remote computer, it loses the connection to its video output controllers, for a period that depends on the remote mode used:

- **Remote assistance mode.** In this mode, the remote computer requests help from you, and allows you to take control of its desktop. When the connection takes place, the remote computer temporarily loses the connection to its video output controllers, but the connection is restored immediately thereafter.
- **Remote connection mode.** In this mode, you initiate the connection to the remote computer. The screen of the remote computer is locked, and the remote computer loses the connection to its video output controllers. From the point of view of the running processes on the remote computer, all the display boards appear to have been disabled and if you view the display properties of the remote computer, its display boards are not listed. Instead, a new virtual display board is listed.

If a MIL application is running on the remote computer, its MIL displays recognize the "Video Device Lost" event and react to it by rescanning the available hardware (real and simulated) and switching to the CPU architecture. When the connection to the video-output controllers is re-established, the MIL displays will detect the "Video Device Restored" event and attempt to reuse the hardware directly.

Continuous grab

Unlike `MdigGrab()` and `MdigProcess()`, `MdigGrabContinuous()` is designed solely for display purposes. When the grab buffer is selected to a display, `MdigGrabContinuous()` will typically bypass the specified buffer, and grab into an intermediate temporary buffer (in display or Host non-paged memory) to minimize CPU usage and improve performance. Only the last frame grabbed is written into the selected buffer.

Overlay

The overlay buffer has the same size as the selected buffer, and the same number of bands and depth as the desktop for windowed displays or as the Video Configuration Format (VCF) for auxiliary displays. For performance reasons, the overlay buffer is typically allocated in Host memory, except when using Matrox Odyssey; in which case, it is allocated on-board for the same reason.

When you need to draw in the overlay buffer and MIL's drawing functions (**Mgra...** and the drawing functions of MIL's analysis modules) are not sufficient, you can usually request the DC (device context) of the overlay buffer and use Windows GDI (Graphics device Interface) functions in it. Since the overlay buffer is allocated on-board for Matrox Odyssey, it will not be possible to get a Windows DC. There are two ways to get around this:

- Allocate an **M_HOST_MEMORY** buffer that has the same size as the overlay buffer, use the GDI functions in it, and then copy it to the overlay buffer using **MbufCopy()**.
- Allocate the display using **MdispAlloc()** with **M_GDI_OVERLAY** to force the allocation of the overlay buffer in Host memory. Note that, if in Host memory, writing to the overlay buffer with MIL functions becomes slower on Matrox Odyssey.

Large buffers

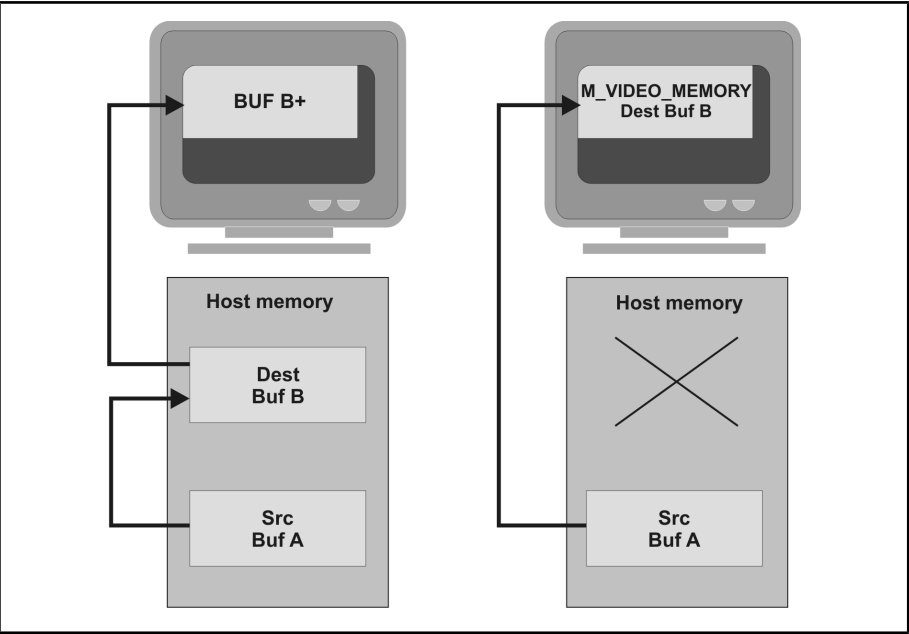
When using a 32-bit operating system, processes are usually limited to 2 Gbytes of virtual address space. However, this address space is typically fragmented because it is used, for example, by the DLLs, the stack, and the heap. This makes it hard to allocate very large buffers because they have to be mapped in this address space, and it rapidly becomes impossible to allocate all the required temporary buffers at the size of the buffer selected to the display.

To save memory when the selected buffer is larger than 25 Mbytes, MIL will automatically switch to a mode where all temporary buffers have the size of the visible part of the selected buffer; this part is usually much smaller than the whole buffer. This mechanism minimizes the amount of address space required to display a large image; however, it will make some operations, like window resizing, slower. This mode is triggered automatically and you should not notice it being used.

If the buffer is too large to display and MIL cannot allocate the required internal buffers, an error is reported.

Optimizing auxiliary displays

When allocating a displayable buffer for any auxiliary display, overriding the default buffer allocation sequence is useful. If you are not using the displayable buffer for processing or are only using it as a destination, storing the buffer only in display/non-paged memory (**M_VIDEO_MEMORY**) will avoid the extra copy operation to the display, without the penalty of slowing down processing.



Internal format of the buffer

Even if the buffer is not in the on-screen area of display memory, the image buffer depth and the display depth (determined by the **.vcf*) should be the same to optimize performance.

Traces and user traces

MIL displays integrate an extended logging mechanism. The mechanism traces every call related to each display, and contains a lot of information regarding the computer and the running application. If you experience a display problem that customer support can't resolve, they will typically instruct you to use MILConfig to enable the logging mechanism. Then, they will ask you to run your MIL application; this will generate a log file. You should then send them this log file. The file is not readable by a user and must be sent to Matrox Imaging for analysis.

Chapter

21

Generating graphics

This chapter describes the graphics functions that are available with MIL. These consist of drawing and text-writing functions.

MIL and graphics in general

The MIL package supports basic drawing and text functions that are useful in typical image processing or machine vision applications. These functions could be used, for example, to create a condition 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()** function 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:

- **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()**.
- **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.

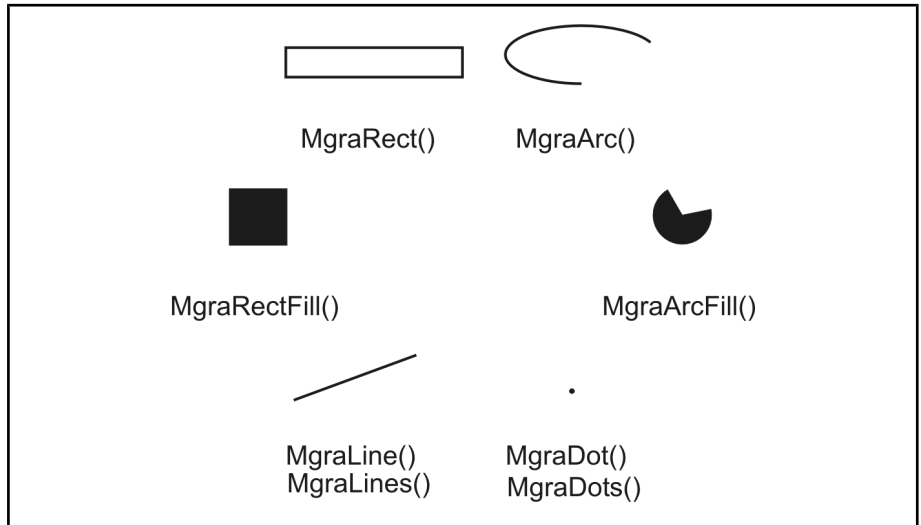
Drawing graphics

With the MIL package, you can draw:

- Lines (**MgraLine()**, **MgraLines()**).
- Rectangles (**MgraRect()** and **MgraRectFill()**).
- Arcs, circles, and ellipses (**MgraArc()** and **MgraArcFill()**).
- Dots (**MgraDot()**, **MgraDots()**).

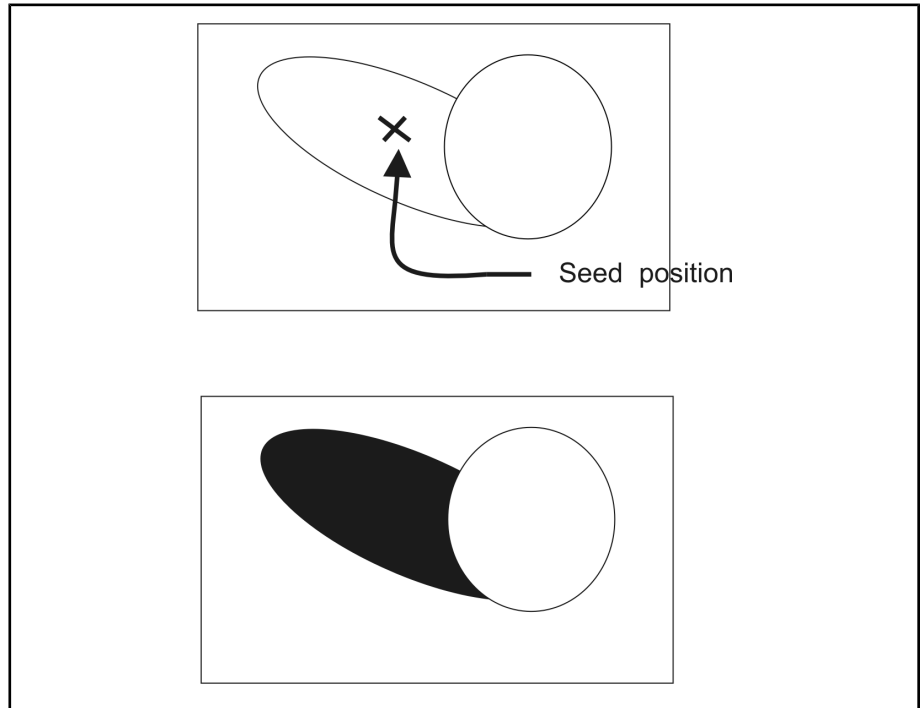
Using **MgraLine()**, **MgraRect()**, **MgraArc()**, or **MgraDot()**, you can draw the outline of most required shapes. The outlines are drawn one pixel wide. You can also call **MgraLines()** and **MgraDots()** to draw multiple lines and dots.

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



Filling shapes

If you need complex filled-in shapes, draw the outline of the shape and use **MgraFill()** to fill it. **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.



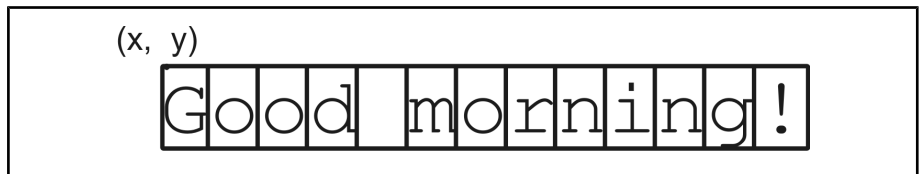
Note, any drawing is clipped outside the boundaries of the buffer.

Note, the **Mgra...** functions use the image coordinate system (pixel coordinates). To draw in a calibrated image using real-world coordinates, you must first convert the coordinates from their real-world value to their pixel value using **McalTransformCoordinate()** or **McalTransformCoordinateList()**.

Writing text

You can also write text in the drawing area, using **MgraText()**. This function 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

22

Grabbing with your digitizer

This chapter discusses how to grab images using a digitizer allocated on your frame grabber and the features available when grabbing.

Cameras and video sources

MIL supports input from any type of video source supported by the imaging board (frame grabber). Each frame grabber has one or more available acquisition paths. MIL refers to the acquisition paths used to grab from one video source as a digitizer.

- ❖ Note, since most video sources 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 data input channel. Once you have finished using a digitizer, you should free it, using **MdigFree()**.

To make a more portable application, you can allocate and use the default system and digitizer set up when installing MIL or using MILConfig. To allocate the default system and digitizer, use **MappAllocDefault()**. Alternatively, you can use **MsysAlloc()** with **M_SYSTEM_DEFAULT** to allocate the default system. Use the returned system identifier with **MdigAlloc()** and set the device number and data format both to **M_DEFAULT** to allocate the default digitizer.

Use **MdigGrab()** or **MdigGrabContinuous()** to grab data from a camera through a digitizer. The data is stored in an image buffer. The image buffer should have as many bands as the grabbed data has color components. If you need to grab and process concurrently, you can use **MdigProcess()**.

Both the image buffer and digitizer must be allocated on the same system.

When developing an application, if you are experiencing difficulty with trigger and exposure timing, it is recommended that you work in continuous grab mode. Once the application is working, you can switch to a more sophisticated camera, if necessary. This approach makes debugging much easier.

Basic concepts

The basic concepts and vocabulary conventions for the MIL digitizers are:

- **Acquisition path.** A path that has the components to digitize or capture a video input signal.
- **Camera.** A generic term to refer to any physical device used for creating image/video data and that can be connected to a frame grabber.
- **Frame grabber.** A type of imaging board which can acquire video data from a camera.
- **Digitizer.** The acquisition path(s) with which to grab from one camera of the specified type. When several MIL digitizers are allocated, their device number, along with their DCE, identify if they represent the same path(s) (but perhaps for a different input format) or independent path(s) for simultaneous acquisition.
- **Data input channel (channel).** Identifies which camera to use when several of its type can be connected to the same acquisition path(s) (for example, grab from channel 0 or channel 1 of digitizer 0).
- **Independent acquisition path.** An acquisition path that can, if required, acquire data from an input source independently from another such path on the same frame grabber. Each independent acquisition path has its own digitizing components to manage all video timing and synchronization for the path.
- **Imaging board.** A generic term to refer to all boards that manipulate video data, including: frame grabbers, vision processors, compression boards, and other specialized boards.
- **Timing control unit.** A hardware unit (for example, a Programmable Synchronization Generator or PSG) that is responsible for managing all video timing and synchronization signals.
- **Vision processor.** A type of imaging board which can process and analyze image/video data. A vision processor includes a CPU and additional memory.

Data format

MdigAlloc() needs the digitizer configuration format (DCF) that corresponds to your camera to perform the digitizer allocation. The DCF defines such settings as the input frequency and resolution, and will determine limits when grabbing an image. Once a digitizer has been allocated, many of the settings are established based on the selected DCF. You can change these settings to values other than the default using **MdigControl()** and other **Mdig...** functions. You can use **MdigInquire()** to inquire about a digitizer's settings.

MIL provides a number of predefined DCFs for the basic cameras supported by your frame grabber. Refer to the MIL Reference with regards to the predefined DCFs available for your particular board. MIL also provides some DCF files that you can load if the predefined DCFs do not suit your needs.

If you find a DCF file that is appropriate for your camera, but you need to adjust some of the more common settings, you can do so directly, without adjusting the file, using the **Mdig...** functions. For more specialized adjustments, you can adjust the DCF file itself, using Matrox Intellicam.

If you cannot find an appropriate DCF file because, perhaps, you have a non-standard camera or other video source, 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 frame grabber supports the camera.

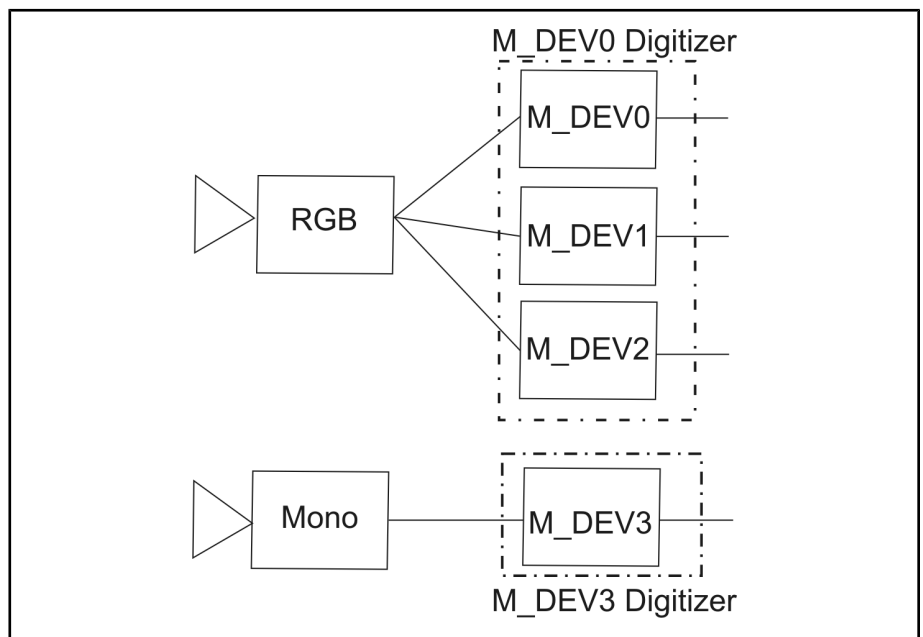
Device number

In addition to the data format, **MdigAlloc()** requires that you specify the device number of the first acquisition path of the digitizer. Its digitizer configuration format (DCF) establishes the total number of acquisition paths used. MIL associates a timing control unit (for example, a PSG) on your frame grabber, with these acquisition paths when grabbing. If a timing control unit or the specified acquisition paths are not available when a grab is requested, the grab request must wait until they become available.

When the Matrox frame grabber has only one timing control unit, you must specify **M_DEV0** or **M_DEFAULT** as the device number for the digitizer, regardless of the number of acquisition paths available.

If a digitizer can grab while another digitizer on the same frame grabber is simultaneously grabbing, it is said to be independent. If it cannot, it is dependent. The maximum number of digitizers from which you can grab simultaneously equals the number of timing control units on your Matrox frame grabber.

In order for the two digitizers depicted below to grab simultaneously, there must be at least two timing control units on your Matrox frame grabber. In the following image, the first camera is a component RGB camera and the second one is a monochrome camera. They are connected to their associated acquisition paths on a frame grabber.



For more information regarding the number of available timing control units, refer to the Installation and Hardware Reference manual and the MIL Board-specific Notes for your Matrox frame grabber.

Multiple cameras

MIL supports applications that require input from different cameras. With most Matrox frame grabbers, you can connect multiple cameras. If each camera connects to an independent acquisition path, then you can allocate a series of independent MIL digitizers and grab from these cameras simultaneously.

Simultaneous acquisition

Simultaneous acquisition is available on Matrox frame grabbers that support multiple independent acquisition paths. To grab from several sources simultaneously, you must allocate multiple digitizers with **MdigAlloc()** and then call **MdigGrab()**, **MdigGrabContinuous()**, or **MdigProcess()** with the identifiers of the required digitizers. When allocating a digitizer, you are committing one or more acquisition paths to a specific camera, depending on the format of the camera (for example, an RGB color camera uses three acquisition paths). To perform simultaneous acquisition, your digitizers must use different acquisition paths. For more information, see the *Device number* section earlier in this chapter.

Refer to the Board Specific Notes for more information regarding the number of cameras from which you can acquire data simultaneously using your Matrox frame grabber.

The following example shows you how to allocate two digitizers on a system and use them to perform two continuous grabs simultaneously.

```

/*****
/*
 * File name: MDigGrabMultiple.cpp
 *
 * Synopsis: This example performs 2 continuous grabs using the 2 digitizers
 *           of a system.
 *
 */

/* Headers. */
#include <mil.h>

/* Grab scale. */
#define GRAB_SCALE 1.0

/* Main function. */
int MosMain(void)

```



```

{
MIL_ID   MilApplication;
MIL_ID   MilSystem;
MIL_ID   MilDigitizer[2];
MIL_ID   MilDisplay[2];
MIL_ID   MilImageDisp[2];

/* Allocations. */
MappAlloc(M_DEFAULT, &MilApplication);
MsysAlloc(MIL_TEXT("M_DEFAULT"), M_DEFAULT, M_DEFAULT, &MilSystem);
MdigAlloc(MilSystem, M_DEV0, MIL_TEXT("M_DEFAULT"), M_DEFAULT, &MilDigitizer[0]);
MdigAlloc(MilSystem, M_DEV1, MIL_TEXT("M_DEFAULT"), M_DEFAULT, &MilDigitizer[1]);
MdispAlloc(MilSystem, M_DEFAULT, MIL_TEXT("M_DEFAULT"), M_DEFAULT, &MilDisplay[0]);
MdispAlloc(MilSystem, M_DEFAULT, MIL_TEXT("M_DEFAULT"), M_DEFAULT, &MilDisplay[1]);

/* Allocate 2 display buffers and clear them. */
MbufAlloc2d(MilSystem,
             (MIL_INT)(MdigInquire(MilDigitizer[0], M_SIZE_X, M_NULL)*GRAB_SCALE),
             (MIL_INT)(MdigInquire(MilDigitizer[0], M_SIZE_Y, M_NULL)*GRAB_SCALE),
             8L+M_UNSIGNED,
             M_IMAGE+M_GRAB+M_PROC+M_DISP, &MilImageDisp[0]);
MbufClear(MilImageDisp[0], 0x0);
MbufAlloc2d(MilSystem,
             (MIL_INT)(MdigInquire(MilDigitizer[1], M_SIZE_X, M_NULL)*GRAB_SCALE),
             (MIL_INT)(MdigInquire(MilDigitizer[1], M_SIZE_Y, M_NULL)*GRAB_SCALE),
             8L+M_UNSIGNED,
             M_IMAGE+M_GRAB+M_PROC+M_DISP, &MilImageDisp[1]);
MbufClear(MilImageDisp[1], 0x80);

/* Display the buffers. */
MdispSelect(MilDisplay[0], MilImageDisp[0]);
MdispSelect(MilDisplay[1], MilImageDisp[1]);

/* Grab continuously on displays at the specified scale. */
MdigControl(MilDigitizer[0], M_GRAB_SCALE, GRAB_SCALE);
MdigGrabContinuous(MilDigitizer[0], MilImageDisp[0]);
MdigControl(MilDigitizer[1], M_GRAB_SCALE, GRAB_SCALE);
MdigGrabContinuous(MilDigitizer[1], MilImageDisp[1]);

/* Print a message. */
MosPrintf(MIL_TEXT("Press <Enter> to stop continuous grab.\n"));
MosGetch();

/* Halt continuous grab. */
MdigHalt(MilDigitizer[0]);
MdigHalt(MilDigitizer[1]);

/* Print a message. */
MosPrintf(MIL_TEXT("Press <Enter> to end.\n"));
MosGetch();

/* Free allocations. */
MbufFree(MilImageDisp[0]);

```

```

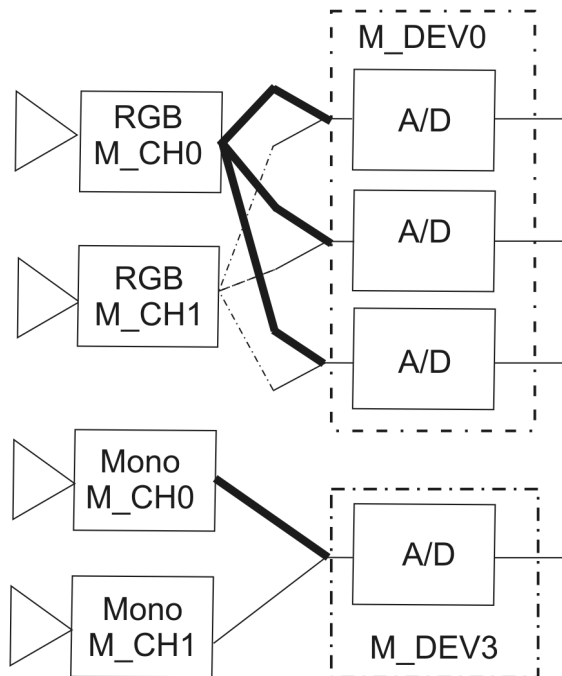
MbufFree(MilImageDisp[1]);
MdispFree(MilDisplay[0]);
MdispFree(MilDisplay[1]);
MdigFree(MilDigitizer[0]);
MdigFree(MilDigitizer[1]);
MsysFree(MilSystem);
MappFree(MilApplication);

return 0;
}

```

Data input channels of acquisition paths

Most acquisition paths have several multiplexed data input channels. This means that they have several data input channels but can only grab from one channel at a time. In this case, the MIL digitizer representing the acquisition path(s) has multiple data input channels. If you have a camera that is connected to a channel other than the first of its digitizer, you must specify the channel, using **MdigChannel()**.



Switching between cameras of the same type

When connecting several cameras of the same data format to different data input channels of a digitizer, allocate a single digitizer with the appropriate DCF for the first camera and use **MdigChannel()** to switch between the others of the same type.

Switching between cameras of different types

When connecting cameras of different data formats to different data input channels of a digitizer, you must use a different DCF for each camera. You could allocate a digitizer, grab the required frame, free the digitizer, and then allocate the digitizer again with the second DCF; this can increase the time required for the operation. MIL can circumvent this problem by using a fast DCF-switching technique which is outlined in the steps below:

1. Make as many calls to **MdigAlloc()** as you have cameras, with different formats, from which you want to grab.
 2. Specify the required digitizer settings using **MdigControl()**, **MdigReference()**, and **MdigHookFunction()** for each allocated digitizer.
 3. Specify the channel to use for the grab using **MdigChannel()**.
 4. Call **MdigGrab()** with an allocated digitizer identifier. Note that if this call uses a digitizer identifier different from the previous call, a DCF switch will occur.
- ❖ If there is a grab in progress with one digitizer, calling any of the following functions with any other digitizer will result in an error: **MdigGrab()**, **MdigGrabContinuous()**, **MdigInquire()**, **MdigProcess()**, **MdigAlloc()**, and **MdigFree()**. To synchronize your grabs with different digitizers, use **MthrWait()**.

Ultra-fast channel switching

If you intend to switch channels after grabbing a single field or frame, use the ultra-fast channel switching mode to minimize time between grabs; set the **M_CAMERA_LOCK** control type to **M_ENABLE+M_FAST**. This mode is optimized for applications that use several cameras of the same type and it does impose certain restrictions:

- Some digitizer hook events might only be generated a few frames after the channel switch: **M_FRAME_START**, **M_FIELD_START**, **M_FIELD_START_ODD**, and **M_FIELD_START_EVEN**.
- Trigger inputs are unavailable.
- In frame mode, the following hook events are unreliable: **M_GRAB_FIELD_END**, **M_GRAB_FIELD_END_ODD**, and **M_GRAB_FIELD_END_EVEN**. Note these hook events are reliable in field mode.

Channel locking and unlocking

If the synchronization between a digitizer and a camera is uncertain, enable **MdigControl()** with **M_CAMERA_LOCK** to lock the digitizer to the synchronization signal of the camera after a channel-switch. This provides additional stability. In this case, the digitizer is unlocked from the first camera and then locked to the next camera after a channel-switch occurs.

Typically, MIL finds the best balance between the fastest lock and the most reliable lock possible, depending on your camera. On some Matrox frame grabbers, you can control channel locking sensitivity using **MdigControl()** with **M_CAMERA_LOCK_SENSITIVITY** and **M_CAMERA_UNLOCK_SENSITIVITY**. Note that the speed and reliability of a camera lock directly affects the subsequent unlock and vice versa (for example, a fast lock might create an unreliable unlock, or a reliable lock might cause a slow unlock).

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**, grabbing every second row and storing them in sequential rows. 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 events occur relative to the grabbed field: **M_GRAB_FRAME_START**, **M_GRAB_END**, and **M_GRAB_FRAME_END**. To achieve 60 fps in NTSC or 50 fps in PAL, the 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 (row) 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

To grab to the display, you should grab into a displayable grab buffer (**MbufAlloc...** with **M_DISP+M_GRAB**) that is selected for display (**MdispSelect()**). The display of the selected buffer is typically maintained with separate internal buffers in display memory or Host non-paged memory. The selected buffer and the display are maintained and synchronized internally.

The synchronization frequency is dependent upon whether performing a monoshot or continuous grab. A monoshot grab updates the selected buffer first and then its display. A continuous grab updates the specified buffer's display immediately after each frame is grabbed; only the last frame is stored in the selected buffer for processing. If you need to save/process each grabbed frame, you should perform the grab using **MdigGrab()** (that is, perform a monoshot grab); alternatively, you can use **MdigProcess()** for its multiple buffering capabilities.

When grabbing, the frame grabber (for example, Matrox Solios) always acts as the bus master. When grabbing to an on-board display section or when grabbing to a graphics controller that supports fast linear-memory accesses to its display memory, the frame grabber can generally transfer all grabbed data directly to display memory. In the latter case, the display memory also has to be physically accessible from the frame grabber. This is the case for most AGP display boards and PCI display boards plugged into the same PCI-X bus segment as the frame grabber.

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]);
}
```

Alternatively, you can make a single call to **MdigProcess()**. For more information, see the *Multiple buffering* subsection in the *Grabbing and processing* section in *Chapter 22: Grabbing with your digitizer*.

In either case, you must allocate buffers to store the frames of the sequence. After you have grabbed a sequence, you can use the **MbufExportSequence()** function to export the sequence of image buffers (compressed or uncompressed 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 images in all format supported by MIL. You can also choose to import the sequence into automatically allocated buffers or previously allocated buffers.

Grabbing and processing

To optimize application performance when grabbing, you can:

- Set the grab mode.
- Perform multiple buffering.

Grab mode

When grabbing data with **MdigGrab()**, you can control the synchronization by setting the **M_GRAB_MODE** control type of **MdigControl()** to **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 function.
- 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 functions to execute while you are still grabbing. This is a useful option when performing multiple buffering, a technique whereby you can grab data into one buffer while processing previously grabbed buffers (discussed below). Note, another call to **MdigGrab()** before the current grab has finished will cause your application to wait until the current grab has finished.
- If your frame grabber 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()**.

Note that **MdigGrabContinuous()** is by definition asynchronous since you must use **MdigHalt()** to stop the grab.

Multiple buffering

Multiple buffering involves grabbing into one image buffer while processing previously grabbed images. This technique allows you to grab and process images concurrently.

To perform multiple buffering in MIL, you can use **MdigProcess()**. It allows you to use a list of previously-allocated buffers to hold a series of sequentially grabbed images and process them as they are being grabbed. These grabs can either continue until stopped (using **MdigProcess()** with **M_START** to start and then **M_STOP** to stop) or continue until all buffers in the list are filled (using **MdigProcess()** with **M_SEQUENCE**). In the former case, **MdigProcess()** grabs round-robin through the list of buffers, however care must be taken to ensure that the average time it takes to process a frame is not greater than the frame rate of a camera, so that frames will not be missed.

MdigProcess() hooks a user-defined function to the modification of any buffer in the specified list. So every time a new frame is grabbed in a buffer in the list, the user-defined function is called. If the hook-handler function modifies the buffer that called it, the hook-handler function for that buffer will not be called again until that buffer is modified by a new grab. This is to avoid infinite recursive calls being generated. Note that, if the buffer is modified by some function other than the one hooked to it, the hooked function will be called. In addition, if the hook-handler function modifies another buffer which has another function hooked, that function will be called.

Note, processing is generally faster if the buffer is not on the display.

The following example shows you how to perform multiple buffering.

```

/*****/
/*
 * File name: MDigProcess.cpp
 *
 * Synopsis: This program shows the use of the MdigProcess() function to do perform
 *           real-time processing.
 *
 *           The user's processing code is written in a hook function that
 *           will be called for each frame grabbed (see ProcessingFunction()).
 *
 * Note: The average processing time must be shorter than the grab time or
 *       some frames will be missed. Also, if the processing results are not

```



```

*           displayed and the frame count is not drawn or printed, the
*           CPU usage is reduced significantly.
*/
#include <mil.h>

/* Number of images in the buffering grab queue.
   Generally, increasing this number gives better real-time grab.
*/
#define BUFFERING_SIZE_MAX 22

/* User's processing function prototype. */
MIL_INT MFTYPE ProcessingFunction(MIL_INT HookType, MIL_ID HookId,
                                void MPTYPE *HookDataPtr);

/* User's processing function hook data structure. */
typedef struct
{
    MIL_ID  MilImageDisp;
    MIL_INT ProcessedImageCount;
} HookDataStruct;

/* Main function. */
/* -----*/

int MosMain(void)
{
    MIL_ID MilApplication;
    MIL_ID MilSystem      ;
    MIL_ID MilDigitizer   ;
    MIL_ID MilDisplay     ;
    MIL_ID MilImageDisp   ;
    MIL_ID MilGrabBufferList[BUFFERING_SIZE_MAX] = { 0 };
    MIL_INT MilGrabBufferListSize;
    MIL_INT ProcessFrameCount = 0;
    MIL_INT NbFrames          = 0, n=0;
    MIL_DOUBLE ProcessFrameRate= 0;
    HookDataStruct UserHookData;

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

    /* Allocate the grab buffers and clear them. */
    MappControl(M_ERROR, M_PRINT_DISABLE);
    for(MilGrabBufferListSize = 0;
        MilGrabBufferListSize<BUFFERING_SIZE_MAX; MilGrabBufferListSize++)
    {
        MbufAlloc2d(MilSystem,
                    MdigInquire(MilDigitizer, M_SIZE_X, M_NULL),
                    MdigInquire(MilDigitizer, M_SIZE_Y, M_NULL),
                    M_DEF_IMAGE_TYPE,
                    M_IMAGE+M_GRAB+M_PROC,

```

```

        &MilGrabBufferList[MilGrabBufferListSize]);

    if (MilGrabBufferList[MilGrabBufferListSize])
    {
        MbufClear(MilGrabBufferList[MilGrabBufferListSize], 0xFF);
    }
    else
        break;
}
MappControl(M_ERROR, M_PRINT_ENABLE);

/* Free buffers to leave space for possible temporary buffers. */
for (n=0; n<2 && NbFrames; n++)
{
    MilGrabBufferListSize--;
    MbufFree(MilGrabBufferList[MilGrabBufferListSize]);
}

/* Print a message. */
MosPrintf(MIL_TEXT("\nMULTIPLE BUFFERED PROCESSING.\n"));
MosPrintf(MIL_TEXT("-----\n\n"));
MosPrintf(MIL_TEXT("Press <Enter> to start.\n\n"));

/* Grab continuously on the display and wait for a key press. */
MdigGrabContinuous(MilDigitizer, MilImageDisp);
MosGetch();

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

/* Initialize the User's processing function data structure. */
UserHookData.MilImageDisp      = MilImageDisp;
UserHookData.ProcessedImageCount = 0;

/* Start the processing. The processing function is called for every frame grabbed. */
MdigProcess(MilDigitizer, MilGrabBufferList, MilGrabBufferListSize,
            M_START, M_DEFAULT, ProcessingFunction, &UserHookData);

/* NOTE: Now the main() is free to perform other tasks
                                           while the processing is executing. */
/* ----- */

/* Print a message and wait for a key press after a minimum number of frames. */
MosPrintf(MIL_TEXT("Press <Enter> to stop.\n\n"));
MosGetch();

/* Stop the processing. */
MdigProcess(MilDigitizer, MilGrabBufferList, MilGrabBufferListSize,
            M_STOP, M_DEFAULT, ProcessingFunction, &UserHookData);

```

```

/* Print statistics. */
MdigInquire(MilDigitizer, M_PROCESS_FRAME_COUNT, &ProcessFrameCount);
MdigInquire(MilDigitizer, M_PROCESS_FRAME_RATE, &ProcessFrameRate);
MosPrintf(MIL_TEXT("\n\n%d frames grabbed at %.1f frames/sec (%.1f ms/frame).\n"),
          ProcessFrameCount, ProcessFrameRate, 1000.0/ProcessFrameRate);
MosPrintf(MIL_TEXT("Press <Enter> to end.\n\n"));
MosGetch();

/* Free the grab buffers. */
while(MilGrabBufferListSize > 0)
    MbufFree(MilGrabBufferList[--MilGrabBufferListSize]);

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

return 0;
}

/* User's processing function called every time a grab buffer is modified. */
/* -----*/

/* Local defines. */
#define STRING_LENGTH_MAX 20
#define STRING_POS_X      20
#define STRING_POS_Y      20

MIL_INT MFTYPE ProcessingFunction(MIL_INT HookType, MIL_ID HookId,
                                void MPTYPE *HookDataPtr)
{
    HookDataStruct *UserHookDataPtr = (HookDataStruct *)HookDataPtr;
    MIL_ID ModifiedBufferId;
    MIL_TEXT_CHAR Text[STRING_LENGTH_MAX]= {MIL_TEXT('\0')};

    /* Retrieve the MIL_ID of the grabbed buffer. */
    MdigGetHookInfo(HookId, M_MODIFIED_BUFFER+M_BUFFER_ID, &ModifiedBufferId);

    /* Print and draw the frame count. */
    UserHookDataPtr->ProcessedImageCount++;
    MosPrintf(MIL_TEXT("Processing frame #d.\r"), UserHookDataPtr->ProcessedImageCount);
    MosSprintf(Text, STRING_LENGTH_MAX, MIL_TEXT("%ld"),
              UserHookDataPtr->ProcessedImageCount);
    MgraText(M_DEFAULT, ModifiedBufferId, STRING_POS_X, STRING_POS_Y, Text);

    /* Perform the processing and update the display. */
    MimArith(ModifiedBufferId, M_NULL, UserHookDataPtr->MilImageDisp, M_NOT);

    return 0;
}

```

Grabbing large images

In some cases, you might need to grab an image whose size is in the order of gigabytes. Ideally, you would use a digitizer whose DCF specifies an appropriate frame size. In practice, this is not always possible because the on-board frame size is restricted by the memory space on board. If you try to set a larger frame size using Matrox Intellicam, you will obtain an error when you call **MdigAlloc()** due to insufficient memory space for the temporary on-board buffers required for the large frame.

A solution to the above problem is to do the following:

1. Allocate a large buffer on the Host using **MbufAlloc...** with **M_GRAB**.
2. Allocate child buffers in the large parent buffer using **MbufChild....** The size of each child buffer will have to be equal to or smaller than the supported frame size specified by the DCF settings.
3. Put each child buffer into an array for easy access later.
4. Grab sectional frames of the large image into each child buffer in the array.

As mentioned in the *Multiple buffering* subsection in the *Grabbing and processing* section in *Chapter 22: Grabbing with your digitizer*, to ensure that no frames are missed during a grab, you can use **MdigProcess()**. **MdigProcess()** grabs a sequence of images into an array of image buffers and can cause a user-defined function to be called after an image has been grabbed into any of these buffers.

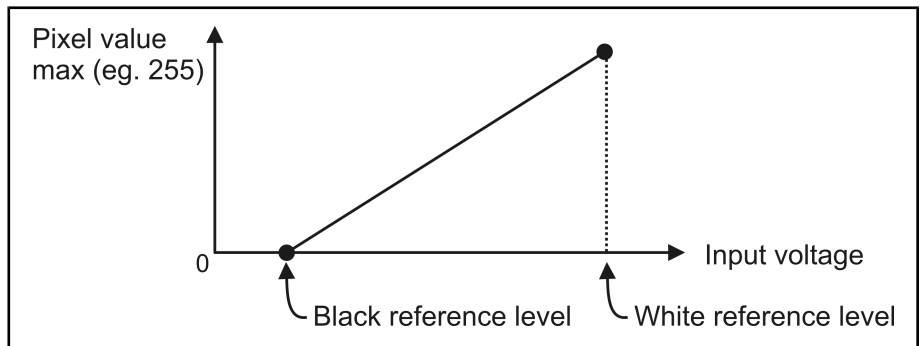
Reference levels, lookup tables, and scaling

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 for your particular board.

$$\text{Value to pass to MdigReference() } = \left(\frac{\text{Voltage needed} - \text{minimum voltage}}{\text{Maximum voltage} - \text{minimum voltage}} \right) (\text{M_MAX_LEVEL} - \text{M_MIN_LEVEL})$$

The smallest voltage increment supported by your board can differ such that consecutive reference-level settings might produce the same result.

Note, the new reference level might not take effect until the next grab, at which point, a certain amount of delay might be incurred as the hardware adjusts to the reference-level changes.

Color image reference levels

When grabbing composite color images, **MdigReference()** provides specific control parameters to adjust the levels of contrast, brightness, hue, and saturation. These levels can be set to values from 0 to 255; use values appropriate for your particular board.

Mapping grabbed data through a LUT

If supported by your digitizer, grabbed data is mapped through a physical lookup table (LUT). By default, the physical LUT is transparent; that is, it is loaded with data so that each input pixel is mapped to its original value. You can load the physical LUT with your own custom LUT data if, for example, you want to correct or precondition grabbed data. To do so, copy a LUT buffer to the digitizer's physical input LUT, using **MdigLut()**. MIL uses the data format (DCF) of the digitizer to determine whether a physical LUT is supported. If it is not, an error is generated.

The characteristics of the LUT buffer and the digitizer's DCF establishes how the physical LUT is actually configured.

Configuration of the digitizer's LUT	Determining factor
Number of entries in the digitizer's physical LUT	Data being grabbed (DCF)
Depth (8- or 16-bit)	LUT buffer
Number of bands	DCF & LUT buffer

The digitizer's physical LUT is typically configured to have the same number of components (bands) as either the LUT buffer or the data to be grabbed (determined by the digitizer's DCF), depending on which has more bands. The digitizer's physical LUT is also configured to have the same number of entries as the maximum possible value per band of the data to grab. The depth of a digitizer's physical LUT is configured to be either 8- or 16-bits per band, depending on if the LUT buffer depth is 8-bit or 16-bit, respectively. Cameras, however, are not so limited. When dealing with a 10- or 12-bit camera, use a 16-bit destination grab buffer and load the digitizer's physical LUT with the difference zero padded.

To copy the data from a LUT buffer to the digitizer's physical LUT, the number of entries in the LUT buffer must match those of the digitizer's physical LUT. In addition, if the digitizer's physical LUT cannot support the depth of the LUT buffer, an error will occur.

LUT buffer data is loaded into the digitizer's physical LUT, as follows:

LUT buffer	Digitizer's physical LUT	Result
1 band	1 band	The LUT buffer is copied directly into the digitizer's physical LUT.
1 band	3 band	The LUT buffer is copied into each component of the digitizer's physical LUT.
3 band	1 band	The first band of the LUT buffer is copied into the digitizer's physical LUT.
3 band	3 band	Each of the LUT buffer's bands are copied into the corresponding component of the digitizer's physical LUT.

If the destination grab buffer depth is larger than that of the digitizer's physical LUT, the destination grab buffer's least significant bits are set to zero when the data is grabbed. If the digitizer's physical LUT depth is greater than that of the destination grab buffer, the most-significant bits of the data (the non-zero values) are used when the data is grabbed.

To revert back to using a transparent LUT, you must copy the default LUT (**M_DEFAULT**) to the digitizer's physical input 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 0.5, only one column and one row out of two are written to the image buffer (starting with the first row and column).

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	115	0	0	0	0	0	0	0
0	215	244	0	0	0	0	0	0	0
0	215	243	196	196	0	0	0	0	0
0	215	111	111	87	87	87	87	86	87
0	0	0	111	115	87	87	87	87	0
0	0	0	0	111	111	115	45	0	0
0	0	0	0	0	111	92	92	0	0
0	0	0	0	0	0	111	111	0	0

Original image



0	0	0	0	0
0	115	0	0	0
0	243	196	0	0
0	0	115	87	87
0	0	0	92	0

Subsampled image
X subsampling factor = 0.5
Y subsampling factor = 0.5

The X- and Y-scaling factors are independent. Note, depending on the Matrox frame grabber and camera used, some scaling factors might not be available.

To disable scaling, set the scaling factors to 1.

Grabbing with triggers and exposures

If your Matrox frame grabber supports trigger input, you can set up your digitizer to perform a triggered grab, that is, to grab a frame upon the occurrence of an event. In this case, nothing is grabbed when you call **MdigGrab()** or **MdigGrabContinuous()**, 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.

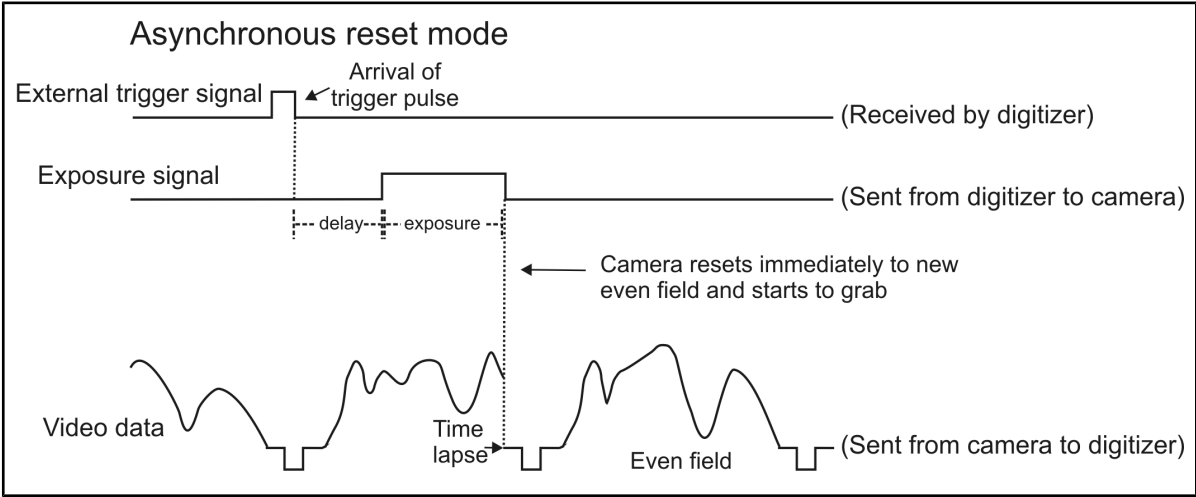
The digitizer configuration format (DCF) can initialize the digitizer to perform a triggered grab and can indicate exactly how it should be carried out. For example, if the DCF specifies that an exposure signal should be generated (for the camera) upon the grab trigger event, the actual grab would only be triggered once the active exposure time was over.

You can use **MdigControl()** with the `M_GRAB_TRIGGER...` control types to override the DCF trigger settings. For example, you can enable/disable whether **MdigGrab()** and **MdigGrabContinuous()** perform 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 `M_GRAB_TRIGGER_MODE`, respectively.

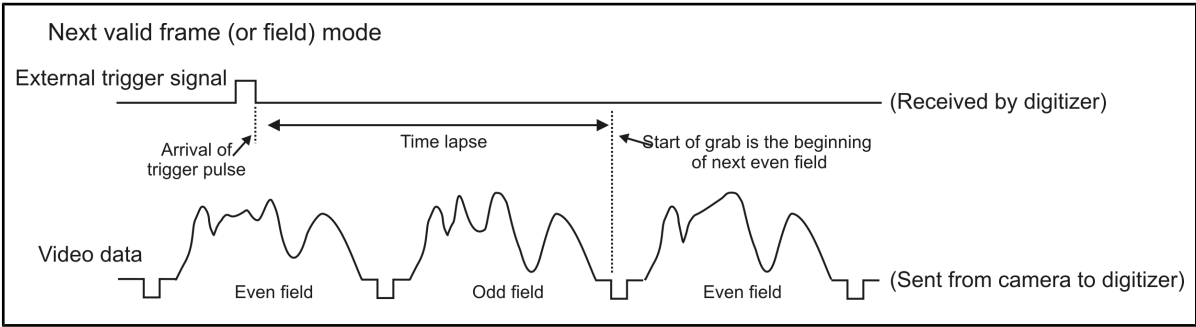
Asynchronous reset mode and next valid frame mode

There are two possible grab activation modes: asynchronous reset mode and next valid frame mode. Note that the grab activation mode is specified in the DCF file.

If your frame grabber and your camera support asynchronous reset mode, you can set up the digitizer to send an exposure signal which resets the camera when the trigger signal is received. Upon being reset, you can have the camera start exposing a new frame or use the exposure signal to determine when to start exposing a new frame. The behavior of the camera varies depending on its model; please consult your camera's documentation for more specific details regarding its interpretation of the exposure signal.

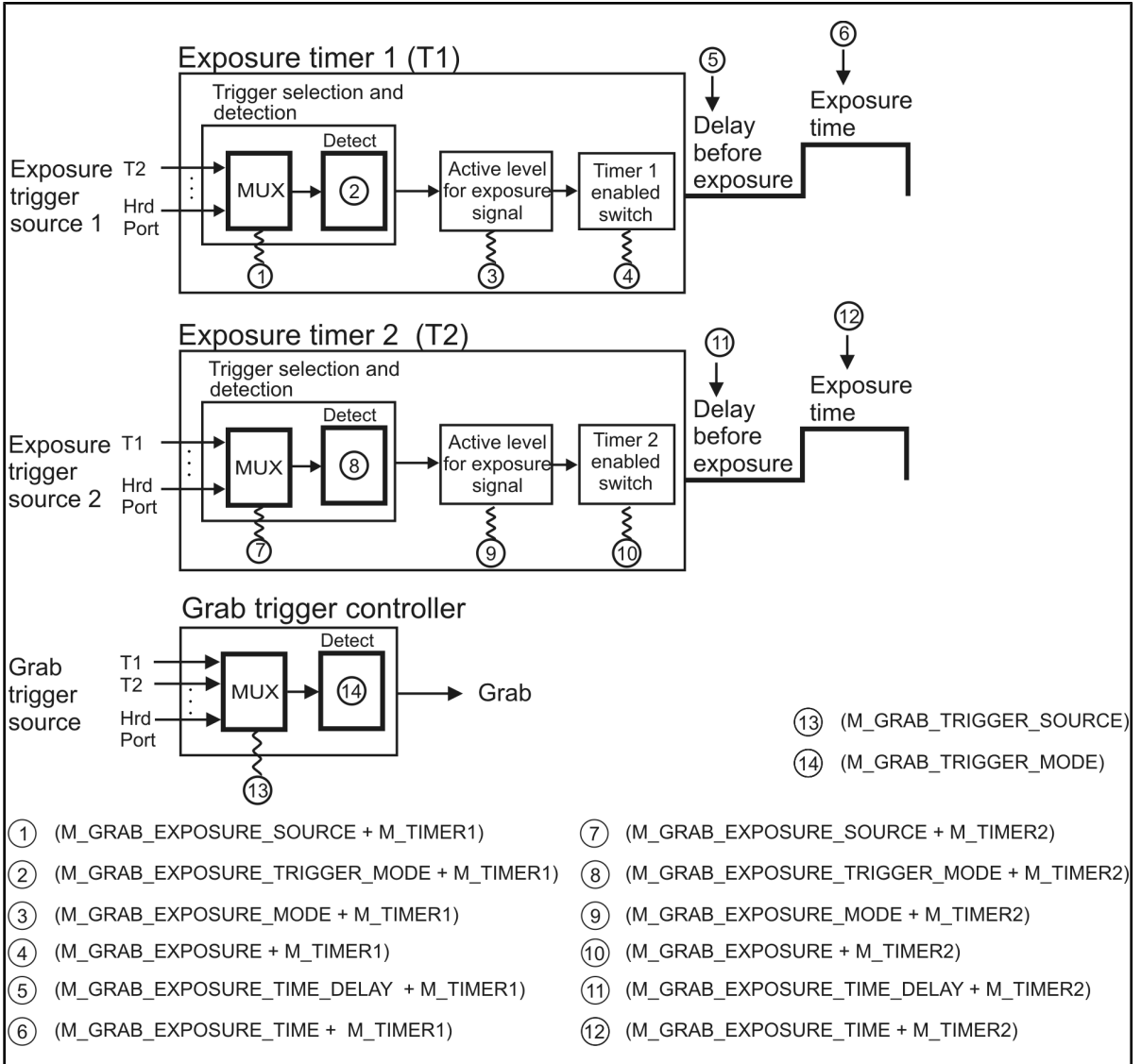


Otherwise, the digitizer waits for the next valid frame (or field) before starting to grab.



Setting the exposure

If your frame grabber and your camera support it, you can set up your digitizer to transmit an exposure signal to your camera, to specify when to start exposing a new frame and the duration for exposure. To do so, you must enable and set-up the required exposure timers and grab trigger connections as illustrated below.



Software triggers

In general, the digitizer's grab trigger receiver and exposure timers can also be triggered by software (use **MdigControl()** with **M_GRAB_TRIGGER_SOURCE** set to **M_SOFTWARE**). In this case, following a grab call, nothing is grabbed until you call a specific function. To issue a software trigger for the grab trigger receiver, 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 that in this case, the grab call must be asynchronous (that is, you must issue the grab with **MdigGrab()** in asynchronous mode or with **MdigGrabContinuous()**), or the grab call must be called on a separate thread.

Note, for a frame grabber without an exposure timer, the exposure time is considered to be zero.

Auto-focusing

Note that this section is not supported with MIL-Lite.

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 the quality of the focus in your images is unacceptable and physically moving the grabbed object 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 operations 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()**.

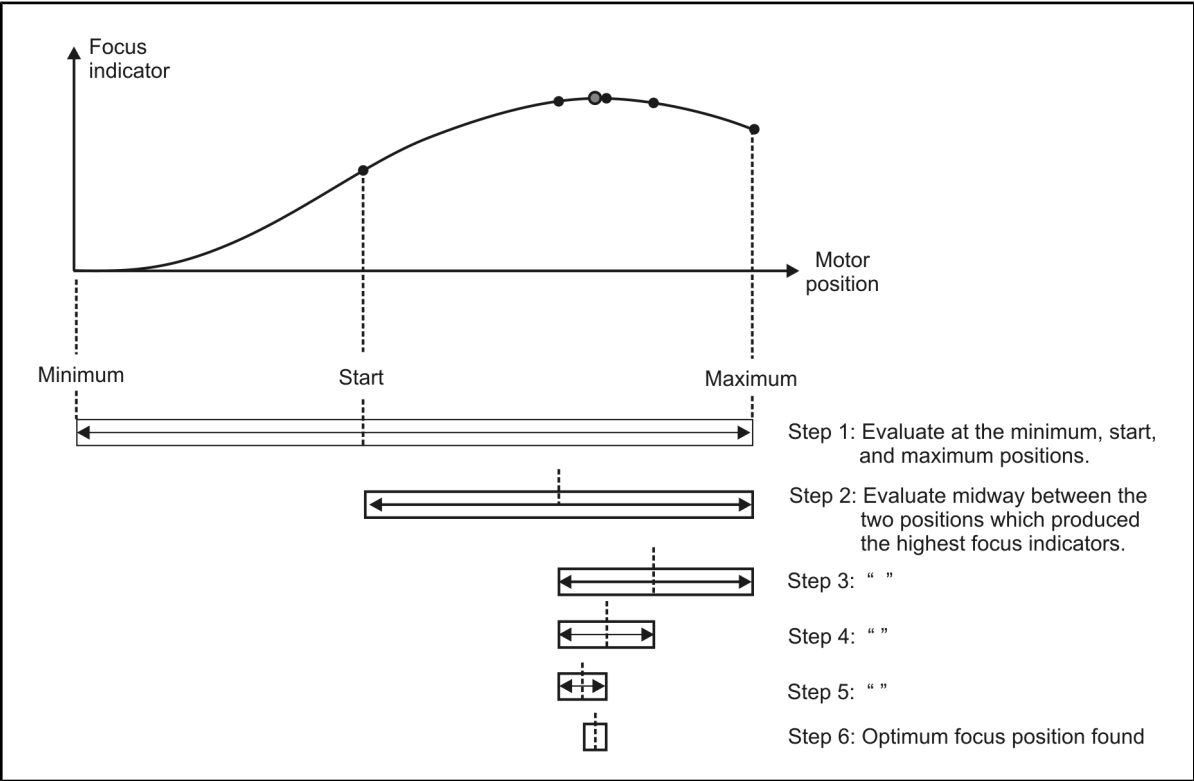
If necessary, you can specify that only a sub-region of the image be analyzed, by passing a child buffer to the function. This is primarily useful if there are objects at different distances within the camera's field of view. In such a case, each object will have a different optimum focus position, so 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. The default search strategy used by MIL is the smart-scan strategy.

Bisection strategy

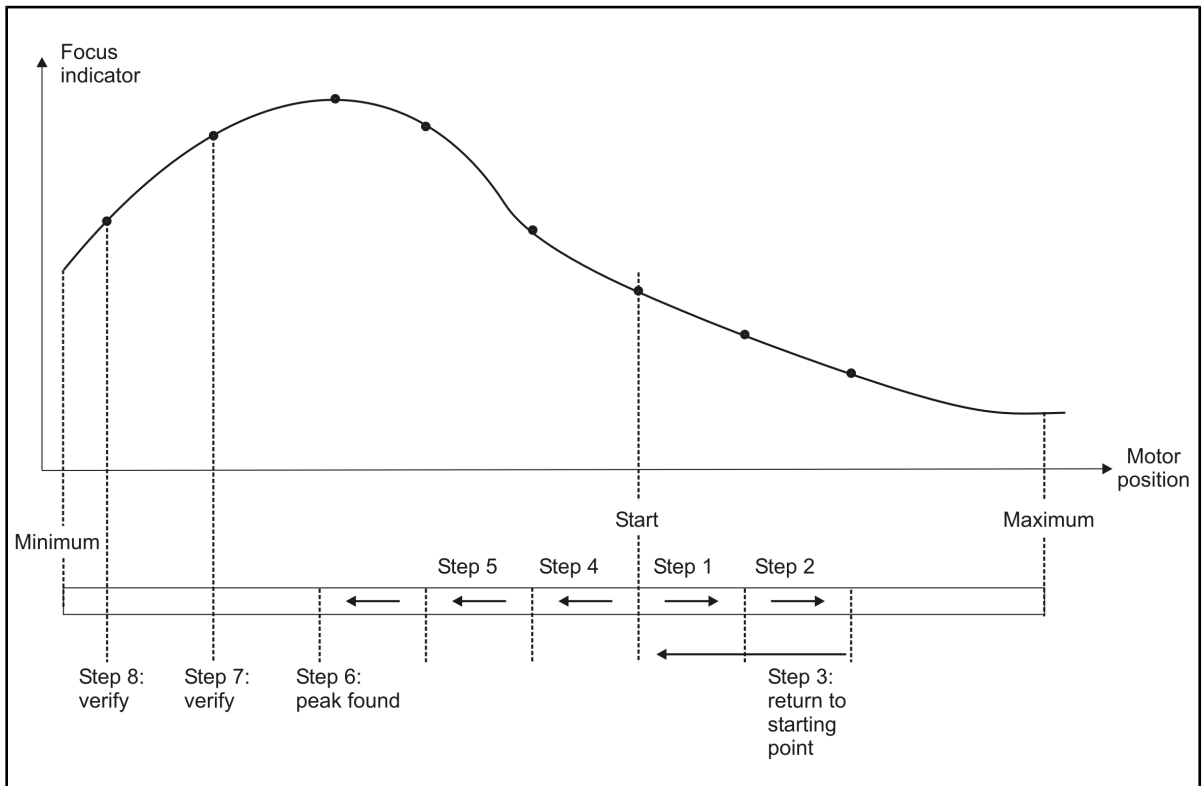
The bisection strategy breaks down the given positional range, step-by-step, until it finds the optimum focus position.



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

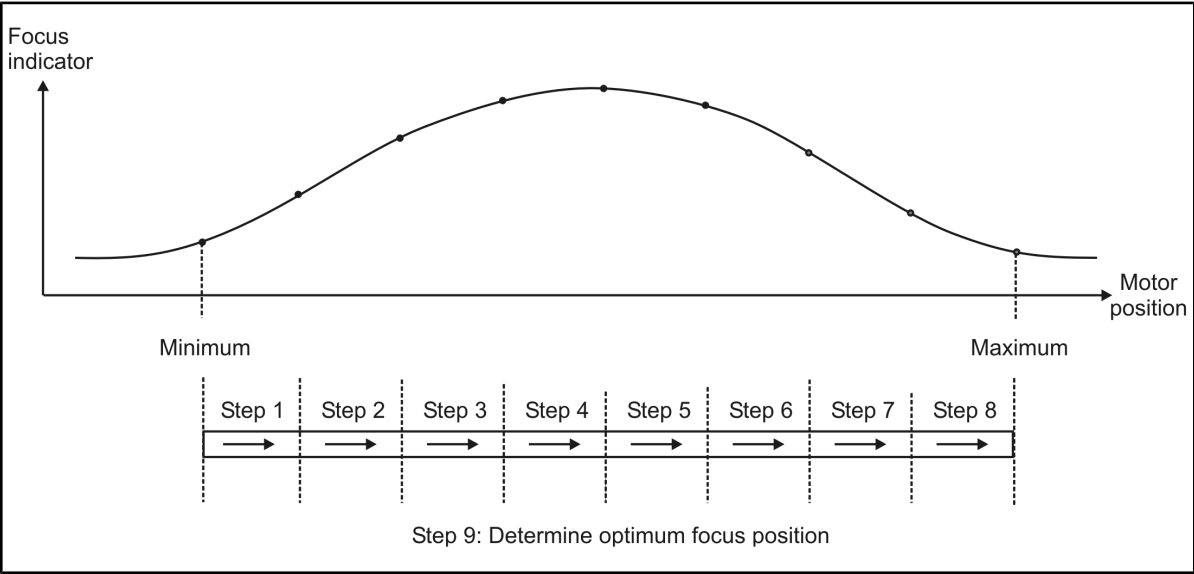
The refocus strategy scans upward or downward until it finds the optimum focus position or until it reaches the minimum or maximum position. While scanning in one direction, if the focus indicator decreases continuously (indicating an out-of-focus condition), the focus position is returned to its starting 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

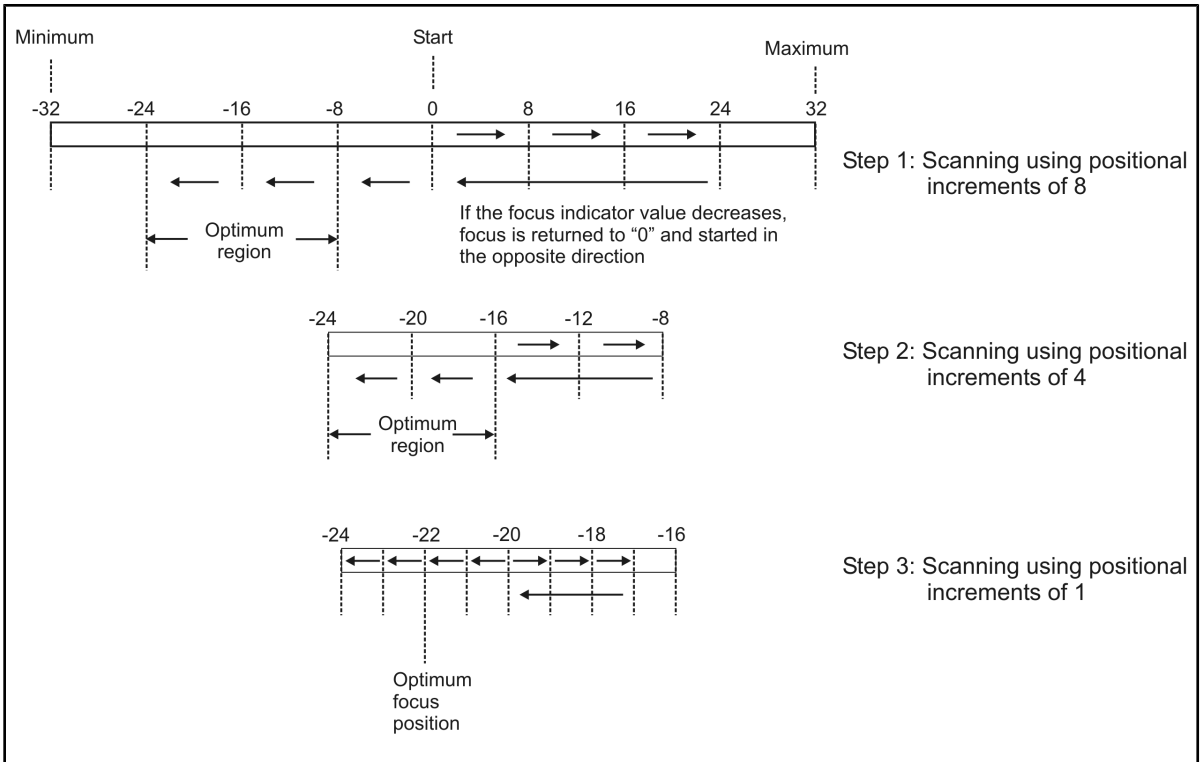
The scan-all strategy scans, by 1, all positions between the minimum and maximum and returns the position which produced the highest focus indicator.



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

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 JPEG2000 compression algorithms.

- ❖ Note that MIL-Lite compression support is reliant upon the presence of Matrox compression acceleration hardware. The compression algorithm supported depends on the hardware. If you don't have Matrox compression acceleration hardware, you can still perform compression under MIL-Lite with the compression/decompression runtime license package installed.

Although processing operations support compressed source and/or destination image buffers, these operations take longer on a compressed image than on an uncompressed image. This is because the processing operation must decompress/compress the image before/after performing its operation. It is recommended that compressed images be used only for data transfer.

JPEG lossless

The JPEG lossless algorithm compresses images without any loss of information. Typically, the algorithm compresses images by a factor of 2:1, although a factor of 4:1 can sometimes be achieved. The JPEG lossless algorithm can compress 8- or 16-bit buffers with 1 or 3 bands.

JPEG lossy

The JPEG lossy algorithm compresses images by a variable factor but introduces some loss of information. The higher the compression factor, the more the compression, but the lower the image quality. The JPEG lossy algorithm can compress 8-bit buffers with 1 or 3 bands. To be compatible with most image-viewing software, MIL allows you to store compressed color images in YUV format.

Interlaced JPEG

MIL can perform a JPEG compression such that the image data is stored in separate fields. This is referred to as an interlaced JPEG compression. Unless otherwise stated, everything that applies to a JPEG compression also applies to an interlaced JPEG compression.

JPEG2000 lossless and lossy

JPEG2000 is another standard for image compression supported by MIL. 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 *JPEG2000* section later in this chapter. Note, however, that JPEG2000 is best suited for archiving purposes because of the processing time required to compress and decompress images; therefore, when JPEG2000 is used for grabbing, there is a risk of missing frames.

JP2 standard

In addition to saving files in the regular JPEG2000 lossless or lossy file format, it is also possible to save files in the JPEG2000 lossless format or lossy format using the JP2 standard. While the JPEG2000 file format is only a dump of the compressed image's data, the JP2 standard file format also contains an additional information header. Typically, this optional information contains intellectual property rights, color palette, color space definition, or application-specific data. The headers of JP2 files saved using MIL only include color specifications. If the image was previously loaded into a MIL buffer from a JP2 file, any information in the file header other than color specification will be lost.

MIL adds the JP2 header when the buffer is exported to a file.

Control options

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

This section discusses general steps in compressing and decompressing images and sequences, as well as other aspects of compression.

Compression

To compress an image and save it in an image buffer:

1. Allocate a buffer in which to hold the compressed image. Use **MbufAlloc...** with **M_COMPRESS** combined with a compression attribute value.
 - ❖ Compressed buffers that are created using the **MbufCreate...** functions should not be used as the destination buffer of a MIL function. However, if a buffer with an **M_COMPRESS** attribute 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 JPEG2000 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()**.

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 JPEG2000 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.

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.

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 JPEG2000 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 control types that specify luminance (such as **M_QUANTIZATION_LUMINANCE**) to control the Y band; to control the U and V bands, use the control types that specify chrominance (such as **M_HUFFMAN_DC_CHROMINANCE**). The control types without these suffixes control all bands. See the MIL 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 - JPEG2000

When you allocate a multi-band buffer for a JPEG2000 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_BANDS** will control all bands.

Application-specific markers

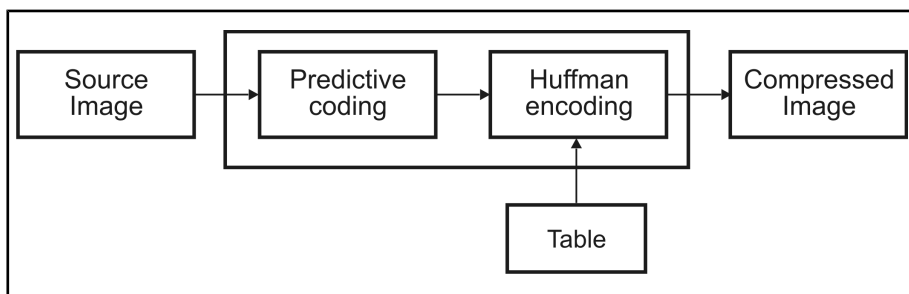
During a compression, MIL adds some application-specific markers to the resulting image. Most other packages will ignore these markers and therefore be able to decompress the file. MIL itself ignores unrecognized markers when it decompresses files.

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 (<http://www.iso.ch>). For techniques to use to improve compression operations, see the *Improving results* section later in this chapter.

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 use `MbufControl()` with the `M_PREDICTOR` control type to specify the predictor.

Huffman encoding

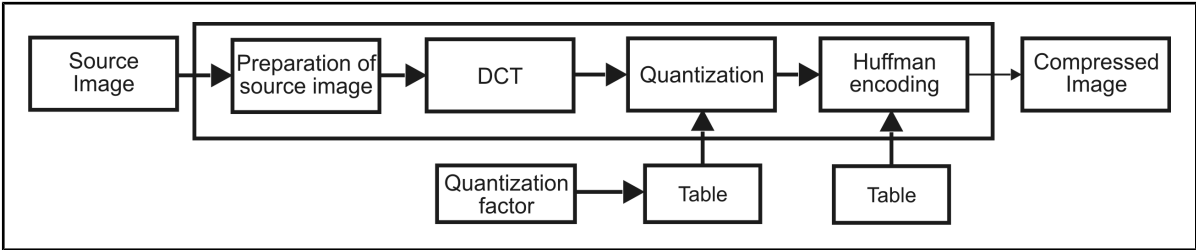
After an image has been predictive coded, Huffman encoding assigns a variable-length "code word" to each value. This code is based on the number of bits by which adjacent values differ. Values are assigned code words according to a DC Huffman table. You can use the default DC Huffman table or you can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

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 **MbufControl()** with the **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 for the quantization factor, which you can change without in-depth knowledge, using `MbufControl()` with `M_Q_FACTOR`). For techniques to use to improve compression operations, see the *Improving results* section later in this chapter. For more detailed information about the JPEG2000 algorithm, see <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.

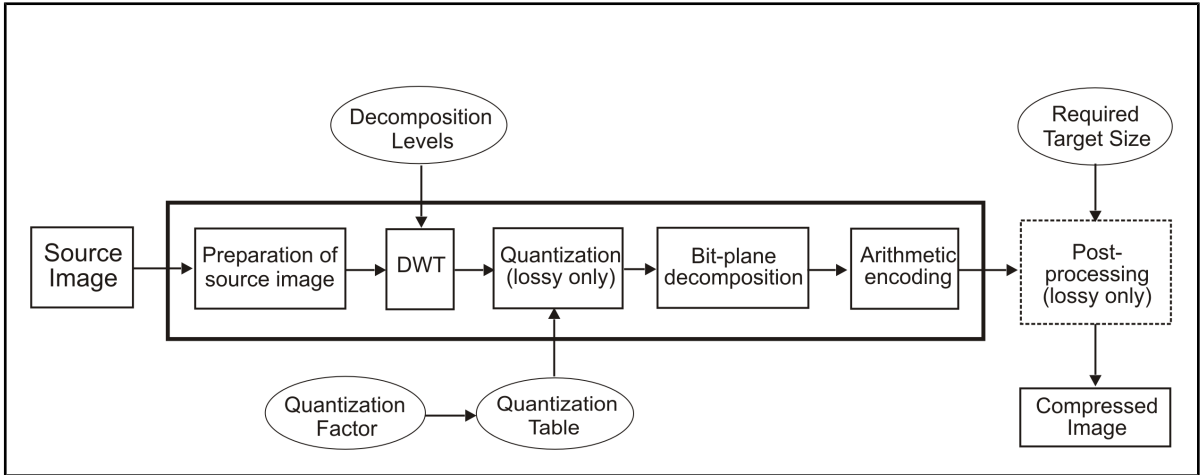
When compressing images with the JPEG2000 algorithm, MIL supports:

- Custom quantization tables (using **MbufControl()** with **M_QUANTIZATION**).
- Custom wavelet settings (setting the number of decomposition levels using **MbufControl()** with **M_DECOMPOSITION_LEVEL**).
- Saving files either in regular JPEG2000 format or in JPEG2000 format with the JP2 standard, as well as saving AVI sequences in JPEG2000 format.
- Band-specific compression settings.
- Easy control over image size and quality (setting the quantization factor using **MbufControl()** with **M_Q_FACTOR**).
- Specifying a target size for the compressed image when performing a JPEG2000 lossy compression (using **MbufControl()** with **M_TARGET_SIZE**).
- Tiling (only supported when using Matrox JPEG2000 compression acceleration hardware).

Note that MIL does not support the following JPEG2000 features:

- Progressive encoding/decoding (layering).
- Random bit-stream access (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).

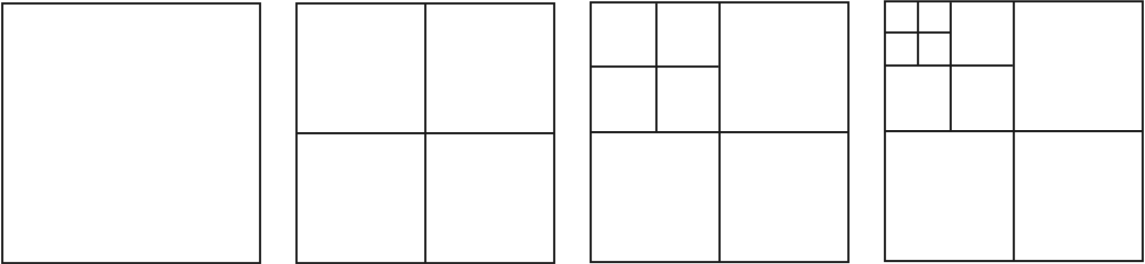
Preparation of source image

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.

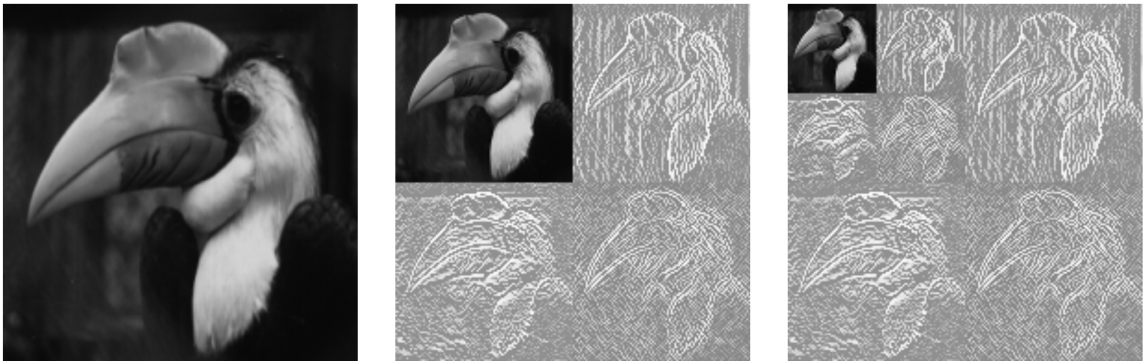
After the computational shift, a color conversion is performed if the source and destination buffers are in different formats, for example an RGB source buffer and a YUV destination buffer. Note that conversion from RGB to YUV naturally introduces some loss.

Discrete wavelet transform (DWT)

After preparation of the source image, a lossy or lossless discrete wavelet transform (DWT) is applied to 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 applied 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 3 decomposition levels



Example of a Discrete Wavelet Transform of 2 decomposition levels

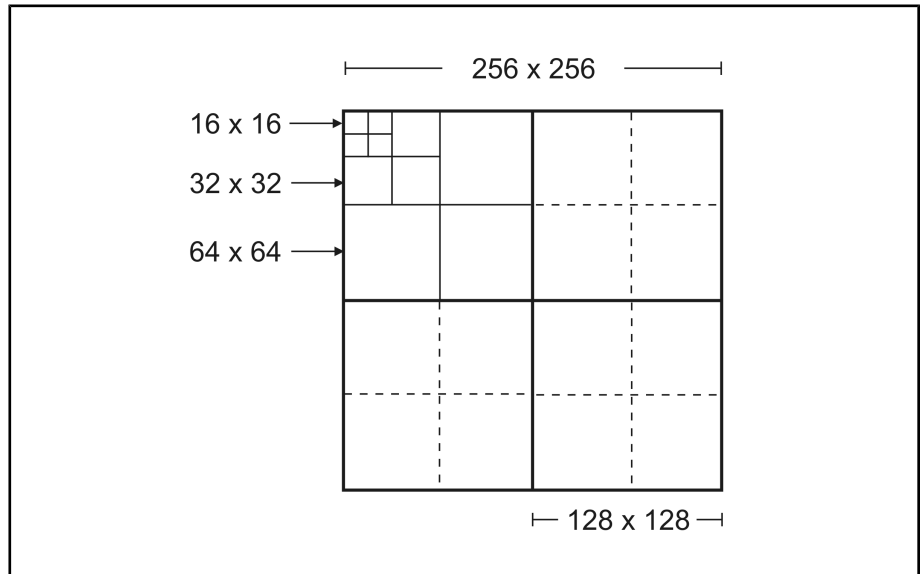
The default number of decomposition levels is 5, unless the width or the height of the image is less than or equal to 16 pixels; in which case, the number of decomposition levels is chosen so that the width or the height of the 4 smallest sub-bands of the image is 1 pixel. You can override the default using **MbufControl()** with **M_DECOMPOSITION_LEVEL**. For example, if your original image is 1024 x 768, by default, the number of decomposition levels is 5, and the size of the 4 smallest sub-bands is 32 x 24. If you set the **M_DECOMPOSITION_LEVEL** control type to 6, then the size of the smallest sub-bands would be 16 x 12. The largest number of decomposition levels is the one that yields a width or a height of 1 pixel for the 4 smallest sub-bands. After allocating the image buffer, you can inquire the number of decomposition levels that will be used, using **MbufInquire()** with **M_DECOMPOSITION_LEVEL**.

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

The larger sub-bands are further segmented into smaller regions, referred to as **code-blocks**. Smaller sub-bands are not segmented, and each contains one code-block. For example, if 64×64 code-blocks are used, the sub-band that is 128×128 contains four code-blocks, while the sub-bands that are 64×64 , 32×32 , and 16×16 each contain one code-block, as illustrated in the diagram below.



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 JPEG2000 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
	2	3
	5	6

❖ JPEG2000 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 (using **MbufControl()** with **M_Q_FACTOR**), 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. For more information, see the *Working with tables* section later in this chapter.

Bit-plane 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.

010	100	001	0	1	0	1	0	0	0	0	1
101	100	000	1	1	0	0	0	0	1	0	0
001	010	111	0	0	1	0	1	1	1	0	1
Segmented region's coefficients			First bit-plane			Second bit-plane			Third bit-plane		

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 (using `MbufControl()` with `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 using **MbufControl()** with **M_Q_FACTOR**.
- Decrease the restart interval.
- Change the quantization table (see the *Working with tables* section later in this chapter).
- Change the Huffman table (see the *Working with tables* section later in this chapter).

For JPEG lossless compression:

- Try the other supported predictors using **MbufControl()** with **M_PREDICTOR**.
- Decrease the restart interval.

For JPEG2000 lossy compression:

- Allocate a YUV buffer for compression.
- Increase the quantization factor using **MbufControl()** with **M_Q_FACTOR**.
- Decrease the target size using **MbufControl()** with **M_TARGET_SIZE**.
- Change the number of decomposition levels of the DWT using with **M_DECOMPOSITION_LEVEL**.
- Change the quantization table (see the *Working with tables* section later in this chapter).

For JPEG2000 lossless compression:

- Change the number of decomposition levels of the DWT using **MbufControl()** with **M_DECOMPOSITION_LEVEL**.

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 values 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 JPEG2000 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 array size for each compression algorithm.

Compression algorithm	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
JPEG2000 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 and store them in the user array using **MbufGet....**

Using your own table

To use your own table:

1. Allocate a buffer with an **M_ARRAY** attribute and of the same data type as the default table.
2. Transfer the custom table values from the user array to the array buffer, using **MbufPut...**, 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 (for example, use the **M_HUFFMAN_AC** control type for an AC Huffman table). Specifying these control types as-is, or combined with **M_ALL_BANDS**, controls all bands for JPEG2000. This is the default setting.

For a JPEG2000 compression, you can associate a different table with each band of a multi-band buffer. To do so, add **M_RED**, **M_BLUE**, or **M_GREEN** to your control type for an RGB buffer, whereas for a YUV buffer, add **M_Y**, **M_U**, or **M_V**.

For JPEG lossy compressions of YUV buffers, use the luminance and 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

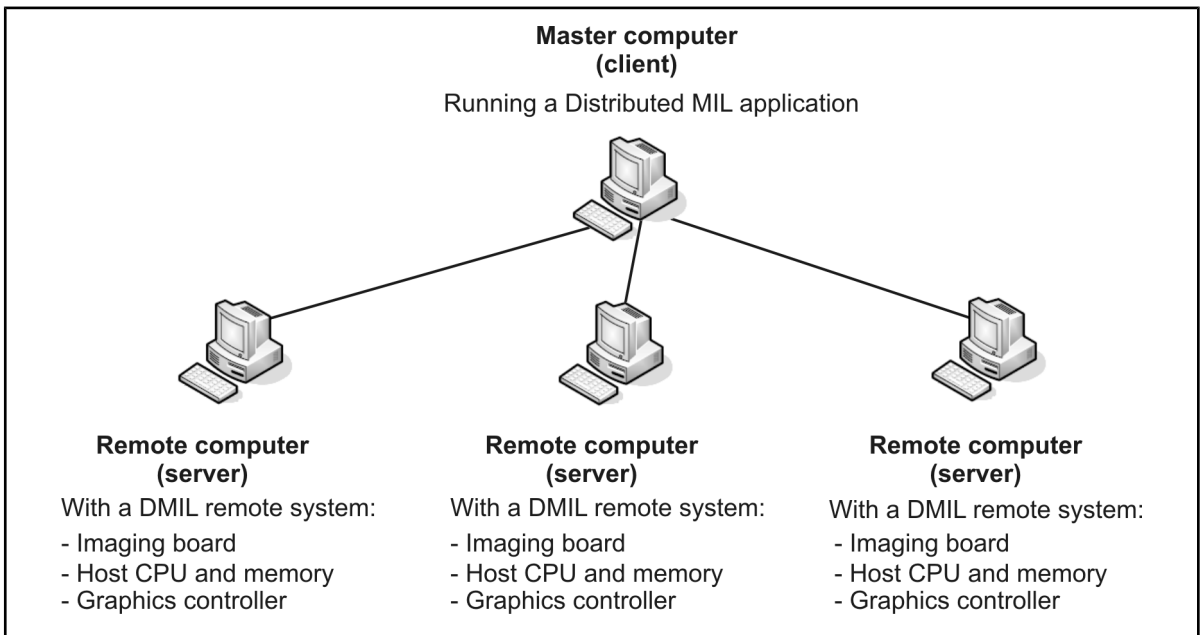
Distributed MIL

This chapter describes how to set up a Distributed MIL application. Distributed MIL allows for multiple MIL systems to collaborate in a MIL application across a network.

Distributed MIL overview

Distributed MIL allows multiple MIL systems to collaborate in a MIL application (**MappAlloc()**) across a network (for example, a local area network, or a wide area network such as the internet). This allows a MIL application running on one computer to use the resources of other computers, for example, to distribute processing or to grab remotely and display locally.

A group of MIL systems working together in an application is called a cluster. Clusters are managed by a MIL application running on a master computer (client). An application can manage up to a maximum of 255 systems; these can be local or remote. When allocated on a remote computer, a system is called a DMIL remote system. You can allocate multiple board-type and Host-type DMIL remote systems on a remote computer (server); Host-type DMIL remote systems make use of the main processor of the remote computer. A simple cluster is illustrated below:



Whether running locally on the master computer or running remotely, all MIL functions behave as usual, unless specific behavior is wanted. In addition, all resources in a cluster can be shared transparently (automatic connection), except when network address translation (NAT), port forwarding, or other address translation tools are used. If these tools are used, a remote computer can only share resources with the master computer.

The remote computers must be running the Distributed MIL server. The server opens a TCP port to wait for connections, and executes requests from the master computer or other remote computers in the cluster (a remote computer considers these all as clients). The master computer runs the MIL application. When the application allocates a DMIL remote system, the Distributed MIL client is started. The Distributed MIL client manages the connection and communication with the remote computers.

To develop or run a Distributed MIL application, MIL/MIL-Lite must be installed on the master and remote computers. All computers should have the same version of MIL installed, including Processing Packs and Updates, with appropriate licenses; see the *Licensing issues* subsection in the *Preparing master and remote computers* section in *Chapter 24: Distributed MIL*.

All the computers in a cluster must be running only one of the following: a 32-bit Windows operating system, a 64-bit Windows operating system, a 32-bit Linux operating system, or a 64-bit Linux operating system. For example, if one computer is running Windows XP 64-bit, another can be running Windows Vista 64-bit, but none can be running Linux or a 32-bit Windows operating system. In addition, if exchanging image buffers across the network, it is recommended that the computers have at least a GigE Ethernet interface (1000BaseT). The connection between computers is made using TCP/IP.

Steps to create a Distributed MIL application

The following steps provide a basic methodology for creating a Distributed MIL application:

1. Allocate a MIL application, using **MappAlloc()**.
2. If required, allocate a system on the master computer, using **MsysAlloc()**. If you don't specify a target computer when allocating a system, MIL allocates the system locally on the current computer (master computer).
3. Allocate a DMIL remote system on each of the remote computers, using **MsysAlloc()** with the target remote computer's identifier (for example, IP address).
4. Call the MIL functions to perform the required operations, specifying objects allocated on these DMIL remote systems. Based on the objects passed, MIL will perform the functions on the most appropriate computer; if some of the objects are not on the selected computer, they will be copied to it. For this reason, MIL functions are most efficient when operating on objects whose systems are allocated on the same computer.
5. Free all allocated objects, in the reverse order from which they were allocated.

Before you can run a Distributed MIL application, you must prepare the master computer and each of the remote computers with the appropriate software. See the *Preparing the master and remote computers* section later in this chapter.

Basic concepts

The basic concepts and vocabulary conventions for this chapter are:

- **Board-type system.** A board-type system consists of a Matrox imaging board (or third-party board that is compatible with MIL), the Host CPU and memory, and any available graphics controller.
- **Cluster.** A cluster is a group of systems working together in a MIL application; the systems can be allocated on the master computer or on a remote computer. Clusters are managed by an ActiveMIL application running on the master computer (client).
- **DMIL remote system.** A system that is running on a remote computer.
- **Distributed MIL.** Distributed MIL is a MIL feature which allows multiple MIL systems to collaborate in a MIL application across a network.
- **Distributed MIL server.** The Distributed MIL server is a program running on a remote computer, which allows the remote computer to be part of a cluster.
- **Host-type system.** A Host-type system consists of the Host CPU and memory, and any available graphics controller.
- **Master computer (or the main Host computer).** The master computer is the computer that is running the main MIL application.
- **Remote computer.** A remote computer is a computer that is not running the main MIL application, but is running a Distributed MIL server, which allows the remote computer to be part of a cluster.
- **Remote system.** A remote system is a MIL system that executes its operations, by default, using a processor whose address space is separate from the main Host processor(s) of the master computer; to execute operations, the remote processor uses a version of MIL that is separate from the one used to run the main application on the main Host processor(s). A remote system can be a DMIL remote system, or a system that uses a Matrox imaging board with an on-board processor (such as, Matrox Odyssey).

- **System.** A system can be either a **board-type system** or a **Host-type system**. A board-type system consists of a Matrox imaging board (or third-party board that is compatible with MIL), the Host CPU and memory, and any available graphics controller. A Host-type system consists of the Host CPU and memory, and any available graphics controller. Systems of a remote and master computer can collaborate within a MIL application using Distributed MIL.

Preparing the master and remote computers

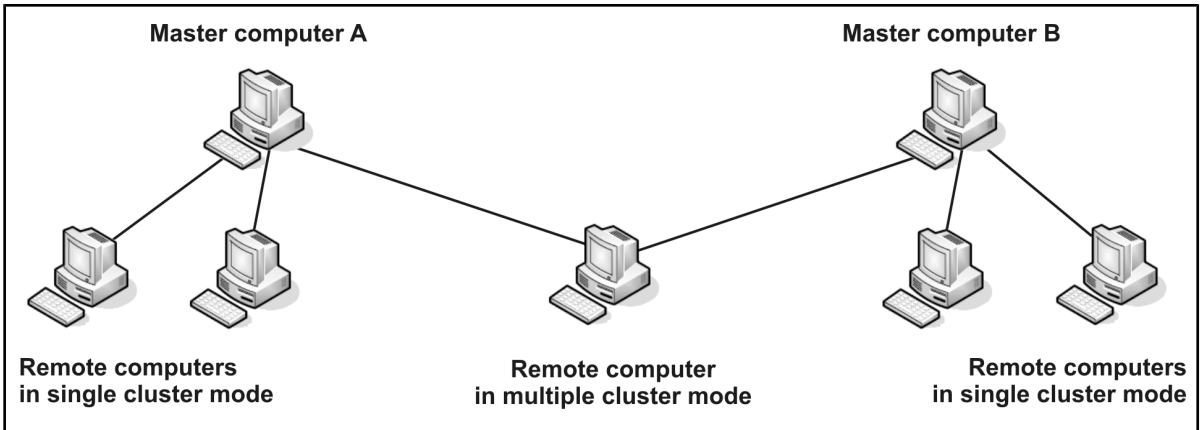
To use Distributed MIL, you must prepare the master computer and the remote computers with the appropriate software.

On the master computer and each of the remote computers, you must install MIL or MIL-Lite. During installation, select the Distributed MIL option from the list of board drivers to install. You will then be presented with a number of options. These options are described in this section; you can later reconfigure these options using MILConfig. Once MIL/MIL-Lite is installed, you must make sure that all the computers in the cluster have the appropriate licenses.

- ❖ Note that all the computers in a cluster must be running only one of the following: a 32-bit Windows operating system, a 64-bit Windows operating system, a 32-bit Linux operating system, or a 64-bit Linux operating system. For example, if one computer is running Windows XP 64-bit, another can be running Windows Vista 64-bit, but none can be running Linux or a 32-bit Windows operating system. In addition, if exchanging image buffers across the network, it is recommended that the computers have at least a GigE Ethernet interface (1000BaseT).

Single or multiple cluster mode

Before installing MIL/MIL-Lite, you must decide whether a remote computer should be configured in single or multiple cluster mode. A remote computer configured in single cluster mode can only belong to one cluster. A remote computer configured in multiple cluster mode can belong to more than one cluster.



Setting up the Distributed MIL server on the remote computers

During MIL/MIL-Lite installation, you can select to install the Distributed MIL server. You should select this option if the current computer will be used as a remote computer. The Distributed MIL server manages all of a remote computer's connections; the server transparently opens and closes the required connections as necessary during the course of an application.

A remote computer must be running the Distributed MIL server to accept new incoming connections and to create new outgoing connections. During installation (and with MILConfig), you can select to start the server as a service or to start it manually.

You will typically want to start the Distributed MIL server as a service. This will cause the server to start automatically when you boot the computer, and to recover automatically if the server crashes; you won't need to logon to the computer to start the server. You can choose to start the service in single cluster mode or in multiple cluster mode.

The Distributed MIL server cannot be running as a service if you plan to allocate a remote GPU system or to present a display on the remote computer; in this case, you must select to start the server manually (**Manually launched**). Then, before a Distributed MIL application can access the computer, you must logon to the computer and run the executable of the Distributed MIL server (*MilNetworkServer.exe*). To run the executable every time you logon, add it to the remote computer's Startup folder. The executable has the following syntax:

```
MilNetworkServer [--multiple-clusters] [--spawn_and_monitor]
```

To run the executable in multiple cluster mode, call the executable with the **--multiple-clusters** option; to run it in single cluster mode, omit this option. If you want the Distributed MIL server to recover from a crash, call the executable with the **--spawn_and_monitor** option too.

Listening port/server connection port

You don't need to manage all the separate connections required between the systems in your cluster. However, you must establish a port on the remote computers to which their Distributed MIL server should listen for new connection requests, and your application should access to initiate new connections.

By default, the Distributed MIL server listens to port 57010 for new connection requests. If the remote computer uses this port for other purposes, you can specify a different listening port when installing MIL/MIL-Lite (or using MILConfig) on this computer. In this case, you should update the port that your application accesses on the remote computer to initiate a new connection. During MIL/MIL-Lite installation (or using MILConfig) on the master computer, you can change the default port that your application accesses (default server connection port). If some of your remote computers don't use this default port, your application must explicitly specify the port to access on these remote computers when allocating a system on them.

- ❖ Note that your application remains more portable if your application does not explicitly specify the port to access during system allocation.

Port range used by remote computers in multiple cluster mode

Depending on the number of DMIL remote systems allocated in a Distributed MIL application, the master computer can have many Distributed MIL connections open at any given time. How MIL manages Distributed MIL connections depends on whether the remote computer is configured in single or multiple cluster mode.

If the remote computer is in single cluster mode, the master computer connects to the remote computer using the default or explicitly specified server connection port.

If the remote computer is in multiple cluster mode, only the initial connection is made using default or explicitly specified server connection port. You must establish a range of ports on the remote computer which the Distributed MIL server could use for subsequent connections. You can either use the default range or specify a range when installing MIL/MIL-Lite (or using MILConfig) on the remote computer. The connection to a remote computer in multiple cluster mode happens as follows:

1. The master computer connects to the remote computer using the default or explicitly specified server connection port.
2. The remote computer spawns a new instance of the Distributed MIL server.
3. The new instance of the Distributed MIL server selects a connection port from the range of ports specified when installing MIL/MIL-Lite (or using MILConfig) on the remote computer.
4. The remote computer informs the master computer of the newly selected port.
5. The master computer disconnects from the remote computer.
6. The master computer reconnects to the remote computer using the port selected in step 3.

In this process, the remote computer dynamically selects the port for its connections depending on which ports in its specified range are available; this port management technique allows for the computer to be part of multiple clusters.

The connection between computers is made using TCP/IP.

When running your application, you can confirm that connections have been set up properly by monitoring the Distributed MIL server's output using MILConfig on the remote computer.

Firewall configuration

You will need to ensure that your networking equipment (for example, firewalls and proxy servers) has been configured properly so as to allow Distributed MIL to create all of its required connections.

The master computer and the remote computers must be able to accept data from each other on the specified TCP port and port range. When dealing with computers that have Windows Firewall enabled, this can be achieved by selecting **Unblock** when the operating system notifies the user that the MIL application needs to accept connections from the network.

To ensure optimal performance, these ports should be unblocked by default. If you are using the Windows Firewall, you can unblock these ports when installing MIL/MIL-Lite (or using MILConfig) by selecting them as Windows Firewall exceptions. Alternatively, you can click on the **Windows Firewall** icon in the Control Panel, switch to the Exceptions tab, and click on the **Add Port** button. In the presented dialog box, specify the ports to unblock.

Instead of unblocking the above-mentioned ports, you can disable the firewall for these ports using the Advanced tab of the Windows Firewall configuration interface. Disabling the firewall will stop the firewall software from watching the specified ports. Note that disabling the firewall on specific ports is not recommended if your computers are connected to a large network (such as, the internet).

Licensing issues

To develop or run a Distributed MIL application, you must have the appropriate licenses:

Product	Mode of operation	Master computer	Remote computers
MIL	Developing	MIL development license.	Run-time licenses for Distributed MIL and any MIL module that the remote computer actually uses.
	Running	Run-time licenses for Distributed MIL and any MIL module that the remote computer actually uses.	
MIL-Lite	Developing	Supplemental licenses for Distributed MIL and any other functionality actually used that requires a supplemental license.	Supplemental licenses for Distributed MIL and any other functionality actually used that requires a supplemental license.
	Running		

All computers in the cluster need a Distributed MIL license to either send or receive a command. In addition, each computer needs the licenses for the modules/functionality that they actually use; sending a command to another computer is not using a module so no specific license is needed in this case. For example, if the application on the master computer calls a Model Finder function using objects allocated on a remote computer, the operation will actually be performed on the remote computer; therefore, only the remote computer needs a Model Finder license.

For more information on licensing, see *Chapter 26: Distribution and licensing*.

Allocating DMIL remote systems

All of the MIL systems in a Distributed MIL application are allocated in the same manner as in a traditional MIL application without remote computers, using **MsysAlloc()**.

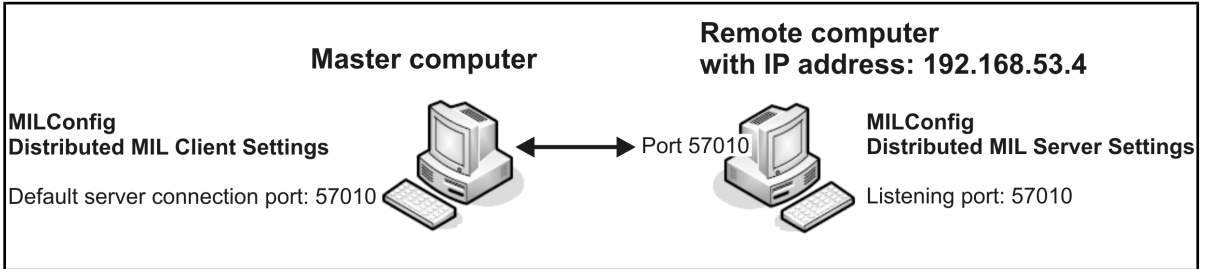
Allocating the systems

To allocate a remote system, the **SystemDescriptor** parameter of **MsysAlloc()** must specify the type of system to allocate, as well as identify the location of the target remote computer on the network. To override the default Distributed MIL server connection port, **SystemDescriptor** should also specify the required port. The full syntax for the **SystemDescriptor** parameter is as follows:

`"dmiltcp://RemoteComputerIdentifier:PortNumber/MILSystemType"`

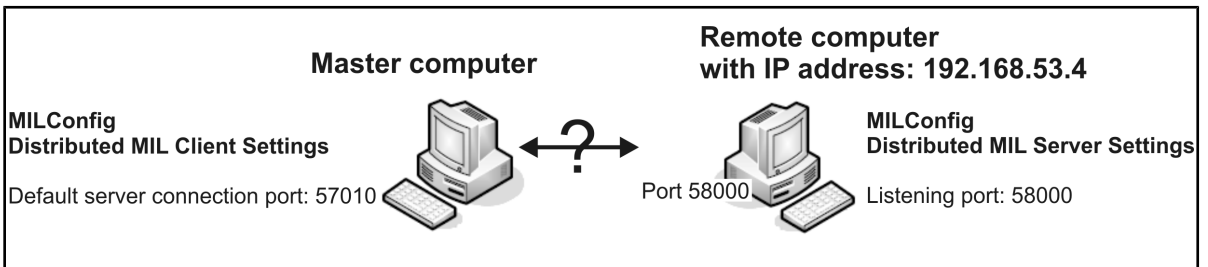
where *RemoteComputerIdentifier* should be replaced with the remote computer's name or IP address; MIL supports both IPv4 and IPv6 addresses. *PortNumber* should be replaced with the port that the master computer should access on the remote computer to initiate new connections, unless the default server connection port is appropriate; in which case, omit the ":" and the port number. *MILSystemType* should be replaced with any valid MIL system type, such as **M_SYSTEM_SOLIOS**. The "://" and "/" are required separators.

For example, in the following case, you don't need to explicitly specify a server connection port; the default matches the port to which the Distributed MIL server is listening on the remote computer.



```
/* Connection made without specifying port. */
MsysAlloc(MIL_TEXT("dmiltcp://192.168.53.4/M_SYSTEM_HOST"), M_DEFAULT, M_DEFAULT,
           &RemoteSystemId);
```

Whereas in the following case, you must explicitly specify a server connection port because by default, it does not correspond to the port to which the Distributed MIL server is listening. The server connection port must be set to 58000 to establish a connection.



```
/* In this case, the connection cannot be made without specifying port - an error */
/* is generated with the following call. */
MsysAlloc(MIL_TEXT("dmiltcp://192.168.53.4/M_SYSTEM_HOST"), M_DEFAULT, M_DEFAULT,
           &RemoteSystemId);

/* Since the port is specified and it matches the listening port on remote computer, */
/* the connection is made with the following call. */
MsysAlloc(MIL_TEXT("dmiltcp://192.168.53.4:58000/M_SYSTEM_HOST"), M_DEFAULT, M_DEFAULT,
           &RemoteSystemId);
```

If you specify a server connection port, ensure that the specified port corresponds to the listening port set when installing MIL/MIL-Lite (or using MILConfig) on the target remote computer.

A typical IPv4 string has the format n.n.n.n, where n is a number between 0 and 255. A typical IPv6 string has the format x:x:x:x:x:x:x, where x is a hexadecimal number between 0000 and FFFF. If you are supplying an IPv6 address, you must use square brackets to separate the address from the port. For example:

```
"dmiltcp://[x:x:x:x:x:x:x]:PortNumber/MILSystemType"
```

- ❖ Before you can allocate a remote GPU system, the Distributed MIL server on the remote slave computer must have access to the remote display desktop. For more information, see the *Setting up the Distributed MIL server on the remote computers* subsection in the *Preparing master and remote computers* section in *Chapter 24: Distributed MIL*.

Setting the default system to a remote system

You can set the default system (**M_SYSTEM_DEFAULT**) to a system on a remote slave computer. You must first register the target Matrox Imaging board (or supported third-party board) on the remote slave computer with the master computer. To do so, configure the **Distributed MIL Client Settings** page in MILConfig on the master computer. Enter the name or IP address of the remote slave computer. From the presented list of possible systems that can be allocated on the remote computer, select which you want to register with the master computer. Click on the **Apply** button. Then, in the **General Default Values** page in MILConfig, you can select one of the registered remote boards as your default system.

- ❖ Note that you can allocate the default system of the remote computer without registering the target board. This means that you can set the **SystemDescriptor** parameter to *"dmiltcp://RemoteComputerIdentifier/M_SYSTEM_DEFAULT"*, without registering the target board on the remote computer with the master computer. In addition, a registered board can still be allocated from another computer.

Using DMIL remote systems

When using a DMIL remote system, you should be aware of certain aspects of MIL.

Execution of MIL functions on remote computers

MIL decides where to execute a MIL function based on the MIL objects passed as parameters to the function. Therefore, to perform the required operations on a remote computer, allocate a DMIL remote system on this computer and then pass objects allocated on this remote system to the corresponding MIL functions.

When all of the MIL objects passed are allocated on the same DMIL remote system, MIL executes the function on the computer (processor) associated with this system.

When the MIL objects passed are allocated on different systems, MIL establishes if there is a computer (processor) associated with one of the systems most suitable to execute the operation. To be suitable, MIL must be able to temporarily copy all of the MIL objects involved, to this computer, if they are not already located there. Only objects allocated with the Buffer module (**Mbuf...**) can be copied to another computer (**portable**); other objects are not portable. If a MIL object is not portable, the computer associated with this object is typically selected as the most suitable. If two of the MIL objects are not portable and are on different computers, an error is generated.

If MIL establishes that a remote computer is the most suitable to execute a MIL function, the master computer sends a command to this computer to execute the function. Then, using inter-system calls, the remote computer typically communicates directly with the other remote computers, to fetch a copy of any required MIL objects located on these computers and to send back any changes. Inter-system calls improve performance when multiple remote computers are involved. If an object is located on the master computer, inter-system calls are not used; the master computer will perform the required copy operations.

Since additional copy operations are avoided, MIL functions are most efficient when operating on objects whose systems are all allocated on the same computer.

Defaults on remote computers

When a function is executed on the remote computer and you select **M_DEFAULT**, it is the default value set up on that remote computer that is used. The default value could have been setup when installing MIL on that computer or using the MILConfig utility on that computer.

The only exception is when allocating a display. In this case, it is the default value on the master computer, unless the display is allocated such that it is displayed on the remote computer. For more information, see the *Displays and DMIL remote systems* section later in this chapter.

Files

When a function takes a file name and is executed on the remote computer, MIL assumes that the file is on the master computer. To specify a file on the remote computer, prefix the specified file name with "remote:////" (for example, "remote:///C:\mydirectory\myfile").

Asynchronous calls

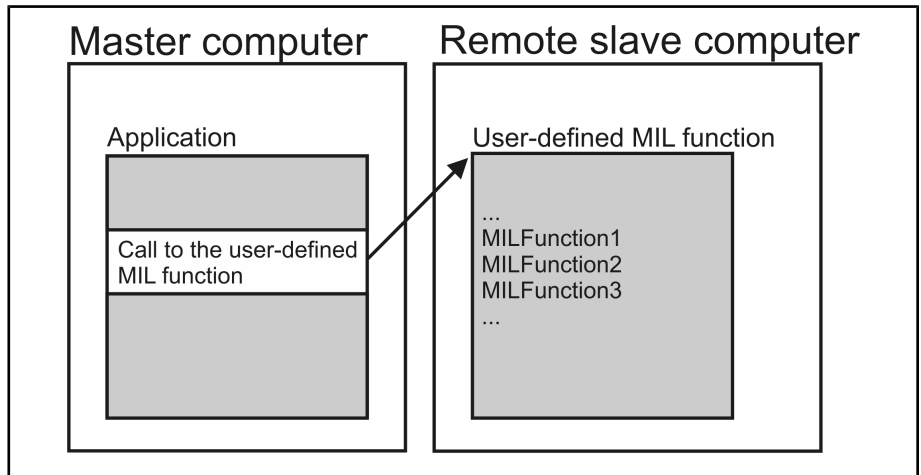
Most functions executed on a remote computer are executed asynchronously. The function execution request is sent to the remote computer and the local call returns immediately, without waiting for the request to complete. Except for functions that return values (for example, allocation and inquire functions), almost all other functions are asynchronous. Asynchronous calls are queued on the remote computer and processed in the order that they were issued on a thread-by-thread basis. Asynchronous calls can be executed asynchronously from their initial designated thread on the remote computer if the DMIL remote target system has an on-board processor (for example, Matrox Odyssey).

Multi-threading

Distributed MIL supports multi-threading. As with systems with on-board processors (for example, Matrox Odyssey), each thread on the master computer has a corresponding thread on the remote computer. All requests made on a specific thread on the master computer are executed on the same corresponding thread on the remote computer; the sequence of calls in a thread is respected. Threads on the remote computer are independent and parallelism on the master computer is respected. Distributed MIL provides full support of the MIL Thread module, regardless of the system allocated.

Executing a user-defined function on the remote computer

When time is of essence and you have a series of functions that need to be executed on a remote computer and some are synchronous, you can create a MIL user-defined function that is executed on the remote computer and that calls the required series of functions.



This avoids unnecessary communication with the master computer. To develop such a MIL user-defined function, see *Chapter 27: The MIL function development module*.

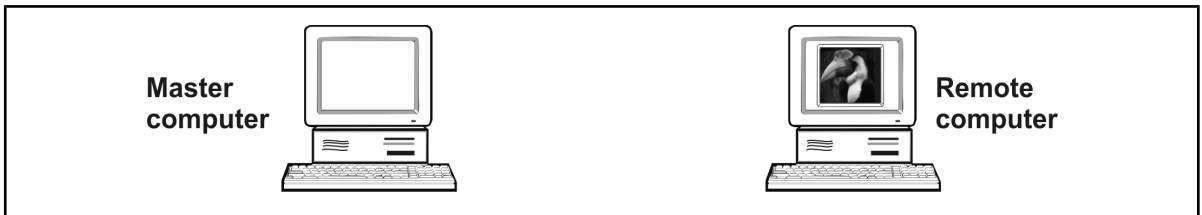
Displays and DMIL remote systems

When you allocate a display on a DMIL remote system, you can select image buffers allocated on that system, to the display. By default, these buffers are displayed on the master computer.



```
/* Allocate a display on a DMIL remote system, which displays image buffers
/* on the master computer. */
MdispAlloc(RemoteSystemId, M_DEFAULT, MIL_TEXT("M_DEFAULT"), M_WINDOWED, &DisplayId);
MdispSelect(DisplayId, RemoteImageBufId);
```

If you want to display the buffers on the remote computer, specify **M_REMOTE_DISPLAY** when allocating the display. This is only supported for MIL windowed displays and for auxiliary displays.



```
/* Allocate a display on a DMIL remote system, which displays image buffers
/* on a remote slave computer. */
MdispAlloc(RemoteSystemId, M_DEFAULT, MIL_TEXT("M_DEFAULT"), M_WINDOWED+M_REMOTE_DISPLAY,
&DisplayId);
MdispSelect(DisplayId, RemoteImageBufId);
```

When displaying on the master computer and the overlay-display mechanism is enabled, the display's overlay buffer is superimposed on the displayed buffer before the transfer.

By default, the display is updated after each modification to the displayed buffer (or the overlay buffer), and the application waits for the data to be transferred to the display before continuing. When a buffer, located on a remote computer, is displayed on the master computer, the update time can significantly slow your application. To reduce delays, you can display in asynchronous mode using **MdispControl()** with **M_ASYNC_UPDATE**. This mode uses multiple internal buffers to queue the updates and allows the application to continue once the update has been queued. This maximizes your bandwidth usage while minimizing processing delays.

To reduce transmission delays when the displayed buffer is modified, only the modified areas of the buffer are transmitted.

By default, MIL sends updates from the remote computer to the master computer as fast as possible, limited by the available bandwidth and transmission delay. However, when updating the display asynchronously, it is possible to set an upper limit to the number of updates to perform per second, using **MdispControl()** with **M_UPDATE_RATE_MAX**. When a limit is specified, you can assume that there will be a minimum delay of $1/\text{M_UPDATE_RATE_MAX}$ between two consecutive updates. Between transmissions, updates are accumulated into one update. Alternatively, whether updating the display synchronously or asynchronously, you can use **MdispControl()** with **M_UPDATE_RATE_DIVIDER** to skip updates. In this case, you specify after how many buffer modifications to update the display with the last modification, regardless of the time lapsed. This is especially useful when performing continuous grabs. When using **M_UPDATE_RATE_DIVIDER**, if fewer than the specified number of buffer modifications occur, the display will not be updated; whereas with **M_UPDATE_RATE_MAX**, the display will eventually be updated.

To reduce bandwidth usage, it is also possible to use compression so that the amount of data transmitted is minimal. You can activate compression using **MdispControl()** with **M_COMPRESSION_TYPE**. When enabled, data is compressed before being transmitted. The best compression rate is obtained with **M_JPEG_LOSSY**. The **M_JPEG2000_LOSSY** and **M_JPEG2000_LOSSLESS** compression types support much higher ratios of compression without

compromising image quality. However, you should avoid the JPEG2000 compression types unless absolutely necessary since compression/decompression time is high; you should only use them when the benefits of compression outweigh the overhead associated with the compression/decompression. For example, you might need to use the JPEG2000 compression types when there are fine details in your image or overlay buffer. For both JPEG and JPEG2000 lossy compression types, you can manually set the compression factor using **MdispControl()** with **M_Q_FACTOR**. By default, the quantization factor for the compression is set to 80, but you can specify an integer value in the range of 1 to 99. The higher the factor, the more the compression, but the image quality suffers.

To change the default settings, you can use the MILConfig utility on the master computer. If you select the **High quality** display option in MILConfig, the display is updated synchronously and compression is disabled by default; if you select the **Optimized for bandwidth usage** display option, the display is updated asynchronously and **M_JPEG_LOSSY** compression is used by default.

Best practice

When developing a Distributed MIL application, you should be aware of some network limitations and adhere to some good practices to develop the most efficient application.

With regards to network usage, you should be aware of the following points:

- Even if high bandwidth is available with Gigabit Ethernet, latency is inherent to network communication.
- Latency can add up quickly when multiple synchronous successive calls are made.
- Saturation of the network link increases latency.
- Collisions on the network link increases latency.

To implement an efficient Distributed MIL application, there are a few good practices that you should adhere to:

- Use a dedicated subnet for remote computers to minimize interference with other network traffic.
- Do not force the thread (**MthrControl()**) or system (**MsysControl()**) to be synchronous (**M_THREAD_MODE** set to **M_SYNCHRONOUS**).
- Use remote files as much as possible. Remote files can also be on a file server accessible by all remote computers.
- Avoid transmitting buffer data back and forth between computers.
- Avoid unnecessary inter-systems calls. Compensation means copy.
- For processing that involves lots of calls for each grabbed frame, consider a user-defined function (**MfuncAlloc()**).
- Allocate displays only if necessary.
- Use asynchronous mode for displays with the lowest possible frame rate.

Chapter

25

Multi-processing, multi-core, and multi-threading

This chapter describes how MIL handles multi-processing on a single core, processing on multiple cores, 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.

Systems without multi-processing

Not all systems support multi-processing. For example, a simple imaging board with only acquisition capability (such as Matrox Meteor-II) cannot ensure either the response time to a function or the independence of a process necessary for multi-processing. On such boards, two different processes cannot use the same system simultaneously. 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.

Systems with multi-processing

Systems that support multi-processing, such as Host or Matrox Odyssey, have on-board resources (such as, processors) that can be shared by different processes. However, multiple processes running at the same time have to share the available processing time, and cannot share data, since the memory used by one process is protected from all other processes.

Transparent multi-core use

When dealing with a system allocated on a computer with multiple CPU cores (processors), MIL can transparently split the processing work of a function between the cores available to the application. Typically, this division of labor can greatly increase the overall execution speed of each function within an application. Unlike multi-processing or multi-threading, multi-core processing works on one process or thread at a time.

When a function is executed by multi-cores, its work is split into multiple parts and transparently sent to multiple cores. When the work is done on all parts, the function returns and the result is made available.

The goal of multi-core processing is to increase the function's process speed. The following is a list of characteristics that can affect the speed gained using multiple core processing:

- **Original speed.** The splitting and merging of the work adds a certain amount of overhead to any function performed using multi-core processing. Therefore, if the original function is normally very fast, performing the function using multi-core processing might not provide a speed increase. This is often observed when processing small buffers. MIL tries to dynamically determine the best multi-core processing fit for the current function call to minimize the overhead; but it cannot be eliminated.
- **I/O access.** If the function has a high ratio of I/O access (such as, memory accesses) compared to the amount of actual processing, the function is I/O dependent. In such cases, performing the function using multi-core processing can cause all the cores involved to fight for access to the same resource (such as, memory). The result is a smaller gain than expected and sometimes even less performance than the original single-core execution.
- **Computer architecture.** There are a lot more variables affecting the performance of a multiple core Host computer than a traditional single processor Host computer. Each of these variables (for example, CPU, cache, bus, and memory) becomes more important in the final performance for each additional processor. Therefore a small change in the architecture of the Host computer can easily result in very different performance numbers.

You can enable and change the multi-core processing default settings using the MILConfig utility and **MappControl()** and **MthrControl()** functions.

Setting the maximum number of cores per thread

MIL multi-core processing allows you to manage CPU core utilization of each thread in a MIL application. By default, when enabled, multi-core processing spreads processing among all the available CPU cores. In most cases, this provides good performance gains without further configuration. In some cases when using multiple threads, spreading processing among all the available CPU cores might lead to multiple threads requesting the same CPU cores at the same time. This might reduce the overall efficiency of your MIL application.

There are two control types to help reduce this problem:

- **MappControl()** with **M_MP_MAX_CORES_PER_THREAD**. This control type sets the absolute maximum number of CPU cores for each thread.
 - **MthrControl()** with **M_MP_MAX_CORES**. This control type sets the maximum number of CPU cores available to the specified thread.
- ❖ Note that **M_MP_MAX_CORES** is limited by **MappControl()** with **M_MP_MAX_CORES_PER_THREAD**. If **M_MP_MAX_CORES** is greater than **MappControl()** with **M_MP_MAX_CORES_PER_THREAD**, the control value of **M_MP_MAX_CORES_PER_THREAD** is used.

You should only adjust these control types if using multiple threads and fine-tuning the performance of your MIL application. If the sum of the maximum number of CPU cores for each thread does not exceed the maximum number of CPU cores available to your MIL application, MIL will try to ensure that the threads use different CPU cores. Note that the total number of CPU cores available to your MIL application is always determined by your operating system and the number of CPU cores installed in your computer.

For example, if your application has access to eight CPU cores and it has two processing threads, with multi-core processing enabled, each thread will try to spread its multi-core processing among all eight cores. However, it might be more efficient to split the CPU cores between the two threads. If the threads have similar processing loads, you could restrict each thread to use four cores each, using **MappControl()** with **M_MP_MAX_CORES_PER_THREAD** set to four.

If one thread is processing-intensive while the other is not, you could give the processing-intensive thread access to the majority of the available CPU cores using **MthrControl()** with **M_MP_MAX_CORES**. For example, if your MIL application has access to 8 CPU cores, you could give your processing-intensive thread access to seven cores, and the other thread one. In this case, **MappControl()** with **M_MP_MAX_CORES_PER_THREAD** should be left to its default setting or be set no lower than seven. Otherwise, **M_MP_MAX_CORES_PER_THREAD** will restrict the processing thread to less CPU cores than specified using **M_MP_MAX_CORES**.

Steps to using multi-core processing

The following steps provide a basic methodology for using multi-core processing controls and inquires:

1. Enable multi-core processing using either the MILConfig utility or **MappControl()** with **M_MP_USE** set to **M_ENABLE**.
2. Optionally, set the absolute maximum number of cores to use per thread, using either the MILConfig utility or **MappControl()** with **M_MP_MAX_CORES_PER_THREAD**. This number should be at least as large as the largest number you plan to assign a thread in your application.

Note that the effective number of cores is always limited by the number of CPU cores available to your MIL application, as per your operating system and the physical number of CPU cores. To determine the number of CPU cores available to your MIL application, use **MappInquire()** with **M_MP_CORES_NUM**.

3. Optionally, disable multi-core processing for one or more threads using **MthrControl()** with **M_MP_USE** set to **M_DISABLE**.
4. Optionally, set the maximum number of cores to use for a specific thread using **MthrControl()** with **M_MP_MAX_CORES**. Note that this control type will have an effect only if it is less than **MappControl()** with **M_MP_MAX_CORES_PER_THREAD**.

To determine the effective number of CPU cores that a specific thread can use, use **MthrInquire()** with **M_MP_MAX_CORES_EFFECTIVE**.

- ❖ Note that, alternately you can disable multi-core processing at the application level (using **MappControl()** with **M_MP_USE** set to **M_DISABLE**) and enable it for a specific thread at the thread level (using **MthrControl()** with **M_MP_USE** set to **M_ENABLE**).

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 operations in other threads.

Threads within a process share the same data. Individual threads can communicate with each other 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.

Resource sharing

Multi-threading does not always result in an increase of speed and efficiency. Threads running simultaneously on the same CPU share the same system resources (such as memory). When using a machine with multiple CPUs under Windows, 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

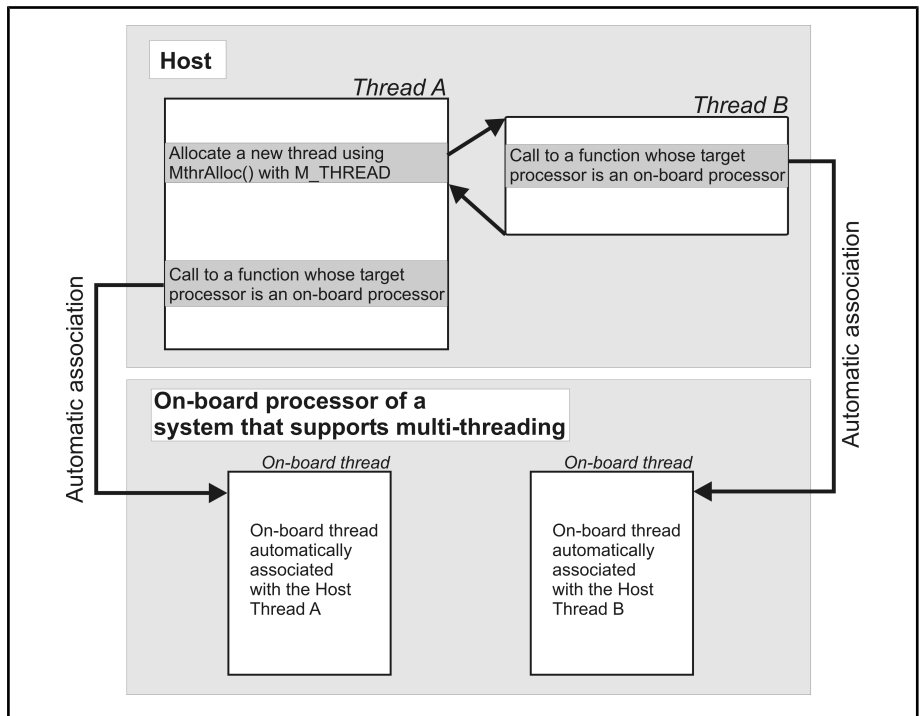
When your application contains several independent processing tasks that can be performed in parallel, you can design it so that each part is controlled by a separate thread (or task).

Creating threads using MIL

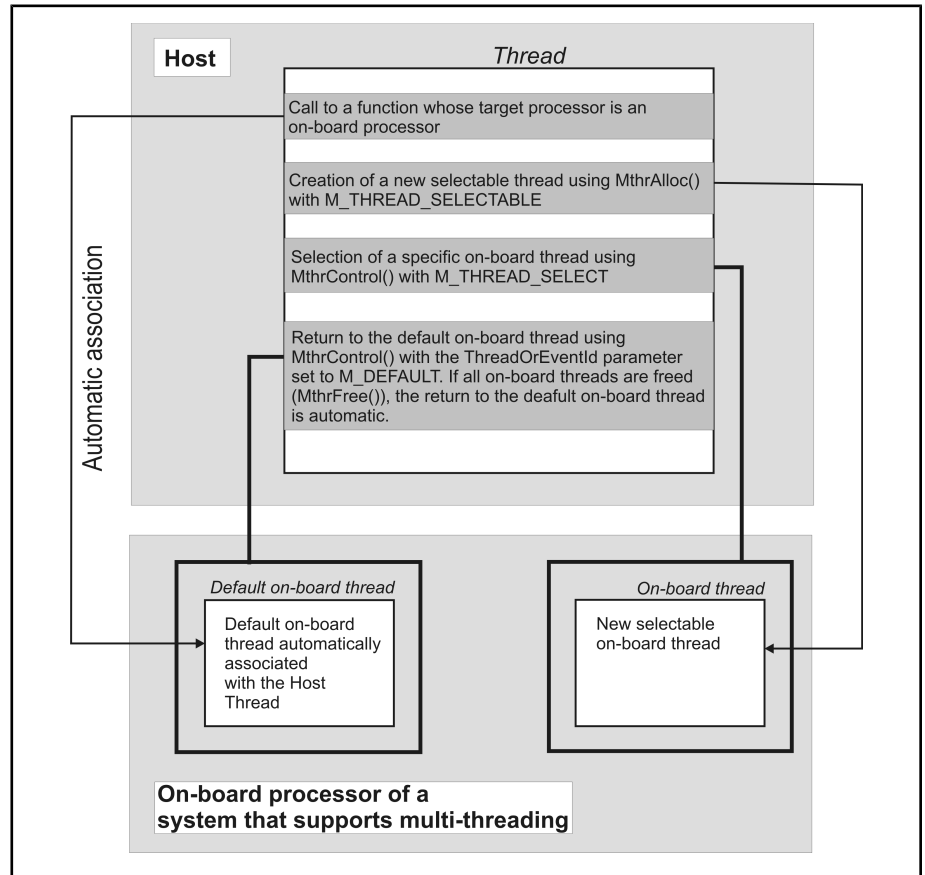
Under multi-thread operating systems, you can create as many threads as you require. You can create threads using commands provided by the operating system, or using the **MthrAlloc()** function provided by MIL. The **MthrAlloc()** function is portable, which means that it can be called from user-defined MIL functions that are executed on systems with on-board processors (for example, Matrox Odyssey). For information on executing functions on systems with on-board processors, see the *Master slave dynamics on a remote system* section in *Chapter 27: The MIL function development module*.

There are two methods for creating threads using the **MthrAlloc()** function:

- With the first method (**M_THREAD**), the **MthrAlloc()** function creates a MIL thread context for the new thread, and allows you to specify a pointer to a function that will be executed by the thread. This user-created function must include all operations, and call all of the functions that you consider as being part of one thread. When a thread contains a function call whose target processor is an on-board processor that supports multi-threading, MIL automatically creates a corresponding thread on that system's on-board processor. The functions of the Thread module allow you to synchronize threads running on the Host and/or various MIL systems.



- With the second method, **MthrAlloc()** creates a selectable thread (**M_SELECTABLE_THREAD**). Selectable threads are threads executed on an on-board processor that supports multi-threading and can be controlled from a single corresponding thread on the Host. Use **MthrControl()** with **M_THREAD_SELECT** to send MIL functions to be executed by a selectable thread.



MIL application context

For each new thread calling MIL functions, MIL creates a new default MIL application context and initializes it to the state of the main MIL application (the first application allocated with **MappAlloc()**). The application context allows the new thread to behave as an independent application, respecting all of MIL environment controls, such as error reporting.

To set the thread's application context to its own initial state (not necessarily the same as the main MIL application), allocate a new application using **MappAlloc()**. Note that the call to **MappAlloc()** must be the first MIL function call in the thread.

Thread execution

MIL functions in any thread are executed as follows:

- If the target processor is the Host CPU, processing in each thread is determined by the operating system.
- If the target processor is an on-board processor of a system that supports multi-threading (such as Matrox Odyssey), MIL automatically creates, and eventually terminates, an on-board thread for each thread that sends commands to the board.

Since the creation of on-board threads is done automatically, you do not have to specify the system on which to create a specific thread. However, you can do so by creating selectable threads on a particular system.

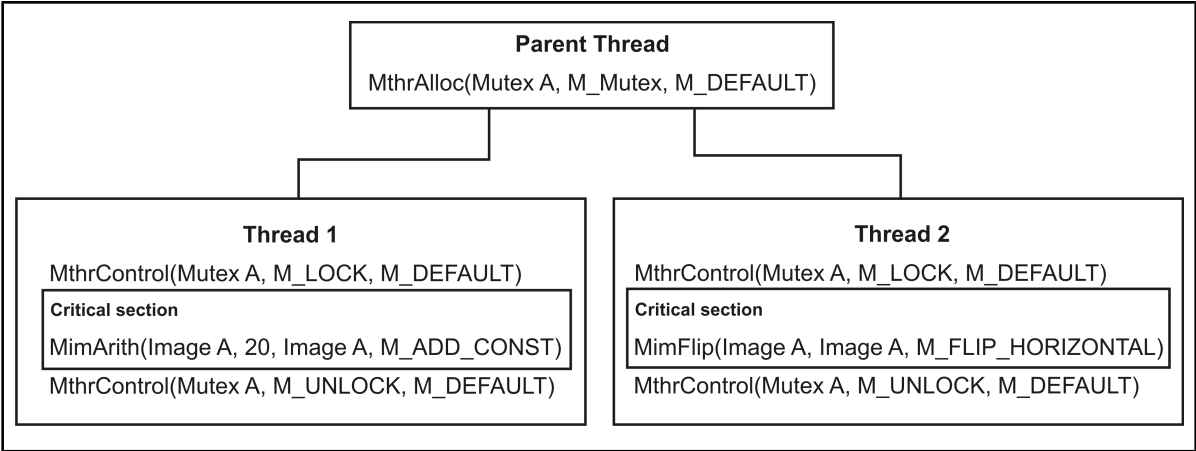
Synchronization and mutex

Thread synchronization is generally done using the Host synchronization services (such as Windows event objects). However, when using a system with an on-board processor, the activities of this processor cannot be synchronized by the Host.

This means that threads continue execution without waiting for the execution of the on-board functions to complete. In most cases, this behavior is acceptable, since it leaves the Host available for other tasks. However, for operations that require sequential execution of functions to return valid results (for example, **MbufGet()** after an **MdigGrab()**), MIL automatically synchronizes the threads, forcing the Host to wait for completion of the earlier function(s).

Explicit synchronization of threads is necessary if functions sharing a common resource might conflict with each other. For example, if two threads sharing the same image buffer are not synchronized, and each thread tries to clear the buffer to a different value, these functions might execute at the same time and the buffer could be cleared to either value or even to a combination of both. Use the synchronization features of the **MthrControl()**, **MthrWait()**, and **MthrWaitMultiple()** functions to synchronize the flow of threads. **MthrWait()** forces the current thread to wait for the completion of the specified thread or the change of state of the specified MIL event. **MthrWaitMultiple()** forces the current thread to wait for a change in state in one or all of the MIL events identified in a user-supplied array of events.

MthrControl() allows you to use mutexes to synchronize the flow of threads. A mutual exclusion object allows threads to synchronize access to shared resources. Once a mutex is allocated on the specified system, you can lock and unlock critical sections of code. Locking a critical section of code ensures that no two threads can access the same data at the same time. To lock a MIL mutex, you must call **MthrControl()** with **M_LOCK** or **M_LOCK_TRY** immediately preceding the section of code to lock. If you lock a mutex, you must unlock it at the end of the critical section of code using **MthrControl()** with **M_UNLOCK**. Once you are finished using the mutex, free it using **MthrFree()**. The use of a mutex is illustrated in the following example with **MimArith** and **MimFlip** accessing the same data.



MthrControl() with **M_LOCK** and **M_LOCK_TRY** are very similar; the difference occurs when one attempts to lock a mutex that is already locked. **MthrControl()** with **M_LOCK** will block the thread, forcing it to wait for the mutex to become unlocked before executing the critical section of code. **MthrControl()** with **M_LOCK_TRY** will not wait for the mutex to unlock. If the mutex is currently locked, the thread will proceed to execute, skipping the critical section of code. Note that once you call **M_LOCK_TRY**, you should immediately call **MthrInquire()** with **M_LOCK_TRY** to inquire whether the mutex was locked.

Thread control

Windows operating systems are multi-process and multi-thread operating systems. They provide various thread control services, including events (used to synchronize threads).

The **MthrAlloc()** function serves as a link between MIL and the operating system; the function allows you to create an operating-system-independent MIL version of these services. Threads and events created using **MthrAlloc()** can be used in addition to, or instead of, the events created using commands of the operating system.

MthrControl() controls and coordinates both threads and events. The **MthrWait()** function synchronizes thread processing by forcing a "wait" state. The **MthrInquire()** function inquires about both the settings of a MIL thread context and the state of an event allocated using **MthrAlloc()**. **MthrFree()** frees the allocated MIL thread context or event.

The **MthrAlloc()** function allows you to specify a particular system on which to allocate MIL thread contexts or events. This permits you to synchronize the execution of functions on a specific MIL system. Also, you can create events whose identifiers will be recognized by multiple systems with on-board processors, using **MthrAlloc()** with **M_EVENT_CREATE**.

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. Errors are logged once the function is executed on the processor of the specified system, and are reported immediately after being logged.

To check for errors, use the **MappGetError()** function. In multi-thread environments, a call to **MappGetError()** returns the last error that occurred in the current thread or, if none, checks for errors in other threads running MIL. To return only errors occurring in the current thread, add **M_THREAD_CURRENT** to the **ErrorType** parameter (**M_CURRENT+M_THREAD_CURRENT**).

Chapter

26

Distribution and licensing

This chapter presents how to distribute MIL or MIL-Lite and license MIL applications.

Distribution of MIL applications

There are some details that you must consider before you can distribute a MIL or MIL-Lite application, either for your customers' use or for your own. When doing this distribution, you must redistribute MIL/MIL-Lite DLLs and device drivers. In addition, when using a MIL processing module, you must purchase an appropriate runtime license; otherwise, your MIL application will not function correctly. MIL does not automatically install a license. The user must install the license, whether it is a provisional, development, or runtime license.

This chapter deals with the different ways to distribute your application, obtain a license, and handle licensing for your customers.

- ❖ Note that MIL-Lite applications will work if a Matrox Imaging board or a supplemental license is present. A MIL installation without any license will work as a MIL-Lite runtime installation if a Matrox Imaging board is present. A MIL installation without any license will work as a MIL-Lite development installation if a Matrox Odyssey board is present.

Redistributing MIL or MIL-Lite DLL files and device drivers with your application

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

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

Conversely, if the version of MIL or MIL-Lite on the computer is different from the application's required version, a decision must be made. The latest version of MIL or MIL-Lite should be kept or installed, otherwise the application cannot be installed on that computer. Note that the application might need to be recompiled with the installed version of MIL or MIL-Lite.

Redistributing directly from the MIL or MIL-Lite DVD

If the target computer (on which you want to install the MIL or MIL-Lite DLLs and device drivers) is immediately accessible, you can install the DLLs directly from the MIL or MIL-Lite DVD. To do so, run the MIL or MIL-Lite setup program and choose the MIL runtime option.

Redistributing using your own setup program

Alternatively, to redistribute the MIL or MIL-Lite DLLs and device drivers, you can have your application's setup program call MIL or MIL-Lite's redistribution setup program in one of two ways:

- **Interactive redistribution.** Prompts your customer for setup information.
- **Silent redistribution.** Uses a custom response file, instead of prompting your customer for information.

Interactive redistribution using your custom DVD

To redistribute the MIL or MIL-Lite DLLs and device drivers, you can have your application's setup program call MIL or MIL-Lite's redistribution setup program using the interactive redistribution mechanism. In this case, your customer will be prompted for information during the setup. There are two ways to implement this mechanism.

To redistribute the DLLs and device drivers and use the default installation options, do as follows:

1. Copy the contents of the MIL or MIL-Lite DVD to your installation directory.
2. Have your redistribution program call the *setup.exe* file, located in your installation directory.
3. The *setup.exe* program will prompt your customer to choose which features to install. Your customer should only select the **MIL Run-time** option. The **MIL Run-time** option installs the required MIL or MIL-Lite DLL files and device drivers on your client's computer.

To limit the contents of the MIL or MIL-Lite DVD to include in your installation directory and restrict the installation options, you can use the *Redist.exe* program. To use this program, do as follows:

1. Launch the *Redist.exe* program located in the *Redist* directory of your MIL or MIL-Lite DVD. The **Matrox Imaging Library Redistribution** dialog box is presented.
2. Specify the location of the source files to copy to your installation directory. To do so, click on the **Browse** button next to the **Source** edit field. From the presented **Open** dialog box, select the *setup.exe* file located in the root of your MIL or MIL-Lite DVD, and then click on the **Open** button.
3. Specify the location of your installation directory. To do so, click on the **Browse** button next to the **Destination** edit field. From the presented **Choose folder** dialog box, select your installation directory, and then click on the **Select** button.
4. Select the redistribution packages (MIL and/or Intellicam), the systems, and the drivers that you want to make available to your customer during installation. Note that if your customer is not using a hardware license-key or a hardware ID key, you can deselect the **SuperPro Plus (Hardware Key)** option to avoid copying the SuperPro Plus driver to your installation directory. In addition, if your customer does not have an on-board display section, you can deselect the **Matrox video card** option to avoid copying the MGA drivers to your installation directory.
5. Click on the **Generate** button to begin copying the limited installation version of MIL or MIL-Lite into your installation directory.
6. Have your redistribution program call the *setup.exe* file, located in the root of the specified installation directory.
7. The *setup.exe* program will prompt your customer to choose which features to install from the features that you have made available.

Silent redistribution

A silent redistribution does not prompt your user for any MIL or MIL-Lite 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 or MIL-Lite 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 or MIL-Lite DLLs and device drivers using a silent redistribution:

1. Copy the contents of MIL or MIL-Lite DVD to your installation directory. Alternatively, you can use the *Redist.exe* program to select the redistribution packages (MIL and/or Intellicam) and drivers to copy in your installation directory. For more details on how to use this program, see the *Interactive redistribution using your custom DVD* section earlier in this chapter.
2. Record your setup options. To do so, start the *setup.exe* program from the command line with the */r* switch and follow the steps for an interactive MIL or MIL-Lite redistribution. Any selected setting is recorded and stored in the *setup.iss* file. The *setup.iss* file will be created in the Windows folder (%WINDIR%) and must be moved to the root folder of your installation directory.
3. Have your redistribution program call the *setup.exe* program with the */s* switch. The setup will silently install MIL or MIL-Lite with the options that were selected during the recording phase and stored in the *setup.iss* file.

Uninstalling

To uninstall previous versions of MIL or MIL-Lite, and all other Matrox Imaging products, call the Matrox Imaging Products Uninstall program. To make uninstalling silent, use the silent uninstall program located in the root directory on the MIL or MIL-Lite DVD. For a silent uninstall, the following files must be present in the same directory:

- *silentuninstall.exe.*
- *setup.exe.*
- *uninstall.iss.*
- *Matrox Imaging.msi.*
- *ISSetup.dll.*

If you want to uninstall MIL and MIL-Lite in silent mode, change the working folder to the root directory or wherever you have copied the above files to and call:

```
silentuninstall.exe
```

or to reboot the computer after uninstalling, call:

```
silentuninstall.exe reboot
```

MIL and MIL-Lite licenses

MIL and MIL-Lite have different licensing terms and mechanisms:

- **MIL-Lite.** With MIL-Lite, there is a permanent development license (for MIL-Lite functionality) provided with Matrox Imaging hardware or when a MIL-Lite supplemental license is present.
- **MIL.** With MIL, there are two types of permanent licenses: a development license and/or a runtime 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 MIL provisional license while waiting for a permanent license.

Please refer to the respective Matrox software license agreement for the legal provisions of using/redistributing MIL or MIL-Lite. The rest of this chapter, with the exception of the last section, deals exclusively with MIL and its licensing mechanisms. The last section pertains to both MIL and MIL-Lite.

- ❖ Note that MIL users also receive the privilege to run (but not debug) MIL-Lite applications if Matrox boards are present.

MIL provisional licenses

MIL temporary license

This license is a 30-day runtime license only. It does not allow for development (debugging) of MIL applications.

The temporary license is only available on certain Matrox imaging hardware (it is preloaded on certain Matrox boards). The license is activated with the MilConfig application. The license is valid for 30 days from the date of activation: these 30 days will begin counting the moment the license is activated with MilConfig.

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.

MIL evaluation license

This license is a 30-day development license. The evaluation license permits you to run and debug MIL applications.

The evaluation license needs to be activated on the internet, and will be valid for 30 days from the date of issue.

To activate the evaluation license, register on the Matrox website (<http://www.matrox.com/imaging/support>) to receive an activation code. This code must be entered in the MIL License Manager application.

Once the evaluation 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 evaluation license expires will immediately disable MIL. In that event, MIL can only be re-used once an appropriate license is installed.

MIL permanent licenses**MIL development license**

A development license permits you to run and debug MIL applications. For the license to take effect, a development hardware license-key (dongle) must be attached to your computer's parallel or USB port (USB port only for Linux). A development hardware license-key is included with the MIL development package.

MIL runtime license

A runtime license is required for each computer that will run a MIL application. It does not allow for development (debugging) of MIL applications.

MIL runtime licenses are purchased according to the requirements of the application. A MIL runtime license grants access to specific modules/features, some of which are grouped together in packages. Consult the MIL datasheet, or your local representative, or Matrox Imaging sales regarding the available runtime packages.

A MIL runtime license can be in one of two formats:

- A hardware license-key (dongle).
- A software license-key.

If you wish, you can hide the MIL runtime licensing process from your customer using the Gencode utility, a text-based version of the MIL License Manager. Gencode is located in the same directory as the MIL License Manager. See the *Gencode utility* section later in this chapter.

Summary of activation procedures for all MIL licenses

The following table outlines the activation procedures required for different operating systems:

Type of MIL license	Activation procedure	
	For Windows XP/Vista and Linux	For Windows CE
Evaluation license	License is web-activated: register on Matrox website.	License is web-activated: register on Matrox website.
Temporary license	License is hardware-resident: activate license using MilConfig.	License is hardware-resident: activate license using portal page.
Development license	Requires hardware license key (dongle).	See note below table.
Runtime license	Requires hardware license key (dongle) or software license key.	See note below table.

- ❖ Under Windows CE, there is only one permanent license: a runtime license. When the user purchases this license, the user receives both development and runtime privileges. To activate this license, the user must have a software license key.

MIL-Lite licenses

MIL-Lite development license

MIL-Lite comes with a permanent development license (for MIL-Lite functionality) provided Matrox Imaging hardware or a MIL-Lite supplemental license is present.

MIL-Lite supplemental license

MIL-Lite supplemental licenses are purchased according to the requirements of the application. A MIL-Lite supplemental license grants access to specific modules/features, some of which are grouped together in packages. Consult the MIL datasheet, or your local representative, or Matrox Imaging sales regarding the available packages.

The following MIL-Lite supplemental licenses are available:

- GPU.
- Distributed MIL.
- Compression/decompression.
- 1394/GigE Vision.

A MIL-Lite supplemental license can be in one of two formats:

- A hardware license-key (dongle).
- A software license-key.

If you wish, you can hide the MIL-Lite supplemental license process from your customer using the Gencode utility, a text-based version of the MIL License Manager. Gencode is located in the same directory as the MIL License Manager. See the *Gencode utility* section later in this chapter.

Summary of MIL and MIL-Lite permanent licenses

The following table outlines the permanent license requirements for different MIL and MIL-Lite activities:

Activity	Requirement	Location
Developing a MIL application	MIL development license.	On the master computer.
Running a MIL application	MIL runtime license.	On each computer running a MIL application.
Developing a MIL-Lite application	Matrox imaging board or a MIL-Lite supplemental license.	On the master computer.
Running a MIL-Lite application	Matrox imaging board or a MIL-Lite supplemental license.	On each computer running a MIL-Lite application.

- ❖ For Distributed MIL, each computer running a MIL application must have a Distributed MIL runtime license.
- ❖ For Distributed MIL, each computer running a MIL-Lite application must have a Distributed MIL supplemental license.

Hardware license-key

A hardware license-key is a type of runtime license. For the license to take effect, the key must be attached to the parallel or USB port of your computer (USB port only for Linux).

If you decide that another computer needs to run a MIL application, you can transfer the hardware license-key. This will allow the second computer to use MIL, but the first computer will have MIL disabled.

To have MIL DLLs recognize the hardware license-key, you must install the SuperPro Plus driver. To install this hardware license-key driver, if you did not install it when you were installing MIL, call the `\Matrox Imaging\Drivers\SuperProDriver\wdm\RainbowSSD.exe` file and select the **Install Sentinel Driver** menu command.

For more information about Sentinel driver installation, refer to the *DriverInfo.htm* file located in the `\Matrox Imaging\Drivers\SuperProDriver` directory. This file also includes information on how to install the Sentinel driver silently.

Software license-key

A software license-key is another type of runtime 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 application on the target computer as described in the *Installation* section in *Chapter 1: Introduction*.
 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 hardware component that you have chosen, you will need a new license.

For easier portability or if your computer does not have any hardware component that acts as a fingerprint, you can purchase a hardware ID-key and use it as your hardware component 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 software license. If you attach the hardware ID-key to a different computer, the runtime 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 include other MIL packages; a hardware ID-key allows you to choose the package you want, unlike a hardware license-key.

4. After you have generated a lock code, the software license-key can be obtained by contacting your local representative or Matrox Imaging sales.
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 licensing process

It is possible to make the presence of MIL licensing non-invasive so that your customer can install your MIL application and not have to deal with the MIL licensing process.

Perhaps the simplest way to hide the MIL licensing process 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

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 Solios board, you could hardcode Gencode's command-line parameter **SystemFingerprint** to the constant that means Matrox Solios.

However implemented, the setup program of your application must internally call Gencode as follows to generate the lock code (the lock code will be stored in the file **Filename**):

```
Gencode /G Filename SystemFingerprint Packages
```

See the *Gencode utility* section later in this chapter for details.

Getting a software license-key

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 by contacting your local representative or Matrox Imaging Sales.
3. Your customer must input the software license-key into whatever activation interface you have created.

Entering the software license-key

Your activation interface must again call Gencode, this time as follows:

```
Gencode /L LicenseCode
```

and input the license code. This will activate your MIL license.

Gencode's command-line parameters are described in the *Gencode utility* section later in this chapter.

Gencode utility

This section describes the Gencode utility.

Synopsis

Generates a lock code for licensing, and activates a software license-key for MIL.

Prototype and description

The following is a table of the functions of the Gencode utility:

Prototype	Description
GENCODE /G <Filename> <SystemFingerprint> <Packages>	With the /G switch, it generates a lock code that is necessary for getting a software license-key from Matrox.
GENCODE /H	With the /H switch, it displays a help screen.
GENCODE /I	With the /I switch, it prints the numeric value of the available settings for the SystemFingerprint parameter.
GENCODE /R<SoftwareKey>	With the /R switch, it registers the license-key for running MIL on the current computer.

Parameters

The following is a list of the parameters available:

- The **Filename** 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 exists, the lock code information will be appended to the existing file. The lock code can also be retrieved from the registry under:

*HKEY_LOCAL_MACHINE\SOFTWARE\Matrox\Matrox Imaging
Library\Runtime\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 settings	Description
3	Matrox hardware license-key.
14	Matrox Solios GigE.
16	Matrox Morphis QxT.
17	Matrox Nexis.
18	Matrox Vio.
19	Matrox 4Sight M.
20	Matrox Solios.
21	Matrox Concord F-Series (IEEE 1394b adapter).
23	Matrox Corona-II.
27	Matrox CronosPlus.
28	Matrox Helios.
29	Matrox Morphis.
30	Matrox Iris.
33	Matrox Concord G-Series (GigE Vision).
34	Matrox Meteor-II /Multi-Channel.
50	Matrox ID-key (no license key storage).
51	Matrox hardware ID-key plus (with license key storage).
62	Any Matrox board.

If the specified board is not on the target computer, 0 is returned.

- The **Packages** parameter identifies which MIL packages are going to be obtained. This is something you would likely set for your customer, since you know which MIL modules your application uses.

The available MIL packages correspond to the following numeric values. If you need to obtain 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 package.
2	Machine Vision package.
4	Identification package.
8	Compression/Decompression package.
16	Geometric Model Finder package.
32	Distributed MIL package.
64	GPU Processing package.
128	Color Analysis package.
256	Edge Finder package.
512	Interface (IEEE 1394 and GigE Vision) using third-party hardware.
1024	String Reader package.
2048	Registration package.
4096	Metrology package.
8192	3D Calibration and Reconstruction package.

The one exception is that the Image Analysis and Machine Vision packages cannot both be included in the value of the **Packages** parameter. Refer to the Matrox Imaging website for more information on these packages.

- ❖ The 3D Calibration and Reconstruction package is only required when working in Tsai-based or robotics calibration mode.
- The **SoftwareKey** parameter is the software license-key code that you receive from Matrox or your local representative to finalize registration.

Protecting your own MIL software application using a Matrox hardware fingerprint

You might want to protect your own MIL software application using a third-party licensing package that requires a hardware fingerprint. In this case, you can use the hardware fingerprint of the Matrox Imaging board that your application uses. The MIL hardware fingerprint is a numerical value, unique to the Matrox hardware component.

You can inquire about the fingerprint of your Matrox Imaging board using **MappInquire()** with **M_..._FINGERPRINT**. For example, to use the hardware fingerprint of your Matrox Morphis QxT board, use **M_MORPHIS_QXT_FINGERPRINT**.

If the inquired Matrox hardware component is present, the requested hardware fingerprint is returned. If it is not present, 0 is returned.

Chapter

27

The MIL function development module

This chapter discusses the purpose of the MIL Function Development module and how to integrate your own functions into MIL.

MIL Function Development module

The MIL Function Development module allows programmers to define custom (user-defined) MIL functions to extend MIL's functionality. Using this module, you can implement functions and integrate them directly into the MIL library, where they behave like standard MIL functions (for example, respecting error handling and tracing). The MIL Function Development module also allows programmers to group related user-defined MIL functions together into user-defined modules. This is useful to create high-level packages on top of MIL and to extend the MIL library function set (for example, by adding new functions with specialized algorithms).

To allow remote execution of user-defined MIL functions, they are created in two parts: a master function and a slave function. The master function performs the parameter registration and transfers the information to the slave function. The slave function receives the parameter information and performs the data processing operations of the user-defined MIL function. Since the slave function is executed separately from the master function, when a remote processor is available, the slave function of a user-defined MIL function that meets certain criteria can be executed remotely by this remote processor. For the purposes of this chapter, the processor which executes the master function is referred to as the master processor and the processor which executes the slave function is referred to as the slave processor. When no remote processor is available the master and slave processors are the Host processor.

The MIL Function Development module also provides a framework by which you can define your own objects and associate them with MIL identifiers. Such associations can be useful when designing objects, within a user-defined MIL function, whose data members should not be accessed directly and whose data structure has specific requirements.

When designing a function, or set of functions, the MIL Function Development module puts at your disposal all the tools necessary to create custom error codes and error messages that are treated as regular MIL errors which can in turn be managed using the **Mapp...** functions.

Steps to create a user-defined MIL function

The following steps provide a basic methodology for using the MIL Function Development module:

1. Create a master function. The name that you will use to call the user-defined MIL function from an application is the same as the name of the master function.
2. Allocate a function context for the new function using **MfuncAlloc()**. The **MfuncAlloc()** function should be the first MIL function called in the master function. The allocated function context provides a MIL identifier for your user-defined MIL function, and allows this function to be treated as a regular MIL function (for example, error checking and tracing will be performed).
3. Register the parameters of the master function in the function context using the appropriate **MfuncParam...** functions. Registering the parameters gives the slave function access to the parameter values. You should register all parameters of the master function using the **MfuncParam...** function corresponding to the appropriate data type.
4. Call the slave function from the master function, using **MfuncCall()**. The slave function must be created as a separate function. Note that you must call **MfuncCall()** from the same thread as **MfuncAlloc()**.
5. Free the function context. At the end of the master function, the created function context must be freed, using the **MfuncFree()** function. This should be the last MIL function called in the master function. Note that you must call **MfuncFree()** from the same thread as **MfuncAlloc()** and **MfuncCall()**.
6. Create the slave function. This function must accept the MIL identifier of the function context as its only parameter. See the description of **MfuncCall()** for a prototype of a slave function. In the slave function, you must recuperate the values of the parameters registered in the master function, using **MfuncParamValue()**, and develop the code that will produce the required results. You can make calls to other MIL functions from the slave function.

Once you have performed the steps listed above, the user-defined MIL function is created, and you can use it in an application.

The following example shows how to create a user-defined MIL function.

```

/*****
/*
* File name: MFunc.cpp
*
* Synopsis: This example shows the use of the MIL Function Developer Tool Kit and how
*           MIL and custom code can be mixed to create a custom MIL function that
*           accesses the data pointer of a MIL buffer directly in order to process it.
*
*           The example creates a Master MIL function that registers all the parameters
*           to MIL and calls the Slave function. The Slave function retrieves all the
*           parameters, gets the pointers to the MIL image buffers, uses them to access
*           the data directly and adds a constant.
*
*           Note: The images must be 8-bit unsigned.
*/
#include <mil.h>

/* MIL function specifications. */
#define FUNCTION_NB_PARAM          3
#define FUNCTION_OPCODE_ADD_CONSTANT 1
#define FUNCTION_PARAMETER_ERROR_CODE 1
#define FUNCTION_SUPPORTED_IMAGE_TYPE (8+M_UNSIGNED)

/* Target image file name. */
#define IMAGE_FILE    M_IMAGE_PATH MIL_TEXT("BoltsNutsWashers.mim")

/* Master and Slave MIL functions declarations. */
void AddConstant(MIL_ID SrcImageId, MIL_ID DstImageId, MIL_INT ConstantToAdd);
void MFTYPE SlaveAddConstant(MIL_ID Func);

/* Master MIL Function definition. */
/* ----- */

void AddConstant(MIL_ID SrcImageId, MIL_ID DstImageId, MIL_INT ConstantToAdd)
{
    MIL_ID    Func;

    /* Allocate a MIL function context that will be used to call a target
       Slave function locally on the Host to do the processing.
    */
    MfuncAlloc(MIL_TEXT("AddConstant"),
               FUNCTION_NB_PARAM,
               SlaveAddConstant, M_NULL, M_NULL,
               M_USER_MODULE_1+FUNCTION_OPCODE_ADD_CONSTANT,

```

```

        M_LOCAL,
        &Func
    );

    /* Register the parameters. */
    MfuncParamId(    Func, 1, SrcImageId, M_IMAGE, M_IN);
    MfuncParamId(    Func, 2, DstImageId, M_IMAGE, M_OUT);
    MfuncParamMilInt(Func, 3, ConstantToAdd);

    /* Call the target Slave function. */
    MfuncCall(Func);

    /* Free the MIL function context. */
    MfuncFree(Func);
}

/* MIL Slave function definition. */
/* ----- */

void MFTYPE SlaveAddConstant(MIL_ID Func)
{
    MIL_ID    SrcImageId, DstImageId;
    MIL_INT    ConstantToAdd, TempValue;
    unsigned char *SrcImageDataPtr, *DstImageDataPtr;
    MIL_INT    SrcImageSizeX, SrcImageSizeY, SrcImagePitchByte;
    MIL_INT    DstImageSizeX, DstImageSizeY, DstImageType, DstImagePitchByte;
    MIL_INT    x, y;

    /* Read the parameters. */
    MfuncParamValue(Func, 1, &SrcImageId);
    MfuncParamValue(Func, 2, &DstImageId);
    MfuncParamValue(Func, 3, &ConstantToAdd);

    /* Lock buffers for direct access. */
    MbufControl(SrcImageId, M_LOCK, M_DEFAULT);
    MbufControl(DstImageId, M_LOCK, M_DEFAULT);

    /* Read image information. */
    MbufInquire(SrcImageId, M_HOST_ADDRESS, &SrcImageDataPtr);
    MbufInquire(SrcImageId, M_SIZE_X,      &SrcImageSizeX);
    MbufInquire(SrcImageId, M_SIZE_Y,      &SrcImageSizeY);
    MbufInquire(SrcImageId, M_TYPE,        &SrcImageType);
    MbufInquire(SrcImageId, M_PITCH_BYTE,  &SrcImagePitchByte);
    MbufInquire(DstImageId, M_HOST_ADDRESS, &DstImageDataPtr);
    MbufInquire(DstImageId, M_SIZE_X,      &DstImageSizeX);
    MbufInquire(DstImageId, M_SIZE_Y,      &DstImageSizeY);
    MbufInquire(DstImageId, M_TYPE,        &DstImageType);
    MbufInquire(DstImageId, M_PITCH_BYTE,  &DstImagePitchByte);

    /* Reduce the destination area to process if necessary. */
    if (SrcImageSizeX < DstImageSizeX)    DstImageSizeX = SrcImageSizeX;
    if (SrcImageSizeY < DstImageSizeY)    DstImageSizeY = SrcImageSizeY;

```

```

/* If images have the proper depth, perform the operation using a custom C code. */
if ((SrcImageType == DstImageType) && (SrcImageType == FUNCTION_SUPPORTED_IMAGE_TYPE))
{
    /* Add the constant to the image. */
    for (y= 0; y < DstImageSizeY; y++)
    {
        for (x= 0; x < DstImageSizeX; x++)
        {
            /* Calculate the value to write. */
            TempValue = (MIL_INT)SrcImageDataPtr[x] + (MIL_INT)ConstantToAdd;

            /* Write the value if no overflow, else saturate. */
            if (TempValue <= 0xff)
                DstImageDataPtr[x] = (unsigned char)TempValue;
            else
                DstImageDataPtr[x] = 0xff;
        }

        /* Move pointer to the next line taking into account the image's pitch. */
        SrcImageDataPtr += SrcImagePitchByte;
        DstImageDataPtr += SrcImagePitchByte;
    }
}
else
{
    /* Report a MIL error. */
    MfuncErrorReport(Func,M_FUNC_ERROR+FUNCTION_PARAMETER_ERROR_CODE,
        MIL_TEXT("Invalid parameter."),
        MIL_TEXT("Image type not supported"),
        MIL_TEXT("Image depth must be 8-bit."),
        M_NULL
    );
}

/* Unlock buffers. */
MbufControl(SrcImageId, M_UNLOCK, M_DEFAULT);
MbufControl(DstImageId, M_UNLOCK, M_DEFAULT);

/* Signal to MIL that the buffers are now in cache (required by Odyssey). */
MbufControl(SrcImageId, M_CACHE_CONTROL, M_IN_CACHE);
MbufControl(DstImageId, M_CACHE_CONTROL, M_IN_CACHE);

/* Signal to MIL that the destination buffer was modified. */
MbufControl(DstImageId, M_MODIFIED, M_DEFAULT);
}

/* Main to test the custom function. */
/* ----- */

int MosMain(void)
{

```

```

MIL_ID MilApplication,    /* Application identifier.    */
      MilSystem,         /* System identifier.        */
      MilDisplay,        /* Display identifier.       */
      MilImage;          /* Image buffer identifier.   */
void* ImageHostAddress;  /* Image buffer host address.*/

/* Allocate default application, system, display and image. */
MappAllocDefault(M_SETUP, &MilApplication, &MilSystem,
                 &MilDisplay, M_NULL, M_NULL);

/* Load source image into a Host memory image buffer. */
MbufAlloc2d(MilSystem,
            MbufDiskInquire(IMAGE_FILE, M_SIZE_X, M_NULL),
            MbufDiskInquire(IMAGE_FILE, M_SIZE_Y, M_NULL),
            8+M_UNSIGNED,
            M_IMAGE+M_DISP+M_HOST_MEMORY,
            &MilImage);
MbufLoad(IMAGE_FILE, MilImage);

MbufControl(MilImage, M_LOCK, M_DEFAULT);
MbufInquire(MilImage, M_HOST_ADDRESS, &ImageHostAddress);
MbufControl(MilImage, M_UNLOCK, M_DEFAULT);

if(ImageHostAddress != M_NULL)
{
    /* Display the image. */
    MdispSelect(MilDisplay, MilImage);

    /* Pause. */
    MosPrintf(MIL_TEXT("\nMIL FUNCTION DEVELOPER'S TOOLKIT:\n"));
    MosPrintf(MIL_TEXT("-----\n\n"));
    MosPrintf(MIL_TEXT("This example creates a custom MIL function that processes\n"));
    MosPrintf(MIL_TEXT("an image using its data pointer directly.\n\n"));
    MosPrintf(MIL_TEXT("Target image was loaded.\n"));
    MosPrintf(MIL_TEXT("Press a key to continue.\n\n"));
    MosGetch();

    /* Process the image directly with the custom MIL function. */
    AddConstant(MilImage, MilImage, 0x40);

    /* Pause */
    MosPrintf(MIL_TEXT("The white level of the image was augmented.\n"));
}
else
{
    MosPrintf(MIL_TEXT("This example cannot be run with this system.\n"));
}

```

```
MosPrintf(MIL_TEXT("Press a key to terminate.\n\n"));
MosGetch();

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

return 0;
}
```

Basic concepts

The basic concepts and vocabulary conventions for the MIL Function Development module are:

- **Asynchronous function.** A function that returns control to the calling thread before it has finished executing.
- **Distributed processing.** Distributed processing is a processing method which involves using more than one processor to perform the required operations. Execution of a function by a remote processor requires special consideration during compilation.
- **Master function.** In a user-defined MIL function, the master function provides the user interface. The slave function is called from within the master function.
- **Master processor.** The master processor is the processor on which the master function is executed.
- **Opcode.** All MIL functions, including user-defined functions, are associated to a unique operation code. Operation codes are, in turn, associated to pointers to their corresponding functions. The operation code is then used to refer to the function indirectly. For simplicity, operation codes are referred to as opcodes.

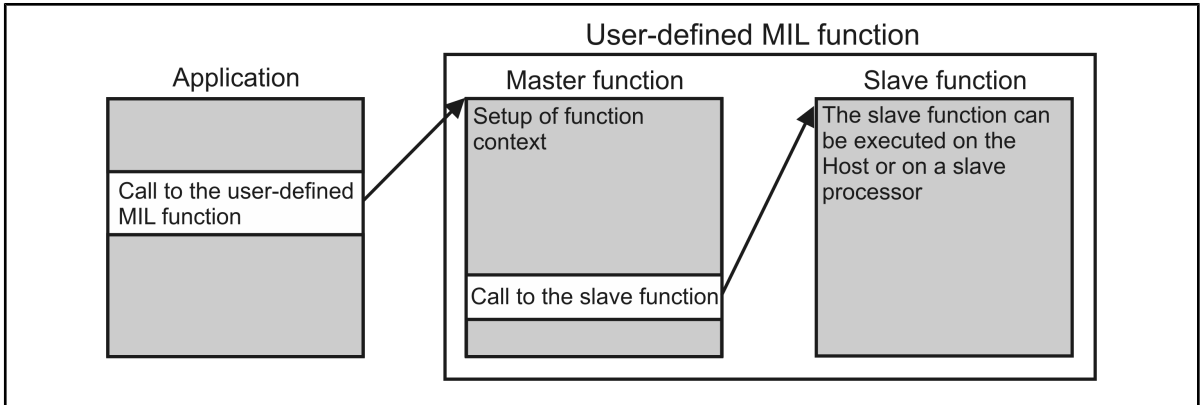
- **Opcode table.** The opcode table is a table in which on-board processors store the associations between opcodes and pointers to MIL functions.
- **Remote processor.** A remote processor is any available processor which is separate from the Host processor.
- **Slave function.** In a user-defined MIL function, the slave function is called by the master function. The slave function performs the data processing operations of the user-defined MIL function.
- **Slave processor.** The slave processor is the processor on which the slave function will be executed. The slave processor can be either the same processor as the master processor or a remote processor.
- **Synchronous function.** A function that returns control to the calling thread only after it has finished executing.

Characteristics of a user-defined MIL function

Each user-defined MIL function consists of a master and a slave function. The master function provides the user interface and sets up the MIL function context for the new function. In the master function, you must allocate the function context for the new function, register the parameters declared by the master function as parameters of the user-defined MIL function, call the slave function, and, in the end, free the allocated MIL identifier of the function.

The slave function is a separate function that actually performs the required operations when the user-defined MIL function is called. The slave function must be implemented as a separate function because it can be executed remotely on a system with a remote processor, such as Matrox Odyssey, or by a slave node in a Distributed MIL application. The only parameter that is passed to the slave

function is the MIL identifier of the function context. The slave function uses this identifier to access the MIL function context and retrieve the parameters from the context using the **MfuncParamValue()** function. You can also make calls to other MIL functions from the slave function.



- ❖ Note that to increase efficiency, parameter checking is not performed for user-defined MIL functions.

Remote and local functions

When allocating your function context using **MfuncAlloc()** in the master function, you must specify whether your user-defined MIL function will be executed on the master computer or the remote computer with **M_LOCAL** or **M_REMOTE**.

User-defined MIL functions allocated as **M_LOCAL** will be executed by the Host processor; whereas user-defined MIL functions allocated as **M_REMOTE** will be executed by a remote processor, if one is available.

Asynchronous and synchronous functions

When allocating your function context using **MfuncAlloc()** in the master function, you must specify whether your user-defined MIL function will run asynchronously or synchronously (**M_ASYNCHRONOUS_FUNCTION** or **M_SYNCHRONOUS_FUNCTION**) with respect to the calling thread.

User-defined MIL functions allocated to run asynchronously will return control to the calling thread immediately after being called. One advantage of running functions asynchronously is that they allow the master function to immediately proceed to the next statement, while the slave processor executes the slave function.

Note that an asynchronous function can modify MIL buffers and parameters passed by reference, but cannot actually return a value. For more information on return values, see the *Return values* subsection in the *Parameter registration and return values* section in *Chapter 27: The MIL function development module*.

- ❖ Note that user-defined MIL functions allocated to run on the Host processor must be run synchronously and user-defined MIL functions allocated to run on the remote processor can be run asynchronously or synchronously.

Modules, opcodes, and error messages

User-defined MIL functions must be associated to a user-defined module; related functions are typically grouped into the same user-defined module. You can create up to 7 user-defined modules, each of which can contain up to 32 user-defined MIL functions. A function's opcode indicates both its module and its offset within the module; it is specified as `M_USER_MODULE_n+m`, where `n` specifies the user-defined module and `m` specifies the function's offset within the module (`MfuncAlloc()` with `SlaveFunctionOpcode`).

A function's opcode must be unique. The uniqueness of the function's opcode is particularly important when retrieving error codes since the function that returned an error is identified by its opcode. The uniqueness is also critical when executing the slave function remotely; see the *Master/slave dynamics on a remote system* section later in this chapter.

User defined error codes

When developing user-defined MIL functions, `MfuncErrorReport()` allows you to create custom error codes and error messages which are treated as normal MIL errors. `MfuncErrorReport()` also allows you to associate each error code to an error message; up to one hundred such custom error code associations can be defined within MIL. To specify whether errors will be reported to the screen, use `MappControl()` with `M_ERROR`; get error information using `MappGetError()`.

Parameter registration and return values

Parameters must be registered in the master function with the **MfuncParam...** function that matches the data type of the parameter (for example, a parameter of type long should be registered using **MfuncParamLong()**); the values of the parameters passed to the master function must then be retrieved from within the slave function using **MfuncParamValue()**.

Certain parameter registration functions require additional details about the parameter being registered:

- When registering parameters of type MIL_ID (using **MfuncParamId()** or **MfuncParamIdPointer()**), you must specify the type of MIL object.
- When registering parameters that accept strings or pointers (using **MfuncParamString()** or **MfuncParamPointer()**), you must specify the size of the data, in bytes.
- When registering parameters of type MIL_ID and parameters that accept strings or pointers (using **MfuncParamId()**, **MfuncParamIdPointer()**, **MfuncParamString()**, or **MfuncParamPointer()**), you must specify whether the parameter will serve as input (**M_IN**), output (**M_OUT**), or both (**M_IN + M_OUT**) for your function. Your slave function should use output parameters to return results (for example, the destination buffer of an operation should be passed as an output parameter).

The **M_IN** and **M_OUT** attributes of the **MfuncParam...** functions do not serve to limit the slave function's ability to modify variables; rather, the **M_OUT** attribute serves to indicate to MIL that when the slave function finishes, MIL must synchronize the slave function's data for that parameter with the rest of the application. Identifying output parameters is particularly important when the slave function is being executed by a remote processor which does not have direct access to the Host's address space. In this case, when the slave function terminates, MIL will internally pass the new values for the output type parameters to the Host. If the content of a buffer is modified in a slave function executed by a remote processor, and the buffer was not registered as an output parameter, your changes will be lost when the slave function finishes.

From within the slave function, you must declare a new variable for each parameter registered in the master function; the data type of each new variable must match the data type of its associated parameter. For example, a parameter of type `MIL_ID` must be registered using `MfuncParamId()` in the master function; then, in the slave function, you must declare a new variable of type `MIL_ID` and use `MfuncParamValue()` to associate the new variable with the value passed as a parameter to the master function.

- ❖ Note that, if you are using an output pointer parameter, your function context must be allocated to run synchronously.

Return values

Only synchronous user-defined MIL functions (`MfuncAlloc()` with `M_SYNCHRONOUS_FUNCTION`) can have a return value. To have a user-defined MIL function return a value, the return value must be generated in the slave function and returned by the master function. However, the slave function of a user-defined MIL function must be declared as being of type `void MFTYPE`; therefore, slave functions can't directly return a value. To return a value, the master function must declare a variable (the return variable) and register a pointer to it as if it was a real `M_OUT` parameter of the master function. The slave function should treat the return variable as a regular `M_OUT` parameter.

Note that the number of parameters specified during context allocation in a user-defined MIL function (`MfuncAlloc()`) limits the number of parameters that the slave function can retrieve. If a user-defined MIL function has a return value, the number of parameters must be set to one greater than is actually passed to the master function. The address of the return variable must be registered as a pointer parameter in the master function (`MfuncParamPointer()` or `MfuncParamIdPointer()` with `Attribute` set to `M_OUT`). Then, in the slave function, you must declare a pointer variable of the same type as the return variable and associate the pointer with the address of the return variable using `MfuncParamValue()`. Recall that, the slave function does not actually store the return value at the memory address provided by the master function; it provides the master function with the new value to be written to memory.

The following portion of MIL code shows the master function of a user-defined MIL function with a return value:

```
int MyFunc( MIL_ID SrcImage ) {
    MIL_ID MyFunctionContext;
    int ReturnValue = 0;

    MfuncAlloc( MIL_TEXT("MyFunc"), 2, MyFuncSlave, M_NULL, M_NULL,
                M_USER_MODULE_1+1, M_SYNCHRONOUS_FUNCTION+M_LOCAL,
                &MyFunctionContext );

    MfuncParamId( MyFunctionContext, 1, SrcImage, M_IMAGE, M_IN+M_PROC );

    MfuncParamPointer( MyFunctionContext, 2, &ReturnValue, sizeof(ReturnValue),
                       M_OUT );

    MfuncCall( MyFunctionContext );

    MfuncFree( MyFunctionContext );

    return ReturnValue;
}
```

The following portion of MIL code shows how to retrieve the address of the return variable and use it to store the return value (in the slave function):

```
void MFTYPE MyFuncSlave( MIL_ID Func ){
    MIL_ID SrcImage;

    int *ReturnValuePtr;
    int ValueToReturn = 0;

    MfuncParamValue( Func, 1, &SrcImage );
    MfuncParamValue( Func, 2, &ReturnValuePtr );

    /* Calculate the return value */
    // ValueToReturn = ...

    *ReturnValuePtr = ValueToReturn;
}
```

Master/slave dynamics on a remote system

When you allocate your function context in the master function (**MfuncAlloc()**), you must provide an opcode for the slave function. If the slave function will be executed on the master computer, you must provide a pointer to the function using **MfuncAlloc()** with **SlaveFunctionPtr**, but if your slave function will be executed by the remote system in a Distributed MIL application you must provide the name of the library file and the name of the function within the library file using **MfuncAlloc()** with **SlaveFunctionDLLName** and **SlaveFunctionName**, respectively. You must also specify, using the **InitFlag** parameter, if the slave function must be run on the master computer (**M_LOCAL**) or, when possible, on the remote computer (**M_REMOTE**). **MfuncCall()** uses the pointer to call the slave function when it is to be executed on the master computer, and uses either the opcode or the library file to call the slave function when it is to be executed on the remote computer. Even if the slave function will be executed on the master computer, the opcode is still required for error reporting purposes.

For more information about Distributed MIL and its applications, see *Chapter 24: Distributed MIL*.

Compilation

Depending on the scope requirements of your user-defined function, its master function and its slave function will need to be compiled differently, according to how they will be called and how you want them to be executed. Every user-defined MIL function will have different scope requirements depending on the application in which it is being used. There are 5 possible execution scenarios for a user-defined MIL function:

- The user-defined MIL function will be called and executed by the Host.
- The user-defined MIL function will be called by the Host and executed by the remote system.

In this scenario, the master function should be compiled for the Host and the slave function should be compiled for the slave processor or into a library file, depending on the type of remote system executing the slave function.

- The user-defined MIL function will be called and executed by the remote system.

In this scenario, the master and slave functions should be compiled for the slave processor or into a library file, depending on the type of remote system executing the slave function.

- The user-defined MIL function might be called from either the Host or remote system, and is executed by the remote system.

In this scenario, the master function should be compiled for both the Host and remote system, and the slave function should be compiled for the slave processor or into a library file, depending on the type of remote system executing the slave function.

The recommended way to compile the master and slave functions of a user-defined MIL function is to compile both master and slave functions for both the Host and remote systems. Compiling the function in this way ensures that your MIL application will be portable to computers that might or might not have access to remote systems. Function contexts allocated to run on the remote computer (**MfuncAlloc()** with **InitFlag** set to **M_REMOTE+...**) will, in the absence of a remote processor, be executed on the master computer and function contexts allocated using the default execution type (**MfuncAlloc()** with **InitFlag** set to **M_DEFAULT**) will make use of the remote system when one is available.

Compiling a master and/or slave function for the Host processor is typically the same as compiling any other function of your application. Compiling a master and/or slave function for a remote system with an on-board processor, however, will depend on the type of on-board processor.

For example, for a Matrox Odyssey system, if the master and slave functions are compiled into Matrox Odyssey's shell as well as compiled for the Host processor, then your function can be called from the Host and be executed by Matrox Odyssey's processor; your function can also be called from within a function being executed by Matrox Odyssey's processor.

For more information on developing functions for Matrox Odyssey, refer to Matrox Imaging Library Developer's Toolkit Programming Guide for Matrox Odyssey.

Executing the slave function on a remote system

Typically, MIL automatically determines whether to have the slave function executed by the Host processor or by the remote processor. To determine where to execute the slave function, MIL looks to the values of the user-defined MIL function parameters, registered in the master function. When all of the MIL objects passed as parameters to the function are allocated on the same system, or if one of the parameters is a MIL system identifier, the slave function is executed by the processor associated with that system. If the master function has multiple MIL object parameters, and the MIL objects passed to these parameters are allocated on different systems, an error message is returned.

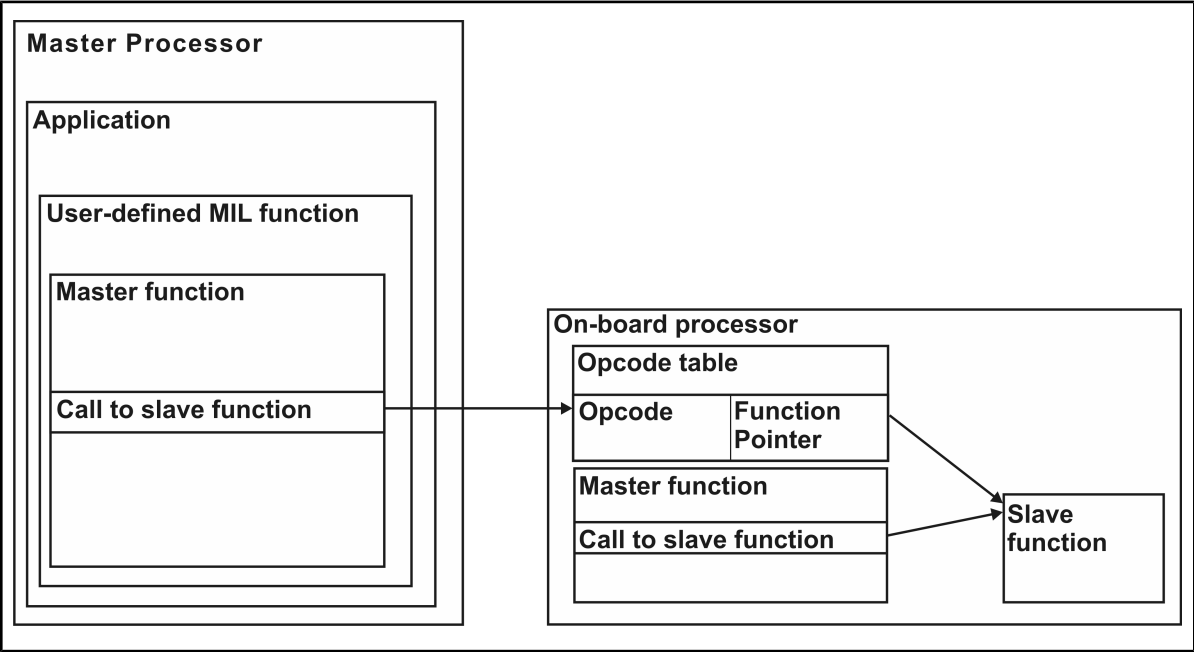
When a call to a user-defined MIL function satisfies the conditions for remote execution listed above, and the remote processor is an on-board processor, you will get a MIL error if the opcode was not correctly added to the opcode table. For more information about opcodes and the opcode table see the *Opcode table* subsection in the *Master slave dynamics on a remote system* section in *Chapter 27: The MIL function development module*.

When a user-defined MIL function has a user-defined MIL object as one of its parameters, and you want the slave function to be executed by a remote processor, you must allocate this object on the system with the remote processor. For more information, see the *Associating a MIL identifier with a user-defined object* section later in this chapter.

- ❖ Note that when the slave function is executed by the remote processor, operations that use other systems cannot be executed. For example, when a slave function is executed on Matrox Odyssey, and there is a call in the slave function to an **MdigGrab()** function that performs a grab on Matrox Morphis, this grab operation will cause an error.

Remote systems with on-board processors
Opcode table

When compiling a new user-defined MIL function for an on-board remote processor, you must add an entry to the on-board opcode table at the position corresponding to the opcode passed to **MfuncAlloc()** within the master function. This step is crucial since calls to slave functions from the Host processor are done via opcodes when they are executed by a remote processor; when the slave processor is given the opcode of a function to perform, it relies on the opcode table to determine which function to actually call.



❖ Note that the manner in which the opcode is added to the opcode table is board-specific. For more information on opcode associations and the opcode table for Matrox Odyssey, refer to Matrox Imaging Library Developer's Toolkit Programming Guide for Matrox Odyssey.

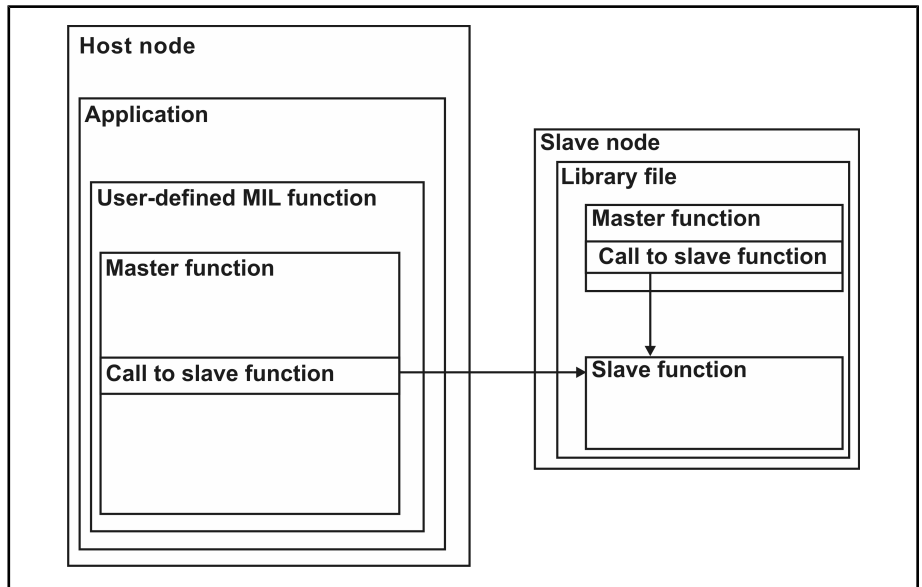
Remote systems in a Distributed MIL cluster

Library file

When compiling a new user-defined MIL function for a remote system in a Distributed MIL cluster, you must compile your user-defined function into a library file.

In the master function, located in your application running on the Host node, you must provide the name of the library file containing the slave function and the name of the slave function, using **MfuncAlloc()** with **SlaveFunctionDLLName** and **SlaveFunctionName**, respectively.

If your function will be called from a slave node, the library file should also contain the master function of your user-defined function. In the master function, located in the library file, you can provide a pointer to your slave function using **MfuncAlloc()** using **SlaveFunctionPtr**, or provide the name of the library file and the name of the slave function, as above. From within the library file, it is faster to provide a pointer to the slave function than it is to provide the name of the library file and the name of the function.



Associating a MIL identifier with a user-defined object

The MIL Function Development module allows you to associate a MIL identifier with an object so that this object can be treated as a standard MIL object. Use the **MfuncAllocId()** function to associate a MIL identifier with any array or custom data structure. Defining your own user-defined MIL objects allows you to create data objects whose exact internal structure has specific requirements and whose data members should not be accessed directly. Once you have associated the user-defined object with a MIL identifier, it is known as a user-defined MIL object, and it will be subject to error checking and tracing as any other MIL object.

Typically, you would create user-defined MIL objects in a user-defined MIL function. In this case, it is recommended that you create a user-defined MIL function whose sole purpose is to allocate the user-defined object and associate it with a MIL identifier. That is, the slave function of the user-defined MIL function only declares the object and associates it with a MIL identifier. This type of function is called a user-defined MIL allocation function. User-defined MIL allocation functions should be allocated using **MfuncAlloc()** with **M_ALLOC**.

It is recommended that you register a MIL system identifier as a parameter of your user-defined MIL allocation function (**MfuncParamId()** with **Params** set to **M_SYSTEM**); this allows you to easily control which system your function will be executed on at runtime, and, by extension, on which system your user-defined object will be allocated. The MIL identifier associated with your newly allocated user-defined object can be retrieved by having your allocation function return it, or by storing it in a pointer parameter of type **MIL_ID** (**MfuncParamIdPointer()**).

- ❖ Note that user-defined objects stored in the memory of a remote processor are not accessible from the Host processor, and vice versa, unless they are stored in shared memory.

To refer to user-defined MIL objects, use their MIL identifiers. To refer to the actual data grouped in this user-defined MIL object, use a pointer to the object. You can retrieve the address of the object using the **MfuncInquire()** function with **M_OBJECT_PTR**.

For example, for an object with this structure:

```
/* Define the user-defined object's structure */
struct MyStruct {
    int MyData1;
    int MyData2;
    int MyData3;
};
```

The following portion of MIL code shows how to allocate a custom object and associate it with a MIL identifier; this type of code would typically be found in a user-defined allocation function.

```
MIL_ID MyObjectId;
struct MyStruct MyObject = { 0 , 0 , 0 };

MyObjectId = MfuncAllocId( M_DEFAULT, M_USER_OBJECT_1+0x0001, &MyObject );
```

The following portion of MIL code shows how to retrieve the address of the user-defined MIL object, allocated above, and use it to store data using a pointer; this type of code would typically be found in a user-defined MIL processing function which accepts the MIL object as a parameter.

```
struct MyStruct *MyObjectPtr = M_NULL;
int Result1 = 0;
int Result2 = 0;
int Result3 = 0;

MfuncInquire( MyObjectId, M_OBJECT_PTR, &MyObjectPtr );

/* Perform the calculations */
/* Result1 = ... */
/* Result2 = ... */
/* Result3 = ... */

/* The user-defined object's data can now be accessed using the pointer
   retrieved by MfuncInquire */
MyObjectPtr->MyData1 = Result1 ;
MyObjectPtr->MyData2 = Result2 ;
MyObjectPtr->MyData3 = Result3 ;
```

When creating a user-defined MIL object, you must specify the MIL type of your object. MIL provides two object type groups (**M_USER_OBJECT_1** and **M_USER_OBJECT_2**), permitting you to distinguish between categories of custom created objects. Each user-defined object group can contain up to 16 user-defined object types (**M_USER_OBJECT_n + m**, where n is 1 or 2, and m specifies the offset within the group).

To inquire about the type of a user-defined MIL object, use the **MfuncInquire()** function with **M_OBJECT_TYPE_EXTENDED**.

Once you have finished using a user-defined MIL object, you must free the MIL identifier associated with this object, using the **MfuncFreeId()** function. Note that you need not free the MIL identifier in the same function where it was associated with the object (**MfuncAllocId()**). It is recommended that you create a user-defined MIL function whose sole purpose is to free the user-defined object and its MIL identifier. This type of function should be allocated using **MfuncAlloc()** with **M_FREE**, and must accept the MIL identifier to be freed as its only parameter.

Chapter

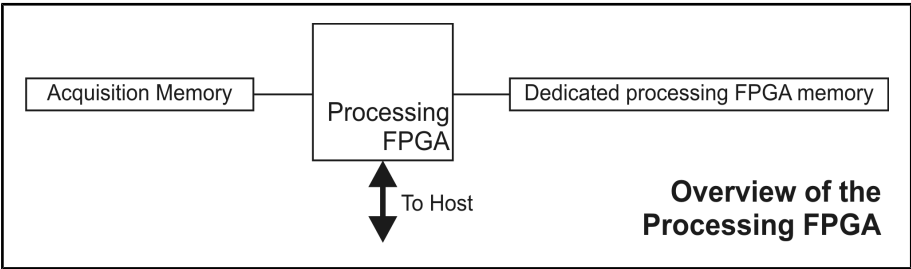
28

Using MIL with a Processing FPGA

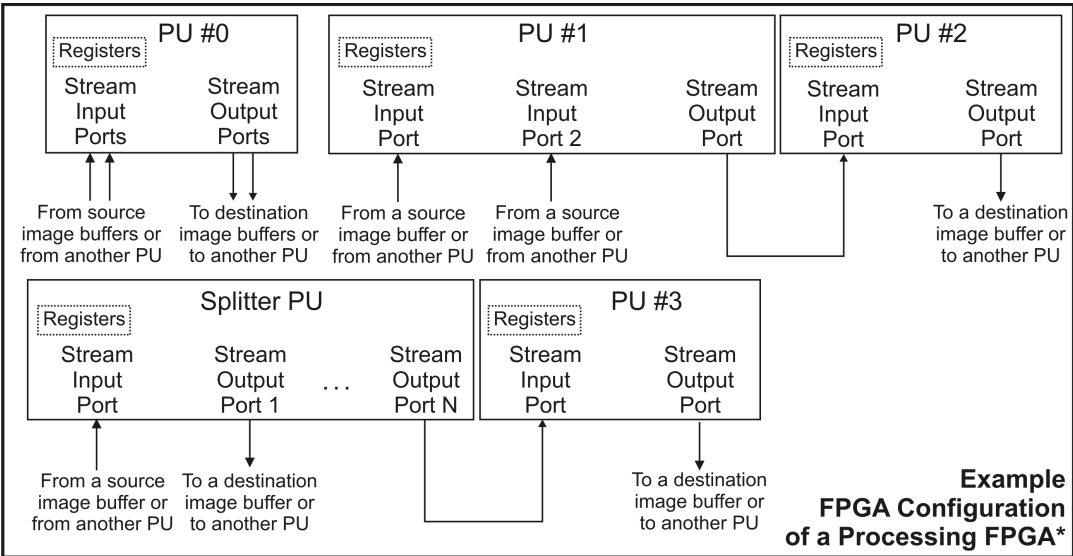
This chapter explains how to use the MIL FPGA module with a Matrox board that has a Processing FPGA loaded with an FPGA configuration.

Using MIL with a Processing FPGA - overview

Some Matrox imaging boards have a Processing FPGA. You can configure this Processing FPGA to perform some required processing operations on-board to free up the Host for other tasks.



Before any processing can take place using the Processing FPGA, you must configure it with an FPGA configuration that contains the appropriate processing units (PUs) to carry out the required set of tasks.



*All interconnections, transfer units and memory controllers are not illustrated. These are handled transparently in MIL.

Several useful Matrox FPGA configurations are distributed with MIL. However, depending on your application's requirements, you might need to develop a custom FPGA configuration using Matrox PUs and, if required, custom PUs. If this is the case, you must purchase and use the Matrox FPGA Developer's Toolkit (FDK).

Once the Processing FPGA is configured, MIL will use it to perform any MIL function that can be performed using the Matrox PUs in the loaded FPGA configuration, if possible. MIL functions cannot be performed using custom PUs.

To use PUs without a corresponding MIL function, or to reduce memory accesses by routing an output of one PU to an input of another PU (cascaded or parallel processing operations), you must use the MIL FPGA module to create, set up, and dispatch the required command(s) to the board. The MIL FPGA module allows you to create a command and set its target PU, specify the image buffers to associate with the stream input and output ports of the PU, and set the PU's user-specific registers (with required processing information). Besides associating image buffers with the ports, the MIL FPGA module also allows you to route the output of one command's PU to an input port of another command's PU, if the PUs are interconnected in the loaded FPGA configuration.

You can create a simple function (primitive function) that calls all the required functions of the MIL FPGA module. However, if incorrect parameters are passed to your primitive function, you are not notified and you cannot use MIL's tracing mechanism to determine the cause of the unexpected results. To enable basic error checking and parameter validation, you must call your primitive function from a user-defined MIL function; to do so, see

Chapter 27: The MIL function development module. Note that this error checking adds a small overhead to the processing function.

When your Matrox imaging board also has an on-board processor, you must use a user-defined MIL function because the MIL FPGA functions must be executed by the on-board processor.

- ❖ If your board has an on-board processor and you don't define your primitive function as a user-defined MIL function, an error will be generated.

Although an FPGA configuration also has components (referred to as transfer units and memory controllers) required to transfer data to/from the required PUs and to/from memory, the MIL FPGA module handles these transparently.

Processing an image with a Processing FPGA

To process images using a Processing FPGA, perform the following steps:

1. Use the MILConfig utility to select the FPGA configuration with which to program the Processing FPGA. When you select a Matrox FPGA configuration, MILConfig will list all the PUs that are in this particular configuration. For a description of PUs and their interconnections, refer to the Matrox FPGA Components Reference help file.
 - ❖ Note that a valid Matrox FPGA configuration file has a (.mbf) or (.firmware) extension and can be found in the *Matrox Imaging\Drivers\Board_Name\Firmware\Processing* folder (for example, Board_Name should be Solios in the case of a Matrox Solios board). For Matrox boards with an on-board processor, the Matrox FPGA configuration can be found under the *Matrox Imaging\Board_Name\Shell* folder (for example, Board_Name can be Odyssey).
2. Call the **MsysAlloc()** function to load the configuration in the Processing FPGA. To load a different FPGA configuration from within your application, you can use **MfpgaLoad()**. You must include the MIL FPGA header file *milfpga.h* to make a call to any Mfpga module function.
 - ❖ Note that for Matrox boards with on-board processors, you must reboot your computer to load the Matrox FPGA configuration, selected with the MILConfig utility, into the Processing FPGA. **MfpgaLoad()** is not supported on these boards.
3. Grab the image(s) into one or more on-board FPGA-accessible buffer(s) (**MbufAlloc...** with **M_GRAB + M_FPGA_ACCESSIBLE**).

4. Process the images using one of the following techniques:
 - Call the required MIL function equivalent to the PU(s) in the loaded FPGA configuration.
 - Set up and call the primitive function (or the user-defined MIL function) that dispatches the required commands to the Processing FPGA on board. To set up the primitive function (or the user-defined MIL function), see the *Steps to develop a function that performs an operation using a Processing FPGA* section later in this chapter.

Note that processing can typically be performed by the Processing FPGA if the source buffer is allocated on-board, the destination buffer is allocated in non-paged Host memory or on-board, and the FPGA configuration includes a path from the PU(s) to the memory banks in which the specified source and destination buffers are located. In the case of Matrox imaging boards with on-board processors, destination buffers cannot be in Host memory; only on-board destination buffers are supported. Using the **M_FPGA_ACCESSIBLE** attribute when allocating a buffer ensures that the buffer is in memory accessible to the Processing FPGA, although there might not be an available path between the PU and the memory bank, depending on the selected FPGA configuration.

- ❖ Note that if calling a standard MIL function that is not associated to the PU(s) in the loaded FPGA configuration, the Host will process the image(s), but the processing might be significantly slower than normal if the images are on-board.

If calling a primitive function and the target PU is not in the loaded FPGA configuration, an error is generated.

Steps to develop a function that performs an operation using a Processing FPGA

To perform a processing operation using your Processing FPGA when there is no equivalent MIL function, you must at least create a primitive function that uses the functions of the MIL Mfpga module to set up and dispatch all appropriate commands to the Processing FPGA. Also, you must include the MIL FPGA header file *milfpga.h* to make a call to any function of the Mfpga module.

To create the primitive function:

1. Inquire the handle of your source and destination buffers using **MfuncInquire()** with **M_BUFFER_INFO**. MIL FPGA functions do not take buffer identifiers; instead, they take the handle of the buffers.
2. To enable basic parameter checking when using Mfpga functions, use **MfpgaControl()** with **M_ERROR** and **M_PRINT_ENABLE**. Note that this will also report errors when attempting to link the target PU with a PU not in the FPGA configuration, and when attempting to use an invalid interrupt.
3. Allocate an FPGA command context for the target PU of the Processing FPGA using **MfpgaCommandAlloc()**. The FPGA command context is used to contain the necessary command information to perform the required operation using the specified PU.
4. Specify the source image buffer(s) using **MfpgaSetSource()**.
5. Specify the destination image buffer(s) using **MfpgaSetDestination()**.
6. Set other processing information in the PU's user-specific registers using **MfpgaSetRegister()**.
7. To route the stream output of one PU to the stream input port of another PU (that is, to cascade the PUs) repeat steps 2 through 5, creating and setting up a FPGA command context for each PU that you need to cascade. Then, link the command contexts, using **MfpgaSetLink()**.
8. Queue the command(s) defined by the command context(s), using **MfpgaCommandQueue()**.

9. Retrieve non-image results from the PU's registers using **MfpgaGetRegister()**.
10. Free the command context using **MfpgaCommandFree()**.

The following code snippet is an example of the type of code that should be included within your primitive function to set up and dispatch a command to a Processing FPGA.

```

/* Inquire the buffer info object for the source and destination buffers. */
MfuncInquire(MilSourceImage, M_BUFFER_INFO, &Src);
MfuncInquire(MilDestinationImage, M_BUFFER_INFO, &Dest);

/* Allocate an asynchronous command context on the AddConstant processing unit.
*/
if(MfpgaCommandAlloc(MfuncBufOwnerSystemId(Src), M_DEVO, FPGA_ADDCONST_FID,
                    M_DEFAULT, M_DEVO, M_ASYNCHRONOUS, M_DEFAULT,
                    &AddConstantContext))
{
    /* Initialize the shadow register structure. */
    InitializeShadowRegisters(&ShadowRegisters, Src, ConstantValue, ControlFlag);

    /* Input0 of AddConstant processing unit is connected to the Src buffer. */
    MfpgaSetSource(AddConstantContext, Src, M_INPUT0, M_DEFAULT);

    /* Output0 of AddConstant processing unit is connected to the Dest buffer. */
    MfpgaSetDestination(AddConstantContext, Dest, M_OUTPUT0, M_DEFAULT);

    /* Pass the initialized shadow register structure, it will be programmed */
    /* when the processing operation is dispatched (before processing starts). */
    MfpgaSetRegister(AddConstantContext, M_USER, 0, sizeof(ShadowRegisters),
                    (void*)&ShadowRegisters), M_WHEN_DISPATCHED);

    /* Queue the processing operation, because this context is asynchronous, */
    /* MfpgaComamndQueue will return before the operation is completed. */
    MfpgaCommandQueue(AddConstantContext, M_DEFAULT, M_DEFAULT);

    /* Free the allocated context. */
    MfpgaCommandFree(AddConstantContext, M_DEFAULT);
}

```

- ❖ Note that to run the primitive function on a board with an on-board processor, you must call your primitive function from the slave function of a user-defined MIL function. The slave and primitive functions must be precompiled and preloaded into the shell of the on-board processor. For more information on creating a user-defined MIL function, see *Chapter 27: The MIL function development module*. Note that even without an

on-board processor, it is useful to call your primitive function from a user-defined MIL function. This not only makes your code more portable, but also takes advantage of the parameter checking and error reporting capabilities that are integrated in MIL.

Whenever you want to associate an FPGA command context with a PU, you must make sure the PU is actually present in the FPGA configuration loaded in the FPGA. In addition, when implementing cascaded or parallel processing operations, it is necessary to make sure that the PUs are interconnected properly in the FPGA configuration. Matrox FPGA configurations are documented in the Matrox FPGA Configurations help file.

Primitive function and execution of operation by PU

The FPGA configuration that you load into your Processing FPGA contains one or more PUs that your primitive function can use to carry out specific image processing operations. With each PU that you want to use, you must associate an FPGA command context, using **MfpgaCommandAlloc()**. An FPGA command context is a container for the information required to select and configure a PU, within a loaded FPGA configuration, to perform an image processing operation. The command context stores information regarding the target PU, the source and destination buffers, operation information, register settings, register read requests, link information, and the execution mode. In essence, the command context specifies the command required to perform a processing operation. Once your command is completed and the FPGA command context is no longer needed, it should be freed using **MfpgaCommandFree()**.

When allocating the command context, you must specify the target PU. To do so, you must specify the PU's function identifier. If your Matrox imaging board has multiple Processing FPGAs on board, you must also specify in which is the required PU located. Several different variations of a Matrox PU can exist, each with slightly different optimizations, functionalities, or restrictions. If the loaded FPGA configuration has multiple versions of the same PU, identify which to use by specifying the required PU's subfunction identifier. Lastly, if the FPGA configuration loaded in the selected Processing FPGA has multiple instances of the required PU, you must specify which to use by specifying its rank. If any of the above-mentioned circumstances do not apply, you can set the corresponding parameter to **M_DEFAULT**.

There are certain PUs that require multiple passes to setup and use. For example, the PU that performs a LUT mapping needs two passes. The first pass sets up the lookup table and the second enables the PU to map the input image through the LUT. You must ensure that you set up this type of PU correctly.

Refer to the Matrox FPGA Configurations help file for a list of the PUs contained in the available Matrox FPGA configurations. In addition, refer to the Matrox FPGA Components Reference help file for a description of the Matrox PUs.

Source and destination image buffers

Before using a PU, you must specify the appropriate source and destination image buffers for the command. Some PUs do not produce image data and, therefore, they do not require destination image buffers. To specify the source and destination buffers, use **MfpgaSetSource()** and **MfpgaSetDestination()**, respectively. These functions require that you pass the handle of the buffer and not its identifier. To get the handle of the buffer, you must call **MfuncInquire()** with **M_BUFFER_INFO**.

All PUs have documented limitations on the types of buffers from which they can receive data using their stream input port(s). They also have documented limitations on the buffers to which they can transmit data from their stream output port(s). You must keep these limitations in mind when allocating your buffer(s). **MfpgaSetDestination()** and **MfpgaSetSource()** will not automatically convert the buffers so that they are appropriate for the operation. For more information, refer to the PU documentation available in the Matrox FPGA Components Reference help file.

- ❖ Note that processing can typically be performed by the Processing FPGA if the source buffer is allocated on-board, the destination buffer is allocated in non-paged Host memory or on-board, and the FPGA configuration includes a path from the PU(s) to the memory banks in which the specified source and destination buffers are located. In the case of Matrox imaging boards with on-board processors, destination buffers cannot be in Host memory; only on-board destination buffers are supported.

When allocating your image buffers for a Processing FPGA operation (**MbufAlloc...**), you can have MIL automatically select the optimal memory location, which is accessible to the Processing FPGA, for the image buffers. To do so, add **M_FPGA_ACCESSIBLE** to the other required attributes (for example, **M_PROC+M_GRAB**). Note that although the buffers will be allocated in memory accessible to the Processing FPGA, you must ensure that a path between the PU and the buffers is available.

To explicitly allocate your buffer in a specific memory location that is accessible to the Processing FPGA, you can add one of the following attributes to **M_FPGA_ACCESSIBLE**. Note that by explicitly allocating your buffer in a specific memory location, you can make your code less portable to other boards with a different number of memory banks or Host memory restrictions. However, you might obtain some performance benefits.

- **M_ON_BOARD**. Explicitly allocates the buffer in on-board memory.
 - **M_HOST_MEMORY**. Explicitly allocates the buffer in Host memory.
 - **M_MEMORY_BANK_n**. Explicitly specifies the on-board memory bank in which to allocate your image buffer. The variable *n* specifies the rank of the on-board memory bank in which your image must be allocated.
- ❖ Note that **M_MEMORY_BANK_n** is not supported on Matrox imaging boards with on-board processors; you must use other buffer attributes to force the allocation of the buffer into a specific memory bank (for example, **M_FPGA_ACCESSIBLE+M_FAST_MEMORY**).

When allocating your image buffers, special care should be taken regarding the buffer width because certain Matrox boards have limitations on the width of allocated buffers. For instance on Matrox Solios, the width of buffers allocated in acquisition memory must be a multiple of 64 bytes and the width of destination buffers on the Host must be a multiple of 8 bytes.

- ❖ Note that in general, child buffers are not supported; only color-band child buffers are supported.

If you want to route the stream output of one PU to the stream input port of another PU, you will have to cascade the PUs using the appropriate functions. For more information on these functions, see the *Cascaded and parallel processing* section later in this chapter.

Setting and retrieving results from PU registers

Each PU has a dedicated register space which consists of three separate sections: a header section, an I/O control section, and a user-specific section. Each of these sections contains registers that hold specific information about the PU. Registers in the header and I/O sections do not require user modification, while registers in the user-specific section allow you to set the PU's operation settings. Most PUs that return non-image results also return these results in registers in the user-specific section.

You can set a PU's user-specific register section using up to four calls to **MfpgaSetRegister()**. Similarly, you can set up a request to retrieve the values of a PU's user-specific register section using up to four calls to **MfpgaGetRegister()**. Four calls is usually more than sufficient because you can write to or read from multiple registers at a single time.

For each PU, you are provided with PU-specific, predefined, C constructs to facilitate setting the PU's user-specific registers. The constructs are supplied in header files (`fpga_*.h`), located in the `\Matrox Imaging\MIL\Examples\BoardNameFDK\Include` directory (for example, BoardNameFDK can be SoliosFDK). For Matrox Odyssey, the C constructs that facilitate setting the PU's user-specific registers are in the `\Matrox Imaging\odyssey\src\headers\local` directory. For custom PUs, Matrox PU Designer,

the utility of the Matrox FDK that allows you to create your own PU, will generate the required register header files for you. You should use these constructs to write values to/read values from registers, instead of hardcoding a register or field offset, since these helps keep your code forward-compatible.

In the header file, there is a structure that represents all the user-specific registers of the PU. For example, this is the structure for all the user-specific registers of the gainoffset PU.

```

/*****
* Section name   : USER
* USEROFF       : 0x60
*****/
typedef struct
{
    FPGA_GAINOFFSET_REG_CTRL    ctrl;    /* Address offset : [0x60] */
    FPGA_GAINOFFSET_REG_CLIPVAL clipval; /* Address offset : [0x68] */
} FPGA_GAINOFFSET_USER_ST;

```

In the header file, there is also a union for each user-specific register of the PU. Each union contains five items. The first four items u64, u32, u16, and u8 represent different sizes of data blocks that you can write to/read from the register. The last item is a C structure that facilitates setting and getting the different fields of the register. This union provides you with different ways of writing to/reading from the register. For example, you could clear all the fields of a register to 0 using the U64 item of the union. Alternatively, you could write specific values to the different register fields. The following is the union for the control (ctrl) register of the gainoffset PU.

```

/*****
* Register name : ctrl
* Address       : 0x60
*****/
typedef union
{
    M_UINT64 u64;
    M_UINT32 u32;
    M_UINT16 u16;
    M_UINT8  u8;

    struct
    {

```

```

M_UINT64 dsttype          : 2; /* Bits<1:0>,      */
/* The data type and depth of the output image. */
M_UINT64 gaintype         : 2; /* Bits<3:2>,      */
/* The data type and depth of the gain image.   */
M_UINT64 offtype          : 2; /* Bits<5:4>,      */
/* The data type and depth of the offset image. */
M_UINT64 srctype          : 2; /* Bits<7:6>,      */
/* The data depth and type of the input image on
/* which to operate.                             */
M_UINT64 shift            : 5; /* Bits<12:8>,     */
/* Number of bits to right-shift.                */
M_UINT64 rsvd0            : 19; /* Bits<31:13>,    */
/* Reserved Field                                */
} f;

} FPGA_GAINOFFSET_REG_CTRL;

```

You should always set up and verify the values of the individual fields of each register before setting these values in the target PU. The following is an example that verifies the values for the registers of the gainoffset PU. If the wrong value is used, an error is returned.

```

/* Sets the gain offset registers */

/* Specifies the stream input port 0 data type. */
switch(Src1Type)
{
case 8+M_UNSIGNED:
    pGOShadow->ctrl.f.srctype = 0x1; break;
case 16+M_UNSIGNED:
    pGOShadow->ctrl.f.srctype = 0x3; break;
default:
    MosPrintf(MIL_TEXT("Gain offset error: Invalid input stream port 0 data type.\n"));
    break;
}

/* Specifies the stream input port 1 data type. */
switch(Src2Type)
{
case 8+M_UNSIGNED:
    pGOShadow->ctrl.f.offtype = 0x1; break;
case 16+M_UNSIGNED:
    pGOShadow->ctrl.f.offtype = 0x3; break;
default:
    MosPrintf(MIL_TEXT("Gain offset error: Invalid input stream port 1 data type.\n"));
    break;
}

```

```

/* Specifies the stream input port 2 data type. */
switch(Src3Type)
{
    case 8+M_UNSIGNED:
        pGOShadow->ctrl.f.gaintype = 0x1;    break;
    case 16+M_UNSIGNED:
        pGOShadow->ctrl.f.gaintype = 0x3;    break;
    default:
        MosPrintf(MIL_TEXT("Gain offset error: Invalid input stream port 3 data type.\n"));
        break;
}

/* Specifies the stream output port 0 data type. */
switch(DstType)
{
    case 8+M_UNSIGNED:
        pGOShadow->ctrl.f.dsttype = 0x1;    break;
    case 16+M_UNSIGNED:
        pGOShadow->ctrl.f.dsttype = 0x3;    break;
    default:
        MosPrintf(MIL_TEXT("Gain offset error: Invalid output stream port 0 data type.\n"));
        break;
}

/* Specifies the clipping value. */
pGOShadow->clipval.f.clipval = 1MaxValue;

/* Computes the number of bits to right-shift the output image. */
j=1;
for(i=0; i<32; i++)
{
    if(Src4 & j)
    {
        pGOShadow->ctrl.f.shift = i;
        break;
    }
    j <<= 1;
}

```

Once the values of the fields of all registers are set up, use **MfpgaSetRegister()** to specify the contents for your PU's registers. Note that when you queue your command with **MfpgaCommandQueue()**, the values set using **MfpgaSetRegister()** will be written to the hardware as a block once the hardware becomes free. For this reason, it is essential that you always set the values of all the fields of the target register before queuing the command.

```

/* Inquire the buffer info object for the source and destination buffers. */
MfuncInquire(MilSourceImage, M_BUFFER_INFO, &Src);
MfuncInquire(MilDestinationImage, M_BUFFER_INFO, &Dest);

/* Allocate an asynchronous command context on the AddConstant processing unit.
*/
if(MfpgaCommandAlloc(MfuncBufOwnerSystemId(Src), M_DEVO, FPGA_ADDCONST_FID,
                    M_DEFAULT, M_DEVO, M_ASYNCHRONOUS, M_DEFAULT,
                    &AddConstantContext))
{
    /* Initialize the shadow register structure. */
    InitializeShadowRegisters(&ShadowRegisters, Src, ConstantValue, ControlFlag);

    /* Input0 of AddConstant processing unit is connected to the Src buffer. */
    MfpgaSetSource(AddConstantContext, Src, M_INPUT0, M_DEFAULT);

    /* Output0 of AddConstant processing unit is connected to the Dest buffer. */
    MfpgaSetDestination(AddConstantContext, Dest, M_OUTPUT0, M_DEFAULT);

    /* Pass the initialized shadow register structure, it will be programmed */
    /* when the processing operation is dispatched (before processing starts). */
    MfpgaSetRegister(AddConstantContext, M_USER, 0, sizeof(ShadowRegisters),
                    (void*)&ShadowRegisters), M_WHEN_DISPATCHED);

    /* Queue the processing operation, because this context is asynchronous, */
    /* MfpgaComamndQueue will return before the operation is completed. */
    MfpgaCommandQueue(AddConstantContext, M_DEFAULT, M_DEFAULT);

    /* Free the allocated context. */
    MfpgaCommandFree(AddConstantContext, M_DEFAULT);
}

```

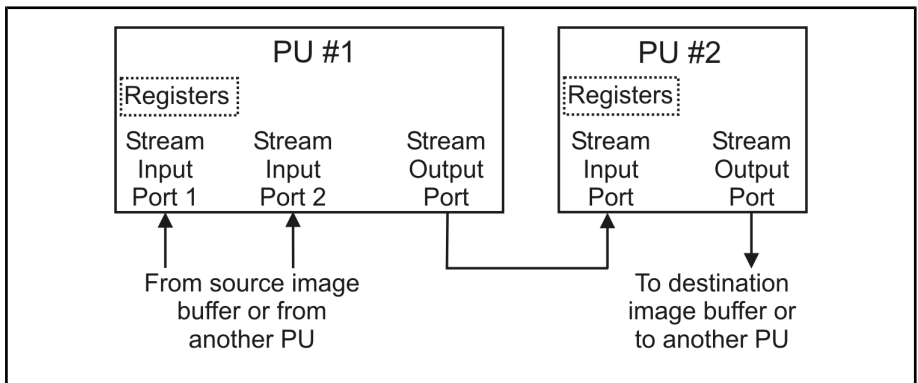
Cascaded and parallel processing

When designing a primitive function for your application, you might encounter situations where the entire operation must be carried out by two or more PUs. For example, your application might require a Bayer filter as well as a LUT mapping.

Depending on the application, you might want specific PU operations to be cascaded (the result of one PU transferred directly to another PU) or carried out in parallel, or a combination of both to reduce the number of memory accesses. If the PUs are appropriately interconnected in the FPGA configuration, MIL allows you to do so. To determine if the required PUs are appropriately interconnected in a Matrox FPGA configuration, see the Matrox FPGA Configurations help file. You can also load an FPGA configuration with Matrox FPGA Assistant; it will show the interconnections.

Cascaded processing operation

You can cascade the processing operations of the PUs associated with two command contexts using **MfpgaSetLink()**.



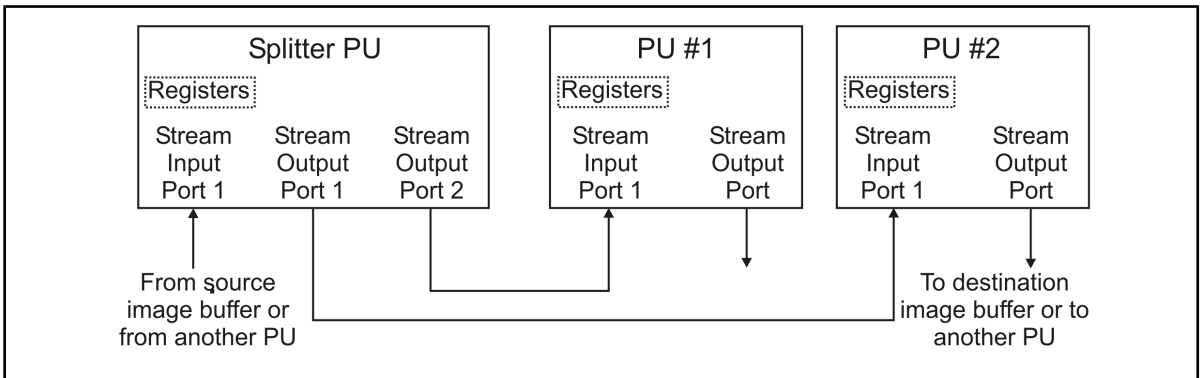
When you link two command contexts, the stream output of one PU is used as the stream input for the other PU. To transfer image data between two PUs, the stream output of the source PU and the stream input of the destination PU must be configured to the same image data format. For more information, refer to the PU documentation available in the Matrox FPGA Components Reference help file.

To cascade several PUs, you must create a command context for each PU and then link the command contexts two at a time. Note that calling `MfpgaSetLink()` to route the output of one PU to the stream input of another PU requires that this path is available between the PUs in the FPGA configuration.

To ensure that the cascaded processing operations start at the right moment in time, there are special considerations when queuing their commands. For more information, see the *Issuing commands to the Processing FPGA and retrieving results* section later in this chapter.

Parallel processing operation

You can also send data simultaneously from memory or from a PU to two or more different PUs. To do so, the FPGA configuration must include a specialized PU called a splitter PU. The splitter PU receives image data at its stream input port and copies this data to all its stream output ports. If you cascade the stream output ports of the splitter PU with the stream input ports of the required PUs, the required PUs will simultaneously process the same data. For example, you can simultaneously send the result of a PU that performs a Bayer operation to a PU that performs a gain and offset operation and a PU that performs a min/max operation.



- ❖ Note that a path between the stream output ports of the splitter PU and the stream input ports of the required PUs must exist. The number of output ports of the splitter PU, included in your FPGA configuration, depends on the PU's subfunction identifier. In addition, the stream outputs of the splitter PU and the stream input(s) of the destination PU(s) must be configured to the same image data format.

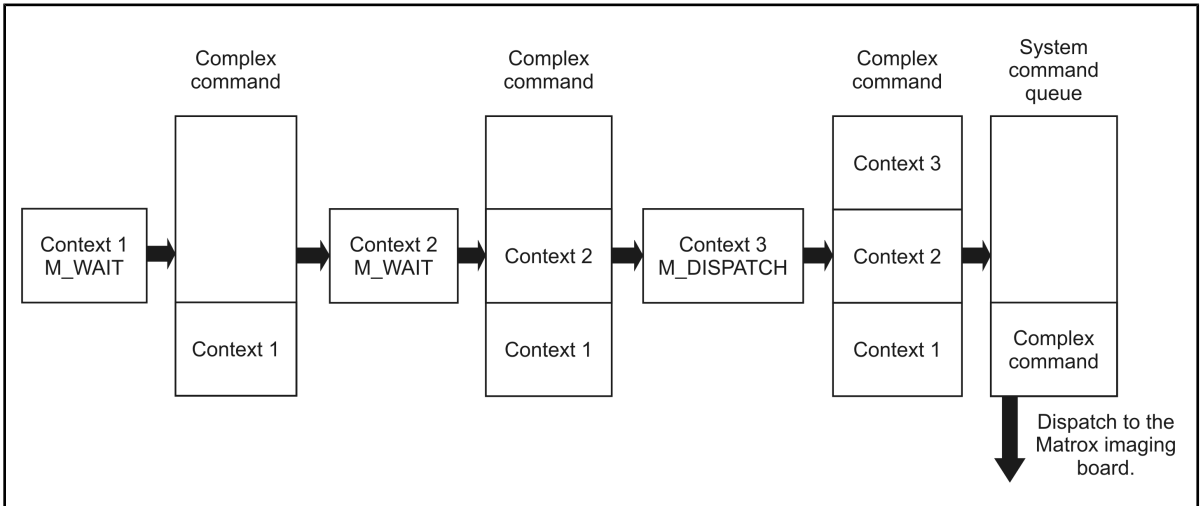
Similar to when cascading processing operations, to ensure that the processing operations are performed in parallel, there are special considerations when queuing their commands. For more information, see the *Issuing commands to the Processing FPGA and retrieving results* section later in this chapter.

- ❖ Note that you must set up the splitter PU just like any other PU. This means that you must create a command context for each required splitter PU present in the FPGA configuration.

Issuing commands to the Processing FPGA and retrieving results

Once you have specified register settings, source and destination image buffers, register result retrieval requests, and optionally linked information for each command context, you must queue the command(s), defined by these contexts, on the thread's system command queue using **MfpgaCommandQueue()**. Once on the queue, the commands will be executed in first-in first-out order as the Processing FPGA resources become available. Note that two commands can typically run at the same time if they don't reference the same buffers and they use different FPGA components that can access these buffers using different paths.

For linked command contexts, you will have to combine the commands, defined by these contexts, into a complex command once you are ready to queue them. You combine commands into a complex command by queuing all linked commands, except the last, using **MfpgaCommandQueue()** with **M_WAIT**. Then you should add the last command to the complex command using **MfpgaCommandQueue()** with **M_DISPATCH**; this closes the complex command and queues it on the system command queue.



The following example shows how to link two command contexts and then combine these commands into a complex command.

```

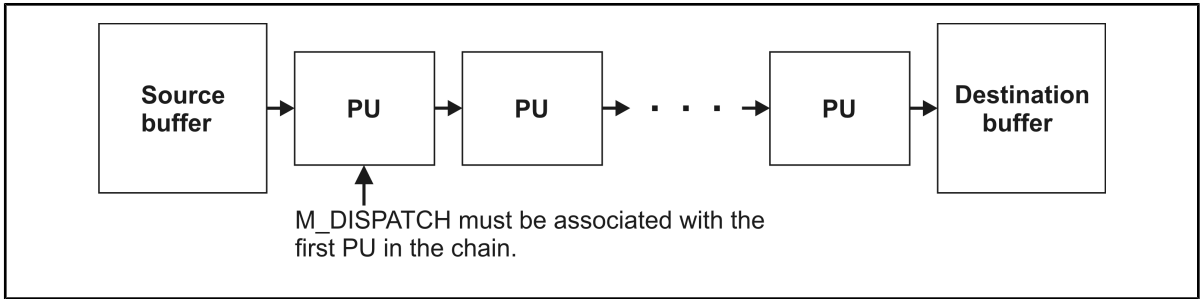
/* Output0 of the OffsetGain is connected to Input0 of the LutMap.          */
MfpgaSetLink(OffsetGainContext, M_OUTPUT0, LutMapContext, M_INPUT0, M_DEFAULT);

/* Queue the processing operation, we use the M_WAIT flag because          */
/* other MfpgaCommandQueue() follows.                                     */
/* This tells the software to wait for other MfpgaCommandQueue() operations. */
MfpgaCommandQueue(LutMapContext, M_DEFAULT, M_WAIT);

/* Queue the processing operation, using the M_DEFAULT flag.              */
/* This tells the software to send the processing operation to the FPGA.    */
MfpgaCommandQueue(OffsetGainContext, M_DEFAULT, M_DEFAULT);

```

For Matrox imaging boards with on-board processors, the order in which you add commands to the complex command plays an important role; the command queued using **MfpgaCommandQueue()** with **M_DISPATCH** must be the command associated with the first PU in the processing chain.



The completion mode of an **MfpgaCommandQueue()** call specifies when the command should be tagged as complete. When tagged as complete, you can read back results from your destination buffer(s) or the variables passed to the register result retrieval requests. For complex commands, the completion mode of the last command added to the complex command determines the completion mode for the entire complex command. All other commands in the complex command should have their completion mode set to **M_DEFAULT**.

When you allocate an FPGA command context, you must specify how the command will be dispatched to the Matrox imaging board. You can specify that a command should run synchronously or asynchronously. A synchronous command (**MfpgaCommandAlloc()** with **M_SYNCHRONOUS**) causes the thread from which the command is dispatched to wait for the command to complete before executing subsequent statements. An asynchronous command (**MfpgaCommandAlloc()** with **M_ASYNCHRONOUS**) causes the thread to continue executing subsequent statements without waiting for the command to complete. For complex commands, MIL uses the synchronous/asynchronous setting of the last command in the complex command (that is, the call to **MfpgaCommandQueue()** with **M_DISPATCH**).

The Host processor supports both **M_SYNCHRONOUS** and **M_ASYNCHRONOUS** settings; it can continue to work if the command is asynchronous. However on Matrox imaging boards with on-board processors, only the **M_SYNCHRONOUS** setting is supported: the thread of the on-board processor executing the call to **MfpgaCommandAlloc()** with **M_DISPATCH** always waits until each command has finished processing (synchronous operation).

Developing a user-defined MIL function to run the primitive function

Although creating a primitive function that calls the appropriate functions of the MIL FPGA module is a critical step to issuing commands to your processing FPGA, sometimes additional steps are required. If your Matrox imaging board has an on-board processor, you must call your primitive function from the slave function of a user-defined MIL function; for more information on how to define a user-defined MIL function, see *Chapter 27: The MIL function development module*.

If your Matrox imaging board does not have an on-board processor (for example, Matrox Solios), you can also call your primitive function from the slave function of a user-defined function to obtain basic error reporting or other built-in MIL features. Doing so might incur some small overhead but provides great benefits and can save considerable development time.

To develop a user-defined MIL function to call the primitive function, perform the following:

1. Create a master function to register and check the parameters passed to the master function before sending the information to the slave function.
 2. Create a slave function to recuperate the values of the parameters registered in the master function and call the primitive function. To log any of your user-defined error messages with the MIL error handling mechanism, use **MfuncErrorReport()**.
- ❖ Note that when your Matrox imaging board has an on-board processor, the slave and primitive functions must be precompiled and preloaded into the shell of the on-board processor.

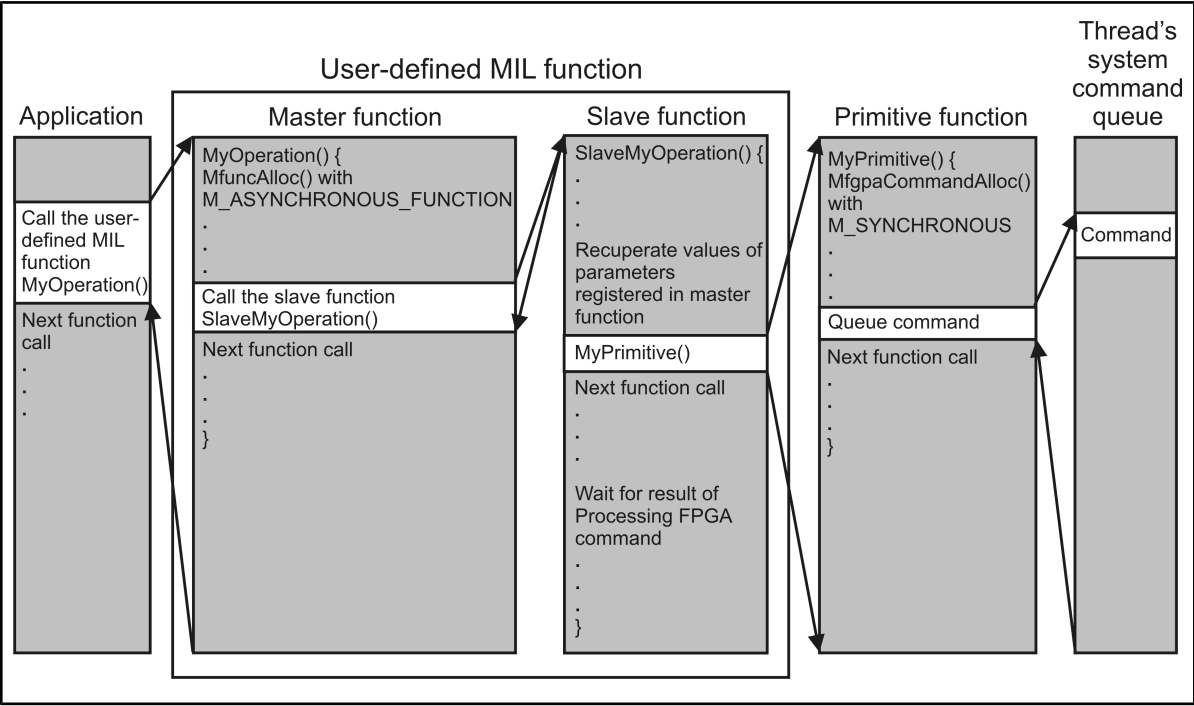
Master function and slave function and execution of operation specified by command context

When you allocate a MIL function context in the master function of your user-defined MIL function using **MfuncAlloc()**, you must specify the identifier to associate with the MIL function. This identifier need not match the PU's function identifier that is specified when allocating the command context using **MfpgaCommandAlloc()**.

Operation synchronization

When you allocate your MIL function context using **MfuncAlloc()**, you must specify whether your master function should wait for the slave function to finish executing before executing the next statement in the master function. This should not be confused with the synchronization specified by **MfpgaCommandAlloc()**, which specifies whether the current thread should issue the Processing FPGA command synchronously or asynchronously.

The following illustrates the primitive function call from the slave function of a user-defined function.



Choosing whether to set the user-defined function as synchronous or asynchronous depends on the application. A synchronous function can be easier to code, but using an asynchronous function allows you to perform Host and Processing FPGA operations simultaneously. Note that for Matrox imaging boards with on-board processors, the thread of the on-board processor always waits until each Processing FPGA command has completed its synchronous operation.

Chapter

29

Using MIL under Linux

This chapter presents the features of the Linux operating system.

Working with Linux

Besides the computer requirements discussed in the *Requirements to run MIL* section in *Chapter 1: Introduction*, note the following when running MIL under Linux:

Licensing

Linux requires the use of a USB port hardware license-key.

Limited and unsupported features

Note the following limitations due to the Linux operating system:

- Buffers cannot be allocated in video memory.
- Linux offers limited control over window attributes, such as window resizing and overlapping.
- Microsoft DirectX is unsupported.
- The device-independent bitmap (DIB) storage format is unsupported.
- No tearing is unsupported in Windowed mode.
- Intellicam is unsupported.
- Processing graphical user interfaces (GUIs) are unsupported.
- Interactive dialog boxes are unsupported.

Board restrictions

The following Matrox boards have restrictions when the Linux operating system is being used:

- **Helios.** Serial ports on the Helios board are not supported.
- **Morphis.** Older Matrox Morphis boards can cause the system to crash at startup. The Linux kernel has difficulty recognizing the Matrox Morphis serial ports. See the *Matrox Morphis Readme* to upgrade the firmware on the Matrox Morphis board to fix this issue.