

Matrox Imaging Library

version 7

Command Reference

Manual no. 10512-701-0710

March 1, 2002

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

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

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

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

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

Texas Instruments™ is a trademark of Texas Instruments Incorporated.

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

© Copyright Matrox Electronic Systems Ltd., 2002. All rights reserved.

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

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

Disclaimer: Matrox Electronic Systems Ltd. reserves the right to make changes in specifications at any time and without notice. The information provided by this document is believed to be accurate and reliable. However, neither Matrox Electronic Systems Ltd. nor its suppliers assume any responsibility for its use; or for any infringements of patents or other rights of third parties resulting from its use. No license is granted under any patents or patent right of Matrox Electronic Systems Ltd.

PRINTED IN CANADA

Contents

Chapter 1 : Programming with MIL. 7

A MIL overview	8
Starting your MIL application	9
Header file and libraries	10
MIL object manipulation concepts	10
Error handling and reporting	11
Tracing an application	12
A quick command reference	13
The application allocation and control module	13
The blob analysis module	14
The buffer allocation and access module. . .	15
The calibration module	17
The code module	19
The digitizer allocation and control module.	19
The display allocation and control module.	20
The basic data generation module.	21
The basic graphics module	21
The basic image processing module	22
The measurement module	25
The geometric model finder module	26
The optical character recognition module . .	27
The basic pattern recognition module. . . .	28
The system allocation and inquiry module .	30

Chapter 2: The command reference descriptions . . . 31

The reference description notes 32

Appendix A: The default setup configuration file . . . 563

The default setup configuration file 564

When you do not want to use defaults 567

Appendix B: The MIL Function Developer's Toolkit 569

The MIL Function Developer's Toolkit 570

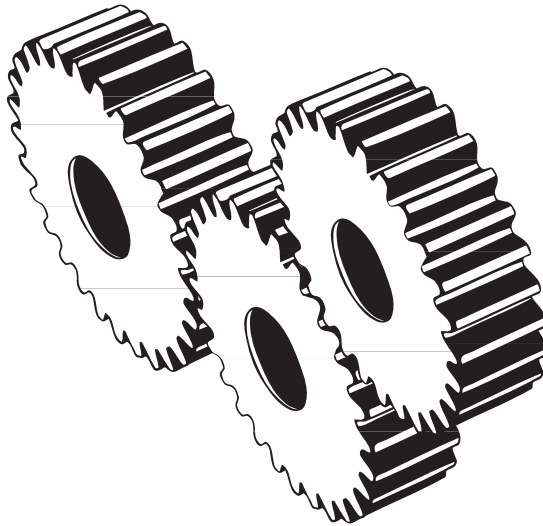
An example using the Function Developer's Toolkit 570

MIL Function Developer's Toolkit Command Reference 573

Index

Product Support

The MIL Command Reference



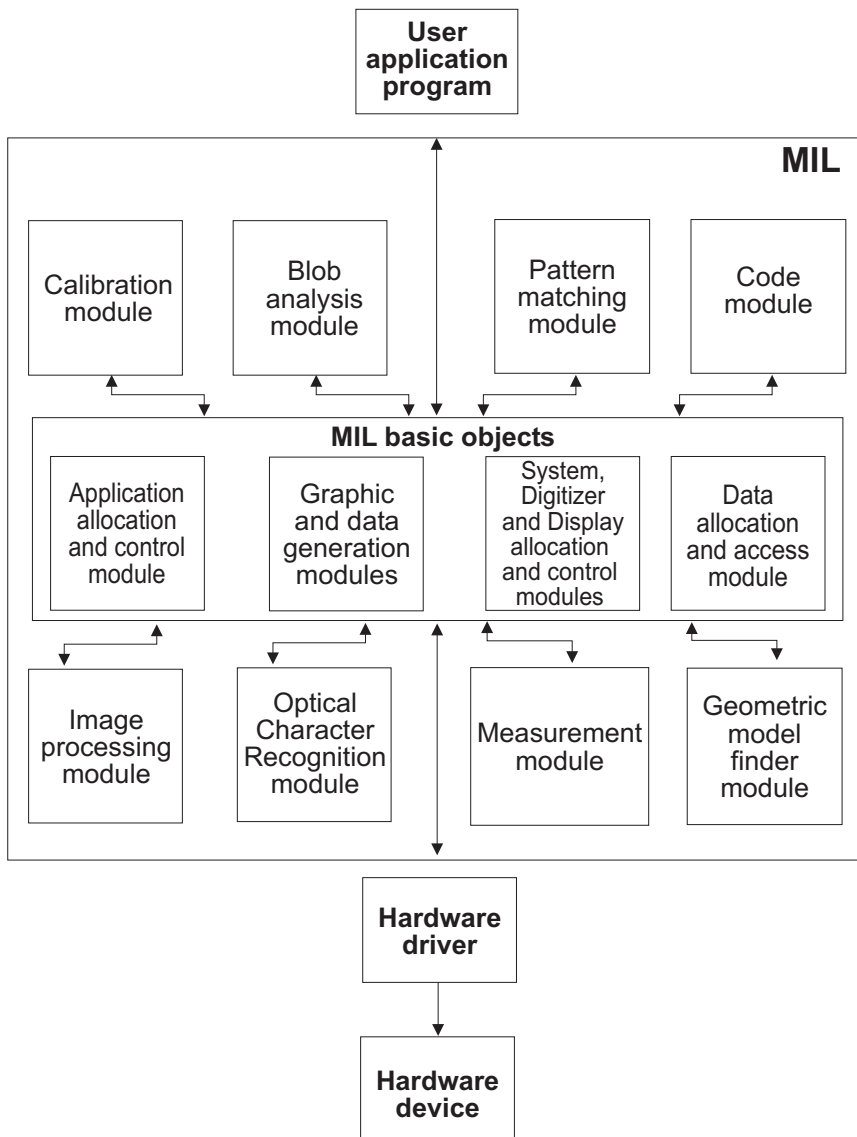
Putting thoughts into motion....

Chapter 1 : Programming with MIL

A MIL overview

The MIL user model

The Matrox imaging library (MIL) is a hardware-independent library divided into different modules based on functionality.



Usage

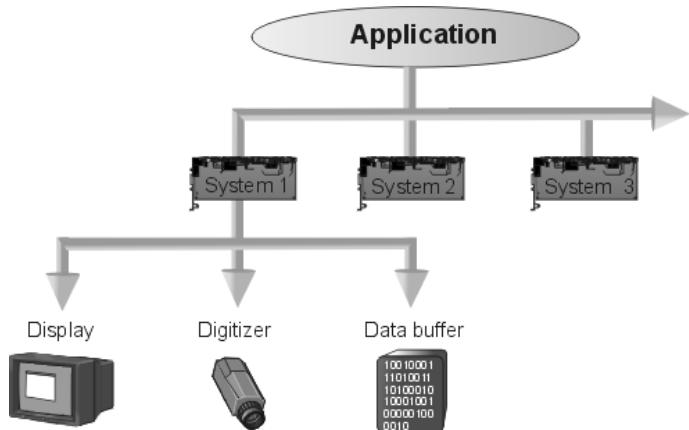
Application and system initialization

The *MIL User Guide* describes how to solve typical applications using the library. Code examples are also provided.

Starting your MIL application

At the beginning of each MIL application, you need to:

1. Allocate your application with **MappAlloc()**. This will create a control and execution environment for your application. Once you have finished using an application, you should free it with **MappFree()**.
2. Allocate your hardware system with **MsysAlloc()**. This will open communication channels and initialize the hardware resources, which includes any available graphics controller. Once Host communication has been established with a system, you can allocate its memory resources, display, and input capabilities.



For typical setups, you will only need to use one system, whereas for more sophisticated setups, you might need to allocate more than one. Use their system identifiers to select between them.

Once you have completely finished using a system, you should free the device, using **MsysFree()**.

Default initialization

If the required system is mapped to the default location specified in the *milsetup.h* file, you can perform the above steps by making a single call to **MappAllocDefault()**. Review the *milsetup.h* file to make sure that the default setup configuration matches your system configuration (refer to *Appendix A: The default setup configuration file* for more information on this file). The **MappAllocDefault()** macro can also allocate a default display, digitizer, and image buffer. Use the **MappFreeDefault()** macro to free the defaults allocated.

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

Header file and libraries

The required header file

To compile a MIL application program, you must include the *mil.h* header file, in addition to the required standard C include files. This *mil.h* file includes all constant definitions, type definitions, and function prototypes. It also includes any required macro files (for example, the *milsetup.h* file for the **MappAllocDefault()** macro).

Linking to the MIL library

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\MIL\LIBRARY\WINNT\MSC\DLL* or user-specified directory.

MIL object manipulation concepts

Data objects

MIL manipulates different types of objects. Objects must be allocated by MIL before they can be used. Besides allocating your MIL application and system (discussed in the previous sub-section), the following objects must also be allocated:

- Displays
- Digitizers
- Buffers

Displays and digitizers

With MIL, display and digitizer objects provide a way to communicate or control dedicated hardware resources. Several of these devices can be allocated at the same time; you use their identifiers to select between them. Note that since MIL finds the best device to use for display, display number parameters should always be set to `M_DEFAULT`. Once you have finished using a device, you should free it, using **MdigFree()** or **MdispFree()**. In the *MIL User Guide*, refer to *Chapter 21: Grabbing with your digitizer* for more information on digitizers and *Chapter 19: Displaying an image* for more information on graphics display controllers.

Buffers

Buffers are simply storage locations for data. The most generally used buffers, referred to as data buffers, are allocated with **MbufAllocColor()**, **MbufAlloc1d()** or **MbufAlloc2d()**; whereas, other data buffer, such as pattern matching model buffers, are allocated with commands that are specific to that MIL module and are only used by that module.

You can manipulate portions of data buffers by allocating sub-buffers or child buffers. Any manipulation performed on the child buffer directly affects the parent buffer and vice versa. Any operation that can be performed on the parent buffer can also be performed on the child buffer. In the *MIL User Guide*, refer to *Chapter 17: Specifying and managing your data buffers* for more information on allocating buffers.

Error handling and reporting

Error reporting

When calling a function, it is a good idea to provide detection and handling of errors, especially when allocating buffers and devices. Otherwise, your program might produce unexpected results. Note, every allocation returns an identifier; `M_NULL` is returned if the allocation was unsuccessful.

MIL has an error-reporting mechanism that is adaptable to your application development stage. By using **MappGetError()**, you can detect errors by having them reported to the Host screen, and by checking the system error code. You can also enable or disable error reporting to the screen with **MappControl()**; by default, errors are reported to the

screen. During application development, having errors reported to the screen is recommended so that you can quickly debug the application.

You can determine the success of a command, using **MappGetError()**, then handle the outcome accordingly. Using **MappHookFunction()**, you can attach (or detach) a user-defined function to MIL errors when they occur. Using **MappGetError()**, you can also get any associated error messages.

The error description

MappGetError() can provide the name of the function that caused an error, a system-error message associated to the error, and more specific sub-messages. Note, it returns the same messages as those printed to the screen when error reporting is enabled.

Tracing an application

Debugging an application

When developing an application, it is often useful to trace the command calls made by the application to debug it.

MIL supports an automatic tracing mechanism that can be enabled or disabled with **MappControl()**. When the MIL tracing mechanism is enabled, as each command is processed, its function name and parameters are reported to the screen. By default, the tracing mechanism is disabled.

You can attach or detach a user-defined function to the start or end of all subsequent MIL function calls, using **MappHookFunction()**.

A quick command reference

This section lists and provides a quick reference description of the commands of each MIL module. It also discusses each module, giving a brief overview of the capabilities of the library. For a complete description of the syntax and use of each command refer to the *Command reference descriptions* chapter.

The application allocation and control module

The application allocation and control module supports the MIL allocation and environment control functions. These include MIL initialization, error reporting, and application tracing functions.

MIL allocation and control commands	Command parameters	Description
MappAlloc()	InitFlag, ApplicationIdPtr	Allocate a MIL application.
MappAllocDefault()	InitFlag, ApplicationIdPtr, SystemIdPtr, DisplayIdPtr, DigIdPtr, ImageBufIdPtr	Allocate MIL application defaults.
MappControl()	ControlType, ControlValue	Control an application environment setting.
MappControlThread()	ControlId, ControlType, ControlValue, ControlVarPtr	Allocate/control MIL application thread(s) or events.
MappFree()	ApplicationId	Free a MIL application.
MappFreeDefault()	ApplicationId, SystemId, DisplayId, DigId, ImageBufId	Free MIL application defaults.
MappGetError()	ErrorType, ErrorPtr	Get error codes and related information.
MappGetHookInfo()	EventId, InfoType, UserVarPtr	Get information about a hooked event.
MappHookFunction()	HookType, HookHandlerPtr, UserDataPtr	Hook a function to an event.
MappInquire()	InquireType, UserVarPtr	Inquire about the application parameter setting.
MappModify()	FirstMILId, SecondMILId, ModificationType, ModificationFlag	Modify specified MIL object(s).
MappTimer()	ControlValue, TimePtr	Control the MIL timer.

The blob analysis module

The blob analysis module is a set of functions that can measure a wide assortment of blob (or object) features, such as the blob area, perimeter, Feret diameter at a given angle, minimum bounding box, and compactness.

Blob analysis commands	Command parameters	Description
MblobAllocFeatureList()	SystemId, FeatureListIdPtr	Allocate a blob analysis feature list.
MblobAllocResult()	SystemId, BlobResIdPtr	Allocate a blob analysis result buffer.
MblobCalculate()	BlobIdentImageId, GrayImageId, FeatureListId, BlobResId	Perform blob analysis calculations.
MblobControl()	BlobResId, ControlType, ControlValue	Control a blob analysis processing mode setting.
MblobDraw()	GraphContId, ResultId, DestImageId, Operation, Label, ControlFlag	Draw features of specified blob results in an image buffer.
MblobFill()	BlobResId, DestImageBufId, Criterion, Value	Fill blobs that meet a given criteria.
MblobFree()	BlobId	Free the blob analysis result buffer or the feature list.
MblobGetLabel()	BlobResId, XPos, YPos, LabelVarPtr	Get the label value of a blob at a specific position.
MblobGetNumber()	BlobResId, CountVarPtr	Get the number of currently included blobs.
MblobGetResult()	BlobResId, Feature, TargetArrayPtr	Read feature values of the included blobs.
MblobGetResultSingle()	BlobResId, LabelVal, Feature, TargetVarPtr	Read the feature value of a single blob.
MblobGetRuns()	BlobResId, LabelVal, ArrayType, RunXPtr, RunYPtr, RunLengthPtr	Get the blob run-length encoding information.
MblobInquire()	BlobResId, InquireType, UserVarPtr	Inquire about a blob analysis processing mode.
MblobLabel()	BlobResId, DestImageBufId, Mode	Draw a labeled image.
MblobReconstruct()	SrcImageBufId, SeedImageBufId, DestImageBufId, Operation, ProcMode	Reconstruct blobs (or blob holes) in an image buffer.
MblobSelect()	BlobResId, Operation, Feature, Condition, CondLow, CondHigh	Select blobs for calculations and result retrieval.

Blob analysis commands	Command parameters	Description
MblobSelectFeature()	FeatureListId, Feature	Select feature(s) to be calculated.
MblobSelectFeret()	FeatureListId, Angle	Add Feret angle to the feature list.
MblobSelectMoment()	FeatureListId, MomType, XMomOrder, YMomOrder	Add specified moment to the feature list.

The buffer allocation and access module

The data buffer allocation and access module is a group of functions that supports all the MIL data buffer manipulations. These tools include those that can allocate, read from, and write to general data buffers.

Data allocation and access commands	Command parameters	Description
MbufAlloc1d()	SystemId, SizeX, Type, Attribute, BufIdPtr	Allocate a 1D data buffer.
MbufAlloc2d()	SystemId, SizeX, SizeY, Type, Attribute, BufIdPtr	Allocate a 2D data buffer.
MbufAllocColor()	SystemId, SizeBand, SizeX, SizeY, Type, Attribute, BufIdPtr	Allocate a color data buffer.
MbufBayer()	SrcImageBufId, DestImageBufId, WhiteBalanceCoefficientsID, ControlFlag	Decode the color information of a single-band, Bayer color-encoded image.
MbufChildColor()	ParentBufId, Band, BufIdPtr	Allocate a child data buffer within a color parent buffer.
MbufChildColor2d()	ParentBufId, Band, OffX, OffY, SizeX, SizeY, BufIdPtr	Allocate a child data buffer within a color parent buffer.
MbufChild1d()	ParentBufId, OffX, SizeX, BufIdPtr	Allocate a 1D child data buffer.
MbufChild2d()	ParentBufId, OffX, OffY, SizeX, SizeY, BufIdPtr	Allocate a 2D child data buffer.
MbufClear()	DestImageBufId, Color	Clears a buffer to a specified color.
MbufControl()	BufId, ControlType, ControlValue	Control specified buffer features.
MbufControlNeighborhood()	BufId, OperationFlag, OperationValue	Change the value of an operation flag associated with a custom kernel or structuring element.
MbufCopy()	SrcBufId, DestBufId	Copy data from one buffer to another.

Data allocation and access commands	Command parameters	Description
MbufCopyClip()	SrcBufId, DestBufId, DestOffX, DestOffY	Copy buffer clipping data outside destination buffer.
MbufCopyColor()	SrcBufId, DestBufId, Band	Copy one or all bands of an image buffer.
MbufCopyColor2d()	SrcBufId, DestBufId, SrcBand, SrcOffX, SrcOffY, DestBand, DestOffX, DestOffY, SizeX, SizeY	Copy a 2D region of one or all bands of an image buffer to another buffer.
MbufCopyCond()	SrcBufId, DestBufId, CondBufId, Condition, CondValue	Copy conditionally the source buffer to the destination buffer.
MbufCopyMask()	SrcBufId, DestBufId, MaskValue	Copy buffer with mask.
MbufCreateColor()	SystemId, SizeBand, SizeX, SizeY, Attribute, ControlFlag, Pitch, ArrayOfDataPtr, BufIdPtr	Create a color data buffer.
MbufCreate2d()	SystemId, SizeX, SizeY, Type, Attribute, ControlFlag, Pitch, DataPtr, BufIdPtr	Create a two-dimensional data buffer.
MbufDiskInquire()	FileName, InquireType, UserVarPtr	Inquire about the buffer data in a file.
MbufExport()	FileName, FileFormat, SrcBufId	Export a data buffer to a file.
MbufExportSequence()	FileName, FileFormatId, BufArrayPtr, NumberOfImages, FrameRate, ControlFlag	Export a sequence of image buffers to an AVI file.
MbufFree()	BufId	Free a data buffer.
MbufGet()	SrcBufId, UserArrayPtr	Get data from a buffer and place it in a user-supplied array.
MbufGetColor()	SrcBufId, DataFormat, Band, UserArrayPtr	Get data from one or all bands of a buffer and place it in a user-supplied array.
MbufGetColor2d()	SrcBufId, DataFormat, Band, OffX, OffY, SizeX, SizeY, UserArrayPtr	Get data from a region of one of all bands of a buffer and place it in a user-supplied array.
MbufGetLine()	ImageBufId, StartX, StartY, EndX, EndY, Mode, NumPixelsPtr, UserArrayPtr	Read a series of pixels within specified coordinates, count them, and store them in a user-defined array.
MbufGet1d()	SrcBufId, OffX, SizeX, UserArrayPtr	Get data from a 1D area of a buffer and place it in a user-supplied array.
MbufGet2d()	SrcBufId, OffX, OffY, SizeX, SizeY, UserArrayPtr	Get data from a 2D area of a buffer and place it in a user-supplied array.

Data allocation and access commands	Command parameters	Description
MbufImport()	FileName, FileFormat, Operation, SystemId, BufIdPtr	Import data from a file into a data buffer.
MbufImportSequence()	FileName, FileFormatId, Operation, SystemId, BufArrayPtr, StartImage, NumberOfImages, ControlFlag	Import a sequence of images from an *.avi file into separate image buffers.
MbufInquire()	BufId, InquireType, UserVarPtr	Inquire about a data buffer parameter setting.
MbufLoad()	FileName, BufId	Load MIL file format data from a file into a data buffer.
MbufPut()	DestBufId, UserArrayPtr	Put data from a user-supplied array into a data buffer.
MbufPutColor()	DestBufId, DataFormat, Band, UserArrayPtr	Put data from a user-supplied array into one or all bands of a data buffer.
MbufPutColor2d()	DestBufId, DataFormat, Band, OffX, OffY, SizeX, SizeY, UserArrayPtr	Put data from a user-supplied array into a region of one of all bands of a data buffer.
MbufPutLine()	ImageBufId, StartX, StartY, EndX, EndY, Mode, NbPixelsPtr, UserArrayPtr	Write a specified series of pixels within specified coordinates on a line.
MbufPut1d()	DestBufId, OffX, SizeX, UserArrayPtr	Put data from a user-supplied array into a 1D area of a buffer.
MbufPut2d()	DestBufId, OffX, OffY, SizeX, SizeY, UserArrayPtr	Put data from a user-supplied array into a 2D area of a buffer.
MbufRestore()	FileName, SystemId, BufIdPtr	Restore Mil file format data from a file into an automatically allocated data buffer.
MbufSave()	FileName, BufId	Save a data buffer in a file using the MIL output file format.

The calibration module

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

Calibration commands	Command parameters	Description
McalAlloc()	Mode, ModeFlag, CalibrationIdPtr	Allocate a calibration object.

Calibration commands	Command parameters	Description
McalAssociate()	CalibrationId, ImageOrDigitizerId, ControlFlag	Associate/disassociate a calibration object to/from an image or digitizer.
McalControl()	CalibrationId, ControlType, ControlValue	Control a calibration object parameter setting.
McalFree()	CalibrationId	Free a calibration object.
McalGrid()	CalibrationId, SrcImageBufId, GridOffsetX, GridOffsetY, GridOffsetZ, RowNumber, ColumnNumber, RowSpacing, ColumnSpacing, Mode, ModeFlag	Calibrate your imaging setup using a grid.
McalInquire()	CalibrationOrMilId, InquireType, UserVarPtr	Inquire about a calibration object setting or about the calibration object associated to an image or digitizer.
McalList()	CalibrationId, XPixArray, YPixArray, XWorldArray, YWorldArray, ZWorld, NumPoint, Mode, ModeFlag	Calibrate your imaging setup using a list of coordinates.
McalRelativeOrigin()	CalibrationId, XOffset, YOffset, ZOffset, AngularOffset, ControlFlag	Change the origin and/or orientation of a relative coordinate system.
McalRestore()	FileName, ControlFlag, CalibrationIdPtr	Restore a calibration object from a file.
McalSave()	FileName, CalibrationId, ControlFlag	Save a calibration object to a file.
McalTransformCoordinate()	CalibrationOrMilId, TransformType, X, Y, ResXPtr, ResYPtr	Convert coordinates between their world and pixel values.
McalTransformCoordinateList()	CalibrationOrMilId, TransformType, NumPoints, SrcXPtr, SrcYPtr, ResXPtr, ResYPtr	Convert a list of coordinates between their world and pixel values.
McalTransformImage()	SrcImageBufId, DestImageBufId, CalibrationId, InterpolationMode, OperationType, ControlFlag	Physically transform an image to remove any distortions.
McalTransformResult()	CalibrationOrMilId, TransformType, ResultType, Result, ResResult	Convert a result between its world and pixel value.

The code module

The code module allows you to read and write DataMatrix codes as well as several types of bar codes.

Code commands	Command parameters	Description
McodeAlloc()	SystemId, CodeType, ControlFlag, CodeIdPtr	Allocate a code object.
McodeControl()	CodeId, ControlType, ControlValue	Control a code object.
McodeFree()	CodeId	Free a code object.
McodeGetResult()	CodeId, ResultType, ResultPtr	Get a result from a read or write operation.
McodeInquire()	CodeId, InquireType, UserVarPtr	Inquire about a code object parameter setting.
McodeRead()	CodeId, ImageBufId, ControlFlag	Read a specific type of code in an image.
McodeRestore()	FileName, SystemId, ControlFlag, CodeIdPtr	Restore a code object previously saved to a file.
McodeSave()	FileName, CodeId, ControlFlag	Save the specified code object in a file.
McodeWrite()	CodeId, ImageBufId, String, ControlFlag	Encode an ASCII string.

The digitizer allocation and control module

The digitizer allocation and control module supports the allocation, manipulation, and control of digitizers.

Digitizer allocation and control commands	Command parameters	Description
MdigAlloc()	SystemId, DigNum, DataFormat, InitFlag, DigIdPtr	Allocate a digitizer.
MdigChannel()	DigId, Channel	Select the active input channel of a digitizer.
MdigControl()	DigId, ControlType, ControlValue	Control the specified digitizer.
MdigFocus()	DigId, DestImageBufId, FocusImageRegionBufId, FocusHookPtr, UserDataPtr, MinPosition, StartPosition, MaxPosition, MaxPositionVariation, ProcMode, ResultPtr	Adjust a camera's lens motor to a position which provides optimum focus.

Digitizer allocation and control commands	Command parameters	Description
MdigFree()	DigId	Free a digitizer.
MdigGrab()	DigId, DestImageBufId	Grab data from an input device into a buffer.
MdigGrabContinuous()	DigId, DestImageBufId	Grab data continuously from an input device.
MdigGrabWait()	DigId, Flag	Wait for the end of the grab in progress.
MdigHalt()	DigId	Halt a continuous grab from an input device.
MdigHookFunction()	DigId, HookType, HookHandlerPtr, UserDataPtr	Hook a function to a digitizer event.
MdigInquire()	DigId, InquireType, UserVarPtr	Inquire about a digitizer parameter setting.
MdigLut()	DigId, LutBufId	Copy a LUT buffer to a digitizer LUT.
MdigReference()	DigId, ReferenceType, ReferenceLevel	Select digitization reference level.

The display allocation and control module

The display allocation and control module supports the allocation, manipulation, and control of displays.

Display allocation and control commands	Command parameters	Description
MdispAlloc()	SystemId, DispNum, DispFormat, InitFlag, DisplayIdPtr	Allocate a display.
MdispControl()	DisplayId, ControlType, ControlValue	Control the behavior of a MIL display window.
MdispDeselect()	DisplayId, ImageBufId	Stop displaying an image buffer.
MdispFree()	DisplayId	Free a display.
MdispHookFunction()	DisplayId, HookType, HookHandlerPtr, UserDataPtr	Hook a function to a display event.
MdispInquire()	DisplayId, InquireType, UserVarPtr	Inquire about a display parameter setting.
MdispLut()	DisplayId, LutBufId	Copy a LUT buffer to a display output LUT.
MdispOverlayKey()	DisplayId, KeyMode, KeyCond, KeyMask, KeyColor	Enable overlay keying.

Display allocation and control commands	Command parameters	Description
MdispPan()	DisplayId, XOffset, YOffset	Pan and scroll a display.
MdispSelect()	DisplayId, ImageBufId	Select an image buffer to display.
MdispSelectWindow()	DisplayId, ImageBufId, ClientWindowHandle	Select an image buffer to display in a user-defined window.
MdispZoom()	DisplayId, XFactor, YFactor	Zoom a display.

The basic data generation module

The basic data generation module provides a limited set of data generation tools that can be used to automatically generate predefined data in a data buffer (for example, generating ramp in a LUT buffer).

Basic data generation commands	Command parameters	Description
MgenLutFunction()	LutBufId, Func, a, b, c, StartIndex, StartXValue, EndIndex	Generate data into a LUT buffer using a specified standard mathematical function.
MgenLutRamp()	LutId, StartIndex, StartValue, EndIndex, EndValue	Generate ramp data into a LUT buffer.
MgenWarpParameter()	InWarpParameter, OutXLutOrCoef, OutYLut, OperationMode, Transform, Val1, Val2	Generate coefficients or LUTs for use with MimWarp() .

The basic graphics module

The basic graphics module provides a limited set of graphic primitives that can be used to create drawings and text annotations in an image.

Basic graphics commands	Command parameters	Description
MgraAlloc()	SystemId, GraphContIdPtr	Allocate a graphics context.
MgraArc()	GraphContId, DestImageBufId, XCenter, YCenter, XRad, YRad, StartAngle, EndAngle	Draw an arc.
MgraArcFill()	GraphContId, DestImageBufId, XCenter, YCenter, XRad, YRad, StartAngle, EndAngle	Draw a filled elliptic arc.

Basic graphics commands	Command parameters	Description
MgraBackColor()	GraphContId, BackgroundColor	Sets the background color of a graphics context.
MgraClear()	GraphContId, DestImageBufId	Clear an image buffer.
MgraColor()	GraphContId, ForegroundColor	Sets the foreground color of a graphics context.
MgraControl()	GraphContId, ControlType, ControlValue	Control the specified graphic context.
MgraDot()	GraphContId, DestImageBufId, XPos, YPos	Draw a dot.
MgraFill()	GraphContId, DestImageBufId, XStart, YStart	Perform a boundary-type seed fill.
MgraFont()	GraphContId, FontName	Associate a text font with a graphics context.
MgraFontScale()	GraphContId, XFontScale, YFontScale	Set the font scale of a graphics context.
MgraFree()	GraphContId	Free a graphics context.
MgraInquire()	GraphContId, InquireType, UserVarPtr	Inquire about the graphic parameters.
MgraLine()	GraphContId, DestImageBufId, XStart, YStart, XEnd, YEnd	Draw a line.
MgraRect()	GraphContId, DestImageBufId, XStart, YStart, XEnd, YEnd	Draw a rectangle.
MgraRectFill()	GraphContId, DestImageBufId, XStart, YStart, XEnd, YEnd	Draw a filled rectangle.
MgraText()	GraphContId, DestImageBufId, XStart, YStart, String	Write text.

The basic image processing module

The basic image processing module gives the user the possibility to do point-to-point, statistical, spatial filtering, morphological, and geometric transformation operations.

Basic image processing commands	Command parameters	Description
MimAllocResult()	SystemId, NbEntries, ResultType, ImResultIdPtr	Allocate an image processing result buffer.
MimArith()	Src1ImageBufId, Src2ImageBufId, DestImageBufId, Operation	Perform a point-to-point arithmetic operation.

Basic image processing commands	Command parameters	Description
MimArithMultiple()	Src1ImageBufId, Src2ImageBufId, Src3ImageBufId, Src4ImageBufId, Src5ImageBufId, DestImageBufId, Operation, OperationFlag	Perform a point-to-point arithmetic operation using multiple source images.
MimBinarize()	SrcImageBufId, DestImageBufId, Condition, CondLow, CondHigh	Perform a point-to-point binary-thresholding operation.
MimClip()	SrcImageBufId, DestImageBufId, Condition, CondLow, CondHigh, WriteLow, WriteHigh	Perform a point-to-point clipping operation.
MimClose()	SrcImageBufId, DestImageBufId, NbIteration, ProcMode	Perform a binary or grayscale closing-type morphological operation.
MimConnectMap()	SrcImageBufId, DestImageBufId, LutBufId	Perform a 3 by 3 binary connectivity mapping.
MimConvert()	SrcImageBufId, DestImageBufId, ConversionType	Perform a color conversion.
MimConvolve()	SrcImageBufId, DestImageBufId, KernelBufId	Perform a general convolution operation.
MimCountDifference()	Src1ImageBufId, Src2ImageBufId, ImResultId	Count image differences.
MimDilate()	SrcImageBufId, DestImageBufId, NbIteration, ProcMode	Perform a binary or grayscale dilation-type morphological operation.
MimDistance()	SrcImageBufId, DestImageBufId, DistanceTransform	Perform a distance transformation.
MimEdgeDetect()	SrcImageBufId, DestIntensityImageBufId, DestAngleImageBufId, KernelId, ControlFlag, Threshold	Perform a specific edge detection operation and produce a gradient intensity and/or gradient angle image.
MimErode()	SrcImageBufId, DestImageBufId, NbIteration, ProcMode	Perform an erosion-type morphological operation.
MimFindExtreme()	SrcImageBufId, ExtremeImResultId, ExtremeType	Find an image buffer's extremes (min, max).
MimFlip()	SrcImageId, DestImageId, Operation, OpFlag	Perform a horizontal or vertical image-flipping rotation
MimFree()	ImResultId	Free an image processing result buffer.
MimGetResult()	ImResultId, ResultType, UserArrayPtr	Get values from an image processing result buffer.

Basic image processing commands	Command parameters	Description
MimGetResult1d()	ImResultId, OffEntry, NbEntries, ResultType, UserArrayPtr	Get values from a 1D region of an image processing result buffer.
MimHistogram()	SrcImageBufId, HistImResultId	Generate the intensity histogram of an image buffer.
MimHistogramEqualize()	SrcImageBufId, DestImageBufId, Method, Alpha, Min, Max	Perform a histogram equalization of an image.
MimInquire()	BufId, InquireType, UserVarPtr	Inquire about an image processing result buffer parameter setting.
MimLabel()	SrcImageBufId, DestImageBufId, ProcMode	Label objects in an image buffer.
MimLocateEvent()	SrcImageBufId, EventImResultId, Condition, CondLow, CondHigh	Find pixel coordinates or values that satisfies a specified condition.
MimLutMap()	SrcImageBufId, DestImageBufId, LutBufId	Perform a point-to-point LUT-mapping operation.
MimMorphic()	SrcImageBufId, DestImageBufId, StructElemBufId, Operation, NbIteration, ProcMode	Perform a morphological transformation using a user-defined kernel.
MimOpen()	SrcImageBufId, DestImageBufId, NbIteration, ProcMode	Perform a binary or grayscale opening-type morphological operation.
MimPolarTransform()	SrcImageBufId, DestImageBufId, CenterPosX, CenterPosY, StartRadius, EndRadius, StartAngle, EndAngle, OperationMode, InterpolationMode, DestSizeXPtr, DestSizeYPtr	Perform a polar-to-rectangular or rectangular-to-polar transform.
MimProject()	SrcImageBufId, ProjImResultId, ProjAngle	Project a 2D image into 1D.
MimRank()	SrcImageBufId, DestImageBufId, StructElemBufId, Rank, ProcMode	Perform a rank filter on the pixels in an image.
MimResize()	SrcImageBufId, DestImageBufId, ScaleFactorX, ScaleFactorY, InterpolationMode	Resize an image.
MimRotate()	SrcImageBufId, DestImageBufId, Angle, SrcCenX, SrcCenY, DstCenX, DstCenY, InterpolationMode	Rotate an image.
MimShift()	SrcImageBufId, DestImageBufId, BitsToShift	Perform a point-to-point bit shift.

Basic image processing commands	Command parameters	Description
MimThick()	SrcImageBufId, DestImageBufId, NbIteration, ProcMode	Perform a binary or grayscale thickening operation on an image.
MimThin()	SrcImageBufId, DestImageBufId, NbIteration, ProcMode	Perform a binary or grayscale thinning operation on an image.
MimTransform()	SrcImageRBufId, SrcImageIBufId, DestImageRBufId, DestImageIBufId, TransformType, ControlFlag	Perform a Fast Fourier transform (FFT) or a Discrete Cosine transform (DCT).
MimTranslate()	SrcImageBufId, DestImageBufId, XDisplacement, YDisplacement, InterpolationMode	Translate an image in X and/or Y displacement.
MimWarp()	SrcImageId, DestImageId, WarpParam1Id, WarpParam2Id, OperationMode, InterpolationMode	Perform a warping.
MimWatershed()	SrcImageId, MarkerImageId, DestImageId, MinimumVariation, ControlFlag	Perform a watershed transformation.
MimZoneOfInfluence()	SrcImageBufId, DestImageBufId, OperationFlag	Perform a zone of influence detection.

The measurement module

The basic measurement module is a set of functions that can be used to take measurements using spatial reference positions in images.

Measurement commands	Command parameters	Description
MmeasAllocContext()	SystemId, ControlFlag, ContextIdPtr	Allocate a measurement context.
MmeasAllocMarker()	SystemId, MarkerType, ControlFlag, MarkerIdPtr	Allocate a measurement marker.
MmeasAllocResult()	SystemId, ResultType, MeasResultIdPtr	Allocate a measurement result buffer.
MmeasCalculate()	ContextId, Marker1Id, Marker2Id, MeasResultId, MeasurementList	Calculate measurements using two markers.
MmeasControl()	ContextId, ControlType, ControlValue	Control a measurement parameter setting.

Measurement commands	Command parameters	Description
MmeasDraw()	GraphContId, MarkerOrResultId, DestImageId, Operation, Index, ControlFlag	Draw features of specific markers or result occurrences in an image buffer.
MmeasFindMarker()	ContextId, ImageBufId, MarkerId, MeasurementList	Find a marker in an image and take its measurements.
MmeasFree()	MeasId	Free a measurement buffer (marker, result, or context).
MmeasGetResult()	MarkerOrMeasResultId, ResultType, FirstResultArrayPtr, SecondResultArrayPtr	Get the results of measurements taken.
MmeasGetResultSingle()	MarkerOrMeasResultId, ResultType, FirstResultArrayPtr, SecondResultArrayPtr, ResultIndex	Get a single result from a multiple marker's result array.
MmeasInquire()	MeasId, InquireType, FirstUserVarPtr, SecondUserVarPtr	Inquire about a marker, result, or context buffer.
MmeasRestoreMarker()	FileName, SystemId, ControlFlag, MarkerIdPtr	Restore a marker from disk.
MmeasSaveMarker()	FileName, MarkerId, ControlFlag	Save a marker to disk.
MmeasSetMarker()	MarkerId, CharacteristicToSet, FirstValue, SecondValue	Set a marker characteristic parameter.

The geometric model finder module

The geometric model finder module is a set of functions which finds models, or sets or image characteristics, in a target image using a geometric search method.

Model finder commands	Command parameters	Description
MmodAlloc()	SystemId, ModelFinderType, ControlFlag, ContextIdPtr	Allocate a model finder context.
MmodAllocResult()	SystemId, ControlFlag, ModResultIdPtr	Allocate a model finder result buffer.
MmodControl()	ContextId, Index, ControlType, ControlValue	Control a model finder context setting.
MmodDefine()	ContextId, ModelType, Param1, Param2, Param3, Param4, Param5	Add a model to, or delete a model from, a model finder context.
MmodDraw()	GraphContId, ContextOrResultId, DestImageId, Operation, Index, ControlFlag	Draw features of specific models or result occurrences in an image buffer.

Model finder commands	Command parameters	Description
MmodFind()	ContextId, TargetImageId, ModResultId	Find a model finder context's models in the target image.
MmodFree()	ObjectId	Free a model finder context or result buffer.
MmodGetResult()	ResultId, ResultIndex, ResultType, ResultArrayPtr	Get the model finder result values.
MmodInquire()	ImageBufId, ModelId, FindResultId, ResultRange	Inquire information from a specified model finder context.
MmodMask()	ContextId, Index, MaskBufferId, MaskType, ControlFlag	Mask regions of a model.
MmodPreprocess()	ContextId, ControlFlag	Preprocess a model finder context.
MmodRestore()	FileName, SystemId, ControlFlag, ContextIdPtr	Restore a model finder context from disk.
MmodSave()	FileName, ContextId, ControlFlag	Save a model finder context to a file.

The optical character recognition module

The optical character recognition module is a set of functions that can be used to read and verify character strings in images.

Optical character recognition commands	Command parameters	Description
MocrAllocFont()	SystemId, FontType, CharNumber, CharBoxSizeX, CharBoxSizeY, CharOffsetX, CharOffsetY, CharSizeX, CharSizeY, CharThickness, StringLength, InitFlag, FontIdPtr	Allocate an OCR font buffer.
MocrAllocResult()	SystemId, InitFlag, OcrResultIdPtr	Allocate an OCR result buffer.
MocrCalibrateFont()	ImageBufId, FontId, String, TargetCharSizeXMin, TargetCharSizeXMax, TargetCharSizeXStep, TargetCharSizeYMin, TargetCharSizeYMax, TargetCharSizeYStep, Operation	Calibrate font character size to match a sample image.
MocrControl()	FontId, ControlType, ControlValue	Control an OCR parameter setting.

Optical character recognition commands	Command parameters	Description
MocrCopyFont()	ImageBufId, FontId, Operation, CharListString	Copy a font character to or from an image buffer.
MocrFree()	FontOrResultId	Free an OCR font or result buffer.
MocrGetResult()	OcrResultId, ResultToGet, ResultPtr	Read results from an OCR result buffer.
MocrHookFunction()	FontId, HookType, HookHandlerPtr, UserDataPtr	Hook a function to an event.
MocrImportFont()	FileName, FileFormat, Operation, CharListString, FontId	Import font data from file on disk.
MocrInquire()	FontId, InquireType, UserVarPtr	Inquire about font character information.
MocrModifyFont()	FontId, Operation, ControlValue	Invert or resize a font to match the target image characters.
MocrReadString()	ImageBufId, FontId, OcrResultId	Read an unknown string from an image.
MocrRestoreFont()	FileName, Operation, SystemId, FontIdPtr	Restore a font from disk.
MocrSaveFont()	FileName, Operation, FontId	Save an existing font to disk.
MocrSetConstraint()	FontId, CharPos, CharPosType, CharValidString	Set character position constraints.
MocrVerifyString()	ImageBufId, FontId, String, OcrResultId	Verify a known string in an image.

The basic pattern recognition module

The basic pattern recognition or pattern matching module is a set of pattern location functions that can be used for alignment, measurement, and inspection.

Basic pattern recognition commands	Command parameters	Description
MpatAllocAutoModel()	SystemId, SrcImageBufId, SizeX, SizeY, PosUncertaintyX, PosUncertaintyY, ModelType, Mode, ModelIdArrayPtr	Automatically allocate a unique pattern matching model from a source image.
MpatAllocModel()	SystemId, SrcImageBufId, OffX, OffY, SizeX, SizeY, ModelType, ModelIdPtr	Allocate a pattern matching model from a source image.

Basic pattern recognition commands	Command parameters	Description
MpatAllocResult()	SystemId, NbEntries, PatResultIdPtr	Allocate a pattern matching result buffer.
MpatAllocRotatedModel()	SystemId, SrcModelId, Angle, InterpolationMode, ModelType, NewModelIdPtr	Rotate a pattern matching model.
MpatCopy()	ModelId, DestImageBufId, CopyMode	Copy a pattern matching model to an image buffer.
MpatDraw()	GraphContId, ModelOrResultId, DestImageId, Operation, Index, ControlFlag	Draw features of a specific model or result occurrences in an image buffer.
MpatFindModel()	ImageBufId, ModelId, PatResultId	Find a pattern matching model in the target image buffer.
MpatFindMultipleModel()	ImageBufId, ModelIdArray, PatResultIdArray, NumModels, SearchMode	Find multiple pattern matching models in the target image buffer.
MpatFindOrientation()	ImageBufId, ModelId, FindResultId, ResultRange	Find the orientation of an image or of an object in an image.
MpatFree()	PatId	Free a pattern matching buffer (model or result buffer).
MpatGetNumber()	PatResultId, CountPtr	Get the number of model occurrences in the target image.
MpatGetResult()	PatResultId, ResultType, UserArrayPtr	Get the pattern matching result values.
MpatInquire()	PatId, ParamToInquire, UserVarPtr	Inquire about the pattern matching model or the result buffer parameter setting.
MpatPreprocModel()	TypicalImageBufId, ModelId, Mode	Preprocess a pattern matching model.
MpatRead()	SystemId, FileHandle, ModelIdPtr	Read a pattern matching model from an open file.
MpatRestore()	SystemId, FileName, ModelIdPtr	Restore a pattern matching model from disk.
MpatSave()	FileName, ModelId	Save a pattern matching model to disk.
MpatSetAcceptance()	ModelId, AcceptanceThreshold	Set the pattern matching acceptance level.

Basic pattern recognition commands	Command parameters	Description
MpatSetAccuracy()	ModelId, Accuracy	Set the pattern matching positional accuracy.
MpatSetAngle()	ModelId, ControlType, ControlValue	Set the angular search control parameters of a model.
MpatSetCenter()	ModelId, OffX, OffY	Set the pattern matching model center.
MpatSetCertainty()	ModelId, CertaintyThreshold	Set the pattern matching certainty level.
MpatSetDontCare()	ModelId, ImageBufId, OffX, OffY, Value	Set the pattern matching model's "don't care" pixels.
MpatSetNumber()	ModelId, NbOccurrences	Set the expected number of occurrences.
MpatSetPosition()	ModelId, OffX, OffY, SizeX, SizeY	Set the pattern matching search position.
MpatSetSearchParameter()	PatId, Parameter, Value	Set a pattern matching internal search parameter.
MpatSetSpeed()	ModelId, SpeedFactor	Set pattern matching search speed.
MpatWrite()	FileHandle, ModelId	Write a pattern matching model to an open file.

The system allocation and inquiry module

The system allocation and inquiry module supports the allocation and inquiry of systems.

System allocation and inquiry commands	Command parameters	Description
MsysAlloc()	SystemTypePtr, SystemNum, InitFlag, SystemIdPtr	Allocate a system.
MsysControl()	SystemId, ControlType, ControlValue	Control system behavior.
MsysFree()	SystemId	Free a system.
MsysInquire()	SystemId, InquireType, UserVarPtr	Inquire about a system parameter setting.

Chapter 2: The command reference descriptions

The reference description notes

The command descriptions are presented in alphabetical order. Consequently, related commands are grouped together because of their nomenclature. For example, all the data buffer allocation and access module commands begin with the letters *Mbuf*.

The *M_* prefix

All predefined MIL constants have been prefixed with *M_* to avoid conflicts with any previously defined user names.

Parameters

All MIL parameters that end with *Id* expect an allocated MIL object identifier. The letters preceding the *Id* indicate the module with which to allocate the identifier. For example, the variable *BufId* must be a buffer identifier created with **MbufAlloc...()**. If the identifier can be any MIL object identifier (that is, created with any MIL module), it is prefaced simply with the sequence "MIL", for example *MILId*.

Examples

The *Matrox Imaging Library User Guide* describes how the MIL commands are used in typical applications. Code examples are also provided.

Command limitations

Some command descriptions have a *Status* section. This section describes any software or hardware limitation that is currently imposed on the command. Some limitations should be corrected in future revisions, but not necessarily.

Word usage

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

In addition, some of these commands are implemented as *macros*. If you are interested in the definition of the macros, you can find them or their file names in the *mil.h* or *milsetup.h* header file.

The use of the words *board-specific* or *system-specific* indicates that the current subject might be valid only when using certain boards or systems.

Fonts

All commands and parameters are presented in **bold** so that you can quickly scan for them. Predefined constants are presented in a smaller font.

MappAlloc

Synopsis Allocate a MIL application.

Format **MIL_ID MappAlloc(InitFlag, ApplicationIdPtr)**

long InitFlag;	Initialization flag
MIL_ID *ApplicationIdPtr;	Storage location for application identifier

Description This function allocates a MIL application. A MIL application must be allocated prior to using any other MIL functions. The MIL functions use the first application that was user-allocated.

The **InitFlag** parameter specifies the type of initialization to perform on the MIL application. This parameter should be set to one of the following values:

M_DEFAULT	Default initialization.
M_QUIET	Suppress the displaying of error messages during the allocation of the application.

The **ApplicationIdPtr** parameter specifies the address of the variable in which the application identifier is to be written. Since the **MappAlloc()** function also returns the application identifier, you can set this parameter to M_NULL. If allocation fails, M_NULL is written as the identifier.

In multi-thread environments, the application is shared by all threads and **Mapp...()** function calls from any thread apply to all threads unless specifically localized to that thread by specifying an M_THREAD_CURRENT flag when calling the function. However, if a new MIL application is allocated within a thread, using **MappAlloc()**, this thread will be isolated from the shared application and all application controls and hooks will be independent. For example, turning off the error print in the new thread, using **MappControl()**, will not affect the printing of errors by the original shared application; nor will such a command called from a thread attached to the original application affect the new application.

Note, upon allocation of a MIL application, a default system (M_SYSTEM_HOST) is automatically allocated. This default Host system can be used in MIL function calls by specifying M_DEFAULT_HOST wherever a system identifier is required.

In addition, a default graphic context is also allocated upon allocation of a MIL application. This default graphic context can be used in MIL graphic function calls by specifying `M_DEFAULT` wherever a graphic context identifier is required.

In multi-thread applications, a default graphic context is allocated for each thread in order to avoid inter-thread interference.

Return value The returned value is the application identifier. If allocation fails, `M_NULL` is returned as the identifier.

See also `MappFree()`, `MappAllocDefault()`

The **ApplicationIdPtr** parameter specifies the address of the variable in which the application identifier is to be written. Upon execution of this function, the default application specified in the *milsetup.h* file is allocated and its identifier returned. Instead of using **MappAllocDefault()**, you can use **MappAlloc()** to allocate an application. Note, an application must be allocated in order to allocate any other object in MIL.

The **SystemIdPtr** parameter specifies the address of the variable in which the system identifier is to be written. Upon execution of this function, the default system specified in the *milsetup.h* file is allocated and its identifier returned. Instead of using **MappAllocDefault()**, you can use **MsysAlloc()** to allocate a system. **MappAlloc()** will also allocate a default Host system. Note, a system must be allocated in order to allocate any other objects on it (display, digitizer or data buffers).

The **DisplayIdPtr** parameter specifies the address of the variable in which the display identifier is to be written. If this parameter is set to `M_NULL`, a display is not allocated; otherwise, the default display specified in the *milsetup.h* file is allocated and its identifier returned.

The **DigIdPtr** parameter specifies the address of the variable in which the digitizer identifier is to be written. If this parameter is set to `M_NULL`, a digitizer is not allocated; otherwise, the default digitizer specified in the *milsetup.h* file is allocated and its identifier returned.

The **ImageBufIdPtr** parameter specifies the address of the variable in which the image buffer identifier is to be written. If this parameter is set to `M_NULL`, an image buffer is not allocated; otherwise, the default image buffer specified in the *milsetup.h* file is allocated and its identifier returned. It is then cleared and displayed on the system's display screen.

The installation utility modifies the *milsetup.h* header file to create the appropriate macros and customize the default setup. If the installation utility is not executed, the default state supported will be undefined.

After installation, if you want to change the default state of **MappAllocDefault()**, edit *milsetup.h* to suit your needs.

Note, if a digitizer is specified and the default camera type (`M_DEF_DIGITIZER_FORMAT`) in the *milsetup.h* file is a 3-band color (RGB) type, then a 3-band image buffer will be allocated by default; otherwise, a 1-band image buffer will be allocated.

Example For example, a typical default setup for a Genesis board in its power-up state with one input device (RS-170 camera) and one default image buffer (full-screen size) on the display is:

```
MappAllocDefault(M_COMPLETE, &System, &Display, &Digitizer, &ImageBuffer);
```

If, for example, you don't need to acquire data from the camera but want to perform the rest of the above setup, you would make the following call:

```
MappAllocDefault(M_COMPLETE, &System, &Display, M_NULL, &ImageBuffer);
```

Note, upon execution of this function, a default graphics context is automatically allocated. This default graphics context can be used in MIL graphic function calls by specifying M_DEFAULT wherever a graphic context identifier is required.

See also **MappFreeDefault()**, **MappAlloc()**, **MsysAlloc()**, **MdispAlloc()**, **MdigAlloc()**, **MbufAllocColor()**, **MbufAlloc1d()**, **MbufAlloc2d()**

MappControl

Synopsis Control an application environment setting.

Format **void MappControl(ControlType, ControlValue)**

long ControlType;	Type of event to control
long ControlValue;	Flag to control event

Description This function controls the output of error messages to the screen, the output of function names and parameters to the screen at the start and end of MIL functions, and parameter checking at the start of MIL functions. It also controls the processing and memory compensation modes.

In multi-thread environments, a **MappControl()** call applies to all application threads running MIL, unless specifically limited to the calling thread by adding `M_THREAD_CURRENT` to the **ControlType** parameter. When you override settings for a specific thread, a subsequent call to change that setting from a global level will not affect that thread.

For example, **MappControl**(`M_TRACE`, `M_PRINT_ENABLE`), called from any application thread, enables trace printing in all threads running MIL. However, **MappControl**(`M_TRACE+M_THREAD_CURRENT`, `M_PRINT_ENABLE`) will enable trace printing in the currently running thread only and will ignore calls from other threads that try to change trace printing.

The **ControlType** and **ControlValue** parameters specify the type of event to control and the setting with which to control the event respectively. These parameters should be set according to the following combinations. To limit the effect to the current thread, add `M_THREAD_CURRENT` to the control type.

ControlType	ControlValue	Result
<code>M_ERROR</code>	<code>M_PRINT_ENABLE</code>	Enable printing of error messages (default).
<code>M_ERROR</code>	<code>M_PRINT_DISABLE</code>	Disable printing of error messages. If error printing is disabled, you can still check for error, using MappGetError() .
<code>M_TRACE</code>	<code>M_PRINT_ENABLE</code>	Enable printing of function names and parameters.

ControlType	ControlValue	Result
M_TRACE	M_PRINT_DISABLE	Disable printing of function names and parameters (default).
M_PARAMETER	M_CHECK_ENABLE	Enable checking of parameters (default).
M_PARAMETER	M_CHECK_DISABLE	Disable checking of parameters. Note, if parameter checking is disabled to accelerate an application, unpredictable behavior can be expected when passing invalid parameters to a function.
M_PROCESSING	M_COMPENSATION_ENABLE	Enable processing compensation; if your system cannot perform a certain processing operation due to its limitations, processing will be done by the Host (default).
M_PROCESSING	M_COMPENSATION_DISABLE	Disable processing compensation.
M_MEMORY	M_COMPENSATION_ENABLE	Enable memory compensation; if your system cannot perform a certain memory (buffer) allocation due to insufficient memory (default).
M_MEMORY	M_COMPENSATION_DISABLE	Disable memory compensation.

See also **MappGetError()**, **MappHookFunction()**, **MappInquire()**

MappControlThread

Synopsis Allocate/control MIL application thread(s) or events.

Format **long MappControlThread(ControlId, ControlType, ControlValue, ControlVarPtr)**

MIL_ID ControlId;	Thread or Event identifier
long ControlType;	Type of control set on thread or event
long ControlValue;	Value of control setting
long *ControlVarPtr;	Storage location for returned value

Description This function allocates/controls MIL application threads or events.

A MIL thread is a command stream used to send MIL commands to the various allocated MIL systems. MIL automatically allocates a MIL thread for each existing HOST thread that is using MIL. **MappControlThread()** allows you to synchronize MIL threads running on the Host and/or various MIL systems.

A MIL event is a marker that can be inserted between commands sent to a given thread. Its state can be set to either M_SINGALED or M_NOT_SINGALED in a given thread and can be inquired about or waited for (**MappControlThread(Event, M_EVENT_WAIT,...)**), until in M_SINGALED state, by other threads in order to monitor the execution of commands.

The event can be one of the following reset types:

Auto-Reset:	Calling MappControlThread(Event, M_EVENT_SET,...) , sets or resets the event state to M_SINGALED or M_NOT_SINGALED. When in M_SINGALED state, the event is automatically reset to M_NOT_SINGALED when a call to MappControlThread(Event, M_EVENT_WAIT, M_DEFAULT,...) returns. This type of event is useful in applications where only one thread waits on a specific event.
Manual-Reset:	Calling MappControlThread(Event, M_EVENT_SET,...) , sets or resets the event state to M_SINGALED or M_NOT_SINGALED. The event state remains unchanged until an explicit call to MappControlThread(Event, M_EVENT_SET,...) is issued. This type of event is useful when multiple threads wait on a specific event.

The **ControlId** parameter specifies the identifier of the thread or event to be controlled. If set to M_DEFAULT, it uses the default MIL thread/event identifier associated with the Host thread. The thread or event can be user-allocated using the M_THREAD_ALLOC or M_EVENT_ALLOC **ControlType** of **MappControlThread()**.

The **ControlType** and **ControlValue** parameters specify the thread or event control operation to be performed. These parameters can be set to the following combinations:

Thread ControlType	ControlValue	Result
M_THREAD_ALLOC	M_DEFAULT	Create a new selectable MIL thread on a multi-thread system (such as Genesis) and return its MIL_ID. Under Windows NT, MIL automatically allocates a default MIL thread for each existing Host thread. Note, ControlId must be set to M_DEFAULT.
M_THREAD_FREE	M_DEFAULT	Free an existing MIL thread. Note that default MIL threads will be automatically freed. *
M_THREAD_SELECT	M_DEFAULT	Select the MIL thread to which subsequent MIL commands will be sent. *
M_THREAD_WAIT	M_DEFAULT	Synchronize commands sent to a thread. Force a wait for completion of all commands currently executing in the thread. Useful for commands sent to systems allowing an immediate return (before execution is actually completed).*
M_THREAD_MODE	M_SYNCHRONOUS	MIL commands sent to the thread are completed (execution terminated) before returning.*
	M_ASYNCHRONOUS	MIL commands sent to the thread return immediately (when the system and command allow an immediate return). (default) *
M_THREAD_IO_MODE	M_SYNCHRONOUS	MIL commands MbufGet...() and MbufPut...() sent to the thread wait, before executing, for the completion of previous MIL commands sent in the thread (default).*
	M_ASYNCHRONOUS	MIL commands MbufGet...() and MbufPut...() sent to the thread execute immediately.*
* No return value is required. ControlVarPtr should be set to M_NULL.		

Event ControlType	ControlValue	Result
M_EVENT_ALLOC	(any of the values listed below)	Create a new MIL synchronization event and return its MIL ID. Note, ControlId must be set to M_DEFAULT.
	M_DEFAULT or M_NOT_SIGNED+M_AUTO_RESET	Event is initialized as M_NOT_SIGNED and as an Auto-Reset type.
	M_SIGNED+M_AUTO_RESET	Event is initialized as M_SIGNED and as an Auto-Reset type.
	M_NOT_SIGNED+M_MANUAL_RESET	Event is initialized as M_NOT_SIGNED and as an Manual-Reset type.
	M_SIGNED+M_MANUAL_RESET	Event is initialized as M_SIGNED and as an Manual-Reset type.
M_EVENT_FREE	M_DEFAULT	Free an existing MIL event.*
M_EVENT_SET	M_SIGNED or M_NOT_SIGNED	Set a MIL event to the specified state.*
M_EVENT_WAIT	M_DEFAULT	Wait for the specified event to be in an M_SIGNED state. If the event is auto-reset, resets to M_NOT_SIGNED after the wait call is returned.*
M_EVENT_STATE	M_DEFAULT	Inquire the state of the MIL event. The return value can be: M_SIGNED or M_NOT_SIGNED.
* No return value is required. ControlVarPtr should be set to M_NULL.		

The **ControlVarPtr** parameter specifies a pointer to the user variable where the return value is to be written. Specify M_NULL if no return value is required (see footnotes of control tables).

Return value The returned value is the requested event state, cast to a long. If no information was requested (controls were only set), the returned value is not valid.

Example mthread.c

MappFree

Synopsis Free a MIL application.

Format **void MappFree(ApplicationId)**

MIL_ID ApplicationId;	Application identifier
-----------------------	------------------------

Description This function deallocates a MIL application previously allocated with **MappAlloc()**.

Prior to freeing a MIL application, ensure that all allocated systems, buffers, displays, and digitizers are freed. **MappFree()** must be the last function called in a MIL application; no other MIL command can be executed after a call to this function.

Note, if you use **MappAllocDefault()** to allocate the default MIL application, you must use **MappFreeDefault()** to free the application.

The **ApplicationId** parameter specifies the application to free.

See also **MappAlloc()**, **MappFreeDefault()**

MappFreeDefault

Synopsis Free MIL application defaults.

Format **void MappFreeDefault(ApplicationId, SystemId, DisplayId, DigId, ImageBufId)**

MIL_ID ApplicationId;	Application identifier
MIL_ID SystemId;	System identifier
MIL_ID DisplayId;	Display identifier
MIL_ID DigId;	Digitizer identifier
MIL_ID ImageBufId;	Image buffer identifier

Description This macro frees the MIL application defaults that were allocated with the **MappAllocDefault()** macro (located in *milsetup.h*). Note, this command does not affect what is being displayed on the system’s display; if you want to clear the display, you should do so, using **MdispDeselect()**, before calling **MappFreeDefault()**.

The **ApplicationId** parameter specifies the identifier of the application to deallocate.

The **SystemId** parameter specifies the identifier of the system to deallocate.

The **DisplayId** parameter specifies the identifier of the display to deallocate. If set to M_NULL, no display is deallocated.

The **DigId** parameter specifies the identifier of the digitizer to deallocate. If set to M_NULL, no digitizer is deallocated.

The **ImageBufId** parameter specifies the identifier of the image buffer to deallocate. If set to M_NULL, no buffer is deallocated.

See also **MappAllocDefault(), MappFree(), MsysFree(), MdispFree(), MdigFree(), MbufFree()**

MappGetError

Synopsis Get error codes and related information.

Format `long MappGetError(ErrorType, ErrorPtr)`

<code>long ErrorType;</code>	Error type
<code>void *ErrorPtr;</code>	Storage location for information

Description This function obtains current or global system error codes, subcodes, messages, submessages, function codes and function names. This function allows you to check for errors after each MIL function call or to get the first error that occurred after a series of MIL function calls.

A typical use of this function is to check whether a buffer allocation call was successful (**MbufAllocColor()**, **MbufAlloc1d()**, and **MbufAlloc2d()**).

This function can also be used when error-reporting to the screen has been disabled, using **MappControl()**, and you want to obtain information about a detected error.

In multi-thread environments, an **MappGetError()** call returns the error of the current thread or, if none, checks for errors in the other threads running MIL. To return only errors in the current thread, add `M_THREAD_CURRENT` to the **ErrorType** parameter (`M_CURRENT+M_THREAD_CURRENT`).

The **ErrorType** parameter specifies the error type. This parameter can be set to one of the following:

ErrorType	Description
<code>M_CURRENT</code>	Get the error code returned by the last command call. The system current-error code is reset to <code>M_NULL_ERROR</code> before each MIL function call and is set to a specific error code if an error occurs while trying to execute the function.
<code>M_CURRENT_SUB_NB</code>	Get the number of error subcodes associated with the current error.
<code>M_CURRENT_SUB_1...3</code>	Get the n^{th} error subcode returned by the last command call. Note, when there is no error, the error subcode(s) is set to <code>M_NULL_ERROR</code> .
<code>M_CURRENT_FCT</code>	Get the function code associated with the current error.

ErrorType	Description
M_CURRENT+ M_MESSAGE	Get the error message associated with the current error. The system current- error message is reset to "NULL" before each MIL function call and is set to a specific error message if an error occurs while trying to execute the function.
M_CURRENT_SUB_1...3+ M_MESSAGE	Get the n th error submessage associated with the current error.
M_CURRENT_FCT+ M_MESSAGE	Get the function name associated with the current error.
M_GLOBAL	Get the error code of the first error that has occurred since the last call to MappGetError (M_GLOBAL...). The global system-error code is reset to M_NULL_ERROR after each MappGetError() call with this setting.
M_GLOBAL_SUB_NB	Get the number of error subcodes associated with the first error that occurred since the last call to MappGetError (M_GLOBAL...).
M_GLOBAL_SUB_1...3	Get the n th error subcode of the first error that has occurred since the call to MappGetError (M_GLOBAL...). Note, when there is no error, the error subcode(s) is set to M_NULL_ERROR.
M_GLOBAL_FCT	Get the function code associated with the first error that has occurred since the last call to MappGetError (M_GLOBAL...).
M_GLOBAL+ M_MESSAGE	Get the error message associated with the first error that has occurred since the last call to MappGetError (M_GLOBAL...).
M_GLOBAL_SUB_1...3+ M_MESSAGE	Get the n th error submessage associated with the first error that has occurred since the last call to MappGetError (M_GLOBAL...).
M_GLOBAL_FCT+ M_MESSAGE	Get the function name associated with the first error that has occurred since the last call to MappGetError (M_GLOBAL...).

The **ErrorPtr** parameter specifies the address of the variable in which the requested information is to be written. If the error code is read and it is equal to M_NULL_ERROR, no error has occurred. Since the **MappGetError()** function also returns the error code or subcode, you can set this parameter to M_NULL.

This variable should be a pointer to a long when getting error codes, subcodes, number of subcodes, and function codes. This variable should be a pointer to a string when getting messages, submessages and function names. The string must be at least M_ERROR_MESSAGE_SIZE characters in size.

Return value The returned value is the requested error code or subcode. When getting error messages, submessages, and function names, the returned value is the associated error code.

Example mshift.c

MappGetHookInfo

Synopsis Get information about a hooked event.

Format **long MappGetHookInfo(EventId, InfoType, UserVarPtr)**

MIL_ID EventId;	Event identifier received from the hook-handler function
long InfoType;	Type of information to get
void *UserVarPtr;	Storage location for the information

Description This function 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 (see **MappHookFunction()**).

Note that functions hooked to an event execute on a distinct thread. This permits the functions to run asynchronously from the operation that fired the event and from functions hooked to other events. Although there is a small queue to permit a certain amount of overlap, hooked functions should not take longer to execute than the period in which two of their associated events can occur. You cannot determine the instance of the event that fired the function, and even if this were possible, this information would generally not be very useful. Typically, a hooked function performs the minimum number of operations required and, if necessary, performs longer processes by launching other threads.

The **EventId** parameter specifies the event identifier received from the hook-handler function.

The **InfoType** parameter specifies the type of information to get.

If the hook handler was called with an M_ERROR_CURRENT **HookType**, supported values for **InfoType** are:

InfoType	Description
M_CURRENT	Error code.
M_CURRENT_SUB_NB	Number of error subcodes.
M_CURRENT_SUB_1	Error subcode 1.
M_CURRENT_SUB_2	Error subcode 2.
M_CURRENT_SUB_3	Error subcode 3.
M_CURRENT_FCT	Function code that caused an error.

InfoType	Description
M_MESSAGE+M_CURRENT	Error message.
M_MESSAGE+M_CURRENT_SUB_1	Error submessage 1.
M_MESSAGE+M_CURRENT_SUB_2	Error submessage 2.
M_MESSAGE+M_CURRENT_SUB_3	Error submessage 3.
M_MESSAGE+M_CURRENT_FCT	Name of the function that caused an error.

If the hook-handler function was called with an M_ERROR_GLOBAL **HookType**, supported values for **InfoType** are:

InfoType	Description
M_GLOBAL	Error code.
M_GLOBAL_SUB_NB	Number of error subcodes.
M_GLOBAL_SUB_1	Error subcode 1.
M_GLOBAL_SUB_2	Error subcode 2.
M_GLOBAL_SUB_3	Error subcode 3.
M_GLOBAL_FCT	Function code that caused an error.
M_MESSAGE+M_GLOBAL	Error message.
M_MESSAGE+M_GLOBAL_SUB_1	Error submessage 1.
M_MESSAGE+M_GLOBAL_SUB_2	Error submessage 2.
M_MESSAGE+M_GLOBAL_SUB_3	Error submessage 3.
M_MESSAGE+M_GLOBAL_FCT	Function name that caused an error.

If the hook-handler function was called with an M_TRACE_START or M_TRACE_END **HookType**, supported values for **InfoType** are:

InfoType	Description
M_CURRENT_FCT	Code of the function that just started or ended.
M_MESSAGE+M_CURRENT_FCT	Name of the function that just started or ended.
M_PARAM_NB	Number of parameters associated to the function call.
M_PARAM_TYPE+n.	Data type of the n th parameter. This can be: M_TYPE_LONG, M_TYPE_SHORT, M_TYPE_CHAR, M_TYPE_DOUBLE, M_TYPE_PTR, M_TYPE_MIL_ID, or M_TYPE_STRING. (The pointer to a string is invalid after exiting the hook function. For future use, copy and save it.)
M_PARAM_VALUE+n	Value of the n th parameter.

If the hook handler was called with an `M_MODIFIED_BUFFER` **HookType**, supported values for **InfoType** are:

InfoType	Description
<code>M_MODIFIED_BUFFER + M_BUFFER_ID</code>	MIL identifier of the modified buffer.
<code>M_MODIFIED_BUFFER + M_REGION_OFFSET_X</code>	X offset, of the modified region in the buffer, as a long value.
<code>M_MODIFIED_BUFFER + M_REGION_OFFSET_Y</code>	Y offset, of the modified region in the buffer, as a long value.
<code>M_MODIFIED_BUFFER + M_REGION_SIZE_X</code>	Width, of the modified region in the buffer, as a long value.
<code>M_MODIFIED_BUFFER + M_REGION_SIZE_Y</code>	Height, of the modified region in the buffer, as a long value.

The **UserVarPtr** parameter specifies the address of the variable in which the requested information is to be written.

UserVarPtr should be a pointer to a long when getting error codes, subcodes, number of subcodes, function codes and parameter types. It should be a pointer to a string when getting error messages, submessages, and function names. The string must be at least `M_ERROR_MESSAGE_SIZE` characters in size. When getting parameter values, **UserVarPtr** should be a pointer to the type specified by the returned value of an `M_PARAM_TYPE+n` request in a previous call to this function.

Return value The returned value is `M_NULL` if successful; a non `M_NULL` value on error, without logging any errors in the application.

See also **MappHookFunction()**

MappHookFunction

Synopsis Hook a function to an event.

Format `void MappHookFunction(HookType, HookHandlerPtr, UserDataPtr)`

<code>long HookType;</code>	Type of event to hook
<code>MAPPHOOKFCTPTR HookHandlerPtr;</code>	Pointer to hook function
<code>void *UserDataPtr;</code>	User data pointer

Description This function allows you to attach or detach a user-defined function to a specified application event. Once a hook-handler function is defined and hooked to an event, it is automatically called when the event occurs.

Note that functions hooked to an event execute on a distinct thread. This permits the functions to run asynchronously from the operation that fired the event and from functions hooked to other events. Although there is a small queue to permit a certain amount of overlap, hooked functions should not take longer to execute than the period in which two of their associated events can occur. You cannot determine the instance of the event that fired the function, and even if this were possible, this information would generally not be very useful. Typically, a hooked function performs the minimum number of operations required and, if necessary, performs longer processes by launching other threads.

You can hook more than one function to an event by making separate calls to **MappHookFunction()** for each function that you want to hook. MIL automatically chains and keeps an internal list of all these hooked functions. When a function is hooked, this new function is added to the end of the list. When the event happens, all user-defined functions in the list will be executed in the same order that they were hooked to the event. You can also remove any function from the list; in this case, MIL preserves the order of the remaining functions in the list.

The user can obtain information concerning the event, using **MappGetHookInfo()**, and take appropriate action before returning control to the application.

This function is typically used to trap errors that occur in an application without checking every MIL command execution with **MappGetError()** or to detect the start or end of certain MIL commands.

In multi-thread environments, an **MappHookFunction()** call hooks or unhooks the function specified by **HookHandlerPtr** to all application threads running MIL, unless specifically limited to the calling thread by adding **M_THREAD_CURRENT** to the **HookType** parameter (for example, to call the hook-handler function only for errors occurring in the current thread, specify **M_ERROR_CURRENT+M_THREAD_CURRENT** as the **HookType** parameter).

The **HookType** parameter specifies the event type. This parameter can be set to one of the following values. Note that a hooked function must be unhooked by combining the **HookType** parameter with **M_UNHOOK**.

HookType	Description
M_ERROR_CURRENT	Call the hook-handler function each time an error occurs.
M_ERROR_GLOBAL	Call the hook-handler function when the first error occurs in a series of MIL calls.
M_TRACE_START	Call the hook-handler function at the start of each MIL function.
M_TRACE_END	Call the hook-handler function at the end of each MIL function.
M_UNHOOK +M_ERROR_CURRENT	Detach the hook-handler function being called each time an error occurs.
M_UNHOOK +M_ERROR_GLOBAL	Detach the hook-handler function being called when the first error occurs in a series of MIL calls.
M_UNHOOK +M_TRACE_START	Detach the hook-handler function being called at the start of each MIL function.
M_UNHOOK +M_TRACE_END	Detach the hook-handler function being called at the end of each MIL function.
M_UNHOOK +M_ERROR_FATAL	Detach the hook-handler function being called before a fatal-error exit.

When setting the **HookType** parameter to **M_TRACE_START** or **M_TRACE_END**, the only MIL function that can be called in a hook function is **MappGetHookInfo()**, otherwise infinite recursion will occur. Note that when hooking the errors, you could end up with infinite recursion if MIL functions generating an error are called within the same hook function.

The **HookHandlerPtr** parameter specifies the address of the function that should be called when an event occurs.

The hook-handler function, pointed to by **HookHandlerPtr**, must be declared as follows:

```
long MFTYPE HookHandler(HookType, EventId, UserDataPtr);
```

long HookType;	Type of event hooked
MIL_ID EventId;	Event identifier to pass to MappGetHookInfo() when inquiring about the hooked event
void MPTYPE *UserDataPtr;	user data pointer

Upon successful completion, the hook-handler function should return M_NULL. Note, MAPPHOOKFCTPTR, MFTYPE and MPTYPE are reserved MIL predefined types for functions and data pointers.

The **UserDataPtr** parameter specifies the address of the user data that you want to make available to the hook-handler function. This address is passed to the hook-handler function, through its *UserDataPtr* parameter, when the specified event occurs. Set this parameter to M_NULL if not used.

Return value The original prototype of this function has been kept for backwards compatibility. However, because of the current chaining method, the function always returns null.

See also **MappGetHookInfo()**, **MappControl()**, **MappGetError()**

MappInquire

Synopsis Inquire about the application parameter setting.

Format **long MappInquire(InquireType, UserVarPtr)**

long InquireType;	Type of information to inquire
void *UserVarPtr;	Storage location for inquired information

Description This function inquires about the specified application control, processing mode, or memory setting.

The **InquireType** parameter specifies the type of information to inquire about. This parameter can be set to one of the following values. See **MappControl()** for more information about these values. In multi-thread environments, you can inquire the status of a control from any thread; however, to inquire the status of a thread-specific parameter, add `M_THREAD_CURRENT` to the **InquireType** parameter (`M_ERROR+M_THREAD_CURRENT`).

InquireType	Description
M_CURRENT_APPLICATION	Identifier of the current MIL application, if any. Returns 0, without generating an error, if no application is allocated.
M_ERROR	Error printing mode (<code>M_PRINT_ENABLE</code> or <code>M_PRINT_DISABLE</code>).
M_MEMORY	Memory compensation mode (<code>M_COMPENSATION_ENABLE</code> or <code>M_COMPENSATION_DISABLE</code>).
M_LICENSE_FINGERPRINT	The hardware component upon which the system fingerprint is based. M_MATROX_VGA_FINGERPRINT M_MATROX_ETHERNET_FINGERPRINT M_MATROX_HARD_ID_KEY_FINGERPRINT M_MATROX_BOARD_FINGERPRINT

InquireType	Description
M_LICENSE_MODULES	The module for which there is a valid license. The value returned will be a bitwise combination of the following: M_LICENSE_LITE M_LICENSE_DEBUG M_LICENSE_IM M_LICENSE_CODE M_LICENSE_MEAS M_LICENSE_PAT M_LICENSE_MOD M_LICENSE_JPEG2000 M_LICENSE_BLOB M_LICENSE_CAL M_LICENSE_OCR M_LICENSE_JPEGSTD
M_PARAMETER	Parameter checking mode (M_CHECK_ENABLE or M_CHECK_DISABLE).
M_PROCESSING	Processing compensation mode (M_COMPENSATION_ENABLE or M_COMPENSATION_DISABLE).
M_TRACE	Trace printing mode (M_PRINT_ENABLE or M_PRINT_DISABLE).
M_VERSION	Version of MIL library.
M_OBJECT_TYPE+(MILId)	Type of the specified MIL object. (M_APPLICATION, M_SYSTEM, M_LUT, M_DISPLAY, M_DIGITIZER, M_IMAGE, M_KERNEL, M_STRUCT_ELEMENT, M_ARRAY, M_HIST_LIST, M_EXTREME_LIST, M_PROJ_LIST, M_EVENT_LIST, M_COUNT_LIST, M_BLOB_OBJECT, M_PAT_OBJECT, M_GRAPHIC_CONTEXT, M_OCR_OBJECT, M_CAL_OBJECT, M_CODE_OBJECT, M_MEAS_OBJECT, M_MOD_OBJECT, M_USER_OBJECT_1, or M_USER_OBJECT_2)

The **UserVarPtr** parameter specifies the address of the variable in which the requested information is to be written. Since the **MapplInquire()** function also returns the requested information, you can set this parameter to M_NULL. The variable should be a pointer to a long, unless you are using M_VERSION, in which case it should be a pointer to a double.

Return value The returned value is the requested system information cast to long.

See also **MapplControl()**

MappModify

Synopsis Modify specified MIL object(s).

Format **void MappModify(FirstMILId, SecondMILId, ModificationType, ModificationFlag)**

MIL_ID FirstMILId;	First MIL object identifier
MIL_ID SecondMILId;	Second MIL object identifier
long ModificationType;	Type of modification
long ModificationFlag;	Modification flag

Description This function modifies the specified MIL object(s) according to the specified operation.

The **FirstMILId** parameter specifies the identifier of the first MIL object to be modified.

The **SecondMILId** parameter specifies the identifier of the second MIL object (if applicable) to be modified.

The **ModificationType** parameter specifies the desired operation. This parameter should be set to the following value:

M_SWAP_ID	Exchange the identifiers of the first and second specified MIL objects
-----------	--

The **ModificationFlag** parameter should be set to M_NULL.

MappTimer

Synopsis Control the MIL timer.

Format `void MappTimer(ControlValue, TimePtr)`

long ControlValue;	Type of modification
double *TimePtr;	Storage location for time

Description This function controls the MIL timer. This is useful for benchmarking operations in a MIL application. The MIL timer resolution varies according to the hardware and operating system used.

The **ControlValue** parameter specifies the control to exert on the MIL timer. It can be set to one of the following:

ControlValue	Description
M_TIMER_RESET	Resets a MIL timer to zero.
M_TIMER_READ	Reads the time (in seconds) of the MIL timer, since the last reset.
M_TIMER_RESOLUTION	Reads the MIL timer resolution (in seconds).
M_TIMER_WAIT	Wait for the specified period of time (in seconds) before returning.

The **TimerPtr** parameter specifies the address of the variable in which to store the timer information produced by the M_TIMER_READ or M_TIMER_RESOLUTION controls. For the M_TIMER_WAIT control, **TimerPtr** specifies the variable from which to read the timer information. For M_TIMER_RESET, set **TimerPtr** to M_NULL.

Example mpatrot.c

MblobAllocFeatureList

Synopsis

Allocate a blob analysis feature list.

Format

MIL_ID SystemId;

MIL_ID *FeatureListIdPtr;

System identifier

Storage location for feature list identifier

Description

This function allocates a feature list. The feature list holds the feature(s) to be calculated by **MblobCalculate()**. You must specify which feature(s) to calculate, using **MblobSelectFeature()**, **MblobSelectFeret()**, and **MblobSelectMoment()**. Immediately after allocation, no features are selected in the feature list. When the feature list is no longer required, release it, using **MblobFree()**.

The **SystemId** parameter specifies the system on which the feature list will be allocated. This parameter must be set to a valid system identifier, M_DEFAULT_HOST or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to allocate the buffer (it can be the Host system or any already allocated system).

The **FeatureListIdPtr** parameter specifies the address of the variable in which the feature list identifier will be written. Since the **MblobAllocFeatureList()** function also returns the feature list identifier, you can set this parameter to M_NULL.

Return value

The returned value is the feature list identifier that you will use to select features to be calculated.

Example

mblob.c

See also

MblobSelectFeature(), **MblobSelectMoment()**, **MblobSelectFeret()**, **MblobCalculate()**, **MblobFree()**

MblobAllocResult

Synopsis Allocate a blob analysis result buffer.

Format **MIL_ID MblobAllocResult(SystemId, BlobResIdPtr)**

MIL_ID SystemId;	System identifier
MIL_ID *BlobResIdPtr;	Storage location for blob analysis result buffer identifier

Description This function allocates a result buffer used to store blob analysis results.

Each blob creates a separate result entry in the blob analysis result buffer. You can retrieve blob analysis results from a result buffer, using **MblobGetResult()** or **MblobGetResultSingle()**. Use the latter to obtain results for a single blob. For more specific results, you can call **MblobGetLabel()** and **MblobGetRuns()**. When the result buffer is no longer required, release it, using **MblobFree()**.

The **SystemId** parameter specifies the system on which the feature list will be allocated. This parameter must be set to a valid system identifier, M_DEFAULT_HOST or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to allocate the buffer (it can be the Host system or any already allocated system).

The **BlobResIdPtr** parameter specifies the address of the variable in which the blob analysis result buffer identifier is to be written. Since the **MblobAllocResult()** function also returns the blob analysis result buffer identifier, you can set this parameter to M_NULL.

The default processing modes for the result buffer are as follows:

Processing Mode	Default Value
M_BLOB_IDENTIFY	M_INDIVIDUAL
M_LATTICE	M_8_CONNECTED
M_FOREGROUND_VALUE	M_NONZERO
M_NUMBER_OF_FERETS	8
M_PIXEL_ASPECT_RATIO	1.0
M_IDENTIFIER_TYPE	M_GRAYSCALE

Use **MblobControl()** to change these values if necessary.

Return value The returned value is a blob analysis result buffer identifier that you will use in calculations and results inquiry.

See also **MblobFree()**, **MblobControl()**, **MblobCalculate()**, **MblobGetNumber()**, **MblobGetResult()**, **MblobGetResultSingle()**, **MblobGetLabel()**, **MblobGetRuns()**

MblobCalculate

Synopsis Perform blob analysis calculations.

Format void MblobCalculate(BlobIdentImageId, GrayImageId, FeatureListId, BlobResId)

MIL_ID BlobIdentImageId;	Blob identifier image identifier
MIL_ID GrayImageId;	Optional grayscale image identifier
MIL_ID FeatureListId;	Feature list identifier
MIL_ID BlobResId;	Blob analysis result buffer identifier

Description This function calculates the features specified in the given feature list for all currently included blobs in the blob identifier image and stores results in the specified result buffer. Features are added to the feature list with **MblobSelectFeature()**, **MblobSelectFeret()**, and **MblobSelectMoment()**. Specific blobs can be selected using **MblobSelect()**.

Calculations on binary features (such as M_AREA) are performed using only the **BlobIdentImageId** parameter. If a grayscale feature (such as M_MAX_PIXEL) is to be calculated, an image buffer must be specified for the **GrayImageId** parameter. In this case, the blob identifier image will be used to identify the blobs and the pixel values in the grayscale image are used to calculate the features. Note that the blob identifier image and the grayscale image must be the same size.

If several calls are made to **MblobCalculate()** with the same image and result buffer, features calculated in one call remain in the result buffer and are not recalculated in subsequent calls. However, if you then use a result buffer with different images or if you change its processing mode with **MblobControl()**, any results already in the buffer become invalid and will be discarded. Therefore, it is more efficient to use a result buffer exclusively in one processing mode with one blob identifier image (or one identifier/grayscale image pair if grayscale features are needed).

The **BlobIdentImageId** parameter specifies the blob identifier image that will be used in calculations. The blob identifier image identifies each blob as a group of touching pixels in the current foreground state (zero or non-zero, depending on the value assigned to M_FOREGROUND_VALUE processing mode in **MblobControl()**). The current M_LATTICE processing

mode (also set with **MblobControl()**), determines when to consider pixels as touching. The blob identifier image must be an unsigned, single band, packed binary, 8-bit or 16-bit grayscale image buffer.

If the identifier image has previously been binarized so that it contains only two extreme values (0 and 1 for 1-bit images, 0 and 0xff for 8-bit images, and 0 and 0xffff for 16-bit images), blob analysis can be performed a little faster. However, you must first change the `M_IDENTIFIER_TYPE` to `M_BINARY`, using **MblobControl()**, to let MIL know that the identifier image has only two states.

Depending on the `M_BLOB_IDENTIFICATION` mode (set with **MblobControl()**), **MblobCalculate()** either treats each blob individually (`M_INDIVIDUAL`), groups all blobs together (`M_WHOLE_IMAGE`), or groups blobs according to their actual pixel value in the blob identifier image (`M_LABELED`).

The **GrayImageId** parameter specifies the grayscale image (not a binary buffer) that will be used to calculate grayscale features. If this parameter is set to `M_NULL`, grayscale features, such as `M_SUM_PIXEL`, cannot be calculated. This parameter is ignored when calculating only binary type features. The grayscale image must be a single band, 8-bit or 16-bit unsigned grayscale image buffer.

The **FeatureListId** parameter specifies the identifier of the feature list buffer, previously allocated with **MblobAllocFeatureList()**, that specifies the feature(s) to calculate.

The **BlobResId** parameter specifies the identifier of a blob analysis result buffer, previously allocated with **MblobAllocResult()**, in which to store calculated results.

Note This function is optimized for packed binary buffers.

Example `mblob.c`

See also **MblobControl()**, **MblobAllocFeatureList()**, **MblobAllocResult()**, **MblobSelectFeature()**, **MblobSelectFeret()**, **MblobSelectMoment()**, **MblobSelect()**, **MblobGetNumber()**, **MblobGetResult()**, **MblobGetResultSingle()**, **MblobGetLabel()**, **MblobGetRuns()**

MblobControl

Synopsis Set a blob analysis processing control.

Format `void MblobControl(BlobResId, ControlType, ControlValue)`

MIL_ID BlobResId;	Blob analysis result buffer identifier
long ControlType;	Processing control to set
double ControlValue;	Value associated with processing control

Description This function changes the processing control associated with the specified blob analysis result buffer. It is normally called immediately after allocating a blob analysis result buffer, using **MblobAllocResult()**, but can be called later (in which case, already calculated results are discarded). If not called, default processing controls and values are used in calculations.

The **BlobResId** parameter specifies the identifier of the blob analysis result buffer.

The **ControlType** and **ControlValue** parameters specify the processing feature to control and the value needed for the control. These two parameters should be set to one of the following (defaults are shown in bold):

ControlType	ControlValue	Description
M_BLOB_IDENTIFICATION	M_INDIVIDUAL	All blobs are measured individually.
	M_WHOLE_IMAGE	All blobs are grouped together.
	M_LABELED	Blobs with the same label value are grouped together.
M_LATTICE	M_8_CONNECTED	Each pixel has 8 neighbors.
	M_4_CONNECTED	Each pixel has 4 neighbors.
M_PIXEL_ASPECT_RATIO	value (default is 1.0)	Pixel width/pixel height.
M_NUMBER_OF_FERETS	value between M_MIN_FERETS and M_MAX_FERETS (default is 8).	The first Feret angle used is always 0°, and the difference between successive angles is 180° / number of Ferets.
M_FOREGROUND_VALUE	M_NONZERO	Blobs consist of non-zero pixels.
	M_ZERO	Blobs consist of zero pixels.
M_IDENTIFIER_TYPE	M_GRAYSCALE	Non-zero pixels can have any value.
	M_BINARY	Non-zero pixels must have the maximum value of the buffer (for example, 0xff for an 8-bit image).

ControlType	ControlValue	Description
M_SAVE_RUNS	M_ENABLE	Calls to MblobCalculate() will save, in the result buffer, run information from the blob identifier image.
	M_DISABLE	Disabling saves time in performing the first call to MblobCalculate() and reduces the memory requirements for the result buffer. However, you cannot use MblobFill() , MblobGetLabel() , MblobGetRuns() , or MblobLabel() . In addition you cannot calculate the chained pixels feature (M_CHAINS), using MblobSelectFeature() .

If the identifier image is already binarized (for example, pixel values for an 8-bit image are either 0 or 0xff), you can change the identifier type to M_BINARY to calculate features faster.

Note, you can use **MblobInquire()** to perform inquiries on specific processing controls associated with a blob analysis result buffer.

Example mblob.c

See also **MblobInquire()**, **MblobCalculate()**, **MblobAllocResult()**

MblobDraw

Synopsis Draw specific features of the blob analysis results in the destination image buffer.

Format **void MblobDraw(GraphContId, ResultId, DestImageId, Operation, Label, ControlFlag)**

MIL_ID GraphContId;	Graphics context identifier
MIL_ID ResultId;	Result buffer identifier
MIL_ID DestImageId;	Destination image buffer identifier
long Operation;	Drawing operation
long Label;	Label of the blob or type of blobs
long ControlFlag;	Control flag

Description This function provides graphics utilities to draw specific blob features in the destination buffer.

The **GraphContId** parameter specifies the graphics context to use. The graphics context must have been previously allocated using **MgraAlloc()**. Set this parameter to M_DEFAULT to use the default graphics context.

The **ResultId** parameter specifies the blob results from which to extract the features to draw.

The **DestImageId** parameter specifies the identifier of the destination image buffer into which to draw. You can also choose to annotate an image non-destructively, by drawing into its display's overlay buffer.

The **Operation** parameter specifies the type of operation to perform. Operations can be added together to draw multiple features at a time. For example, to draw the blob contours, contour holes, and the center of gravity, you would specify M_DRAW_BLOBS_CONTOUR+M_DRAW_HOLES+M_DRAW_POSITION as the **Operation** parameter.

The **Operation** parameter can be set to a combination of the following values:

Operation	Description
M_DRAW_AXIS	Draws a cross at the blobs' center of gravity, respecting the principle and secondary axes. Note that to perform this operation, one feature among M_AXIS_PRINCIPLE_ANGLE or M_AXIS_SECONDARY_ANGLE must have been calculated.
M_DRAW_BLOBS	Draws the blobs. Note that to perform this operation, runs must have been previously saved by setting the MblobControl() M_SAVE_RUNS control type to M_ENABLE.
M_DRAW_BLOBS_CONTOUR	Draws the external outline of the blobs. Note that to perform this operation, runs must have been previously saved by setting the MblobControl() M_SAVE_RUNS control type to M_ENABLE. Also note that when the MblobControl() M_BLOB_IDENTIFICATION control type is set to either M_WHOLE_IMAGE or M_LABELED, both the blob and hole contours will be drawn when performing this operation.
M_DRAW_HOLES	Draws the holes of the blobs. Note that to perform this operation, runs must have been previously saved by setting the MblobControl() M_SAVE_RUNS control type to M_ENABLE and one feature among M_BOX_X_MIN, M_BOX_X_MAX, M_BOX_Y_MIN, or M_BOX_Y_MAX must have been calculated.
M_DRAW_HOLES_CONTOUR	Draws the outline of the blobs' holes. Note that to perform this operation, runs must have been previously saved by setting the MblobControl() M_SAVE_RUNS control type to M_ENABLE. Also note that when the MblobControl() M_BLOB_IDENTIFICATION control type is set to either M_WHOLE_IMAGE or M_LABELED, both the blob and hole contours will be drawn when performing this operation.
M_DRAW_BOX	Draws the bounding box of each blob. Note that to perform this operation, one feature among M_BOX_X_MIN, M_BOX_X_MAX, M_BOX_Y_MIN, or M_BOX_Y_MAX must have been calculated.

Operation	Description
M_DRAW_BOX_CENTER	Draws a cross at the center of the blobs' bounding box. Note that to perform this operation, one feature among M_BOX_X_MIN, M_BOX_X_MAX, M_BOX_Y_MIN, or M_BOX_Y_MAX must have been calculated.
M_DRAW_POSITION or M_DRAW_CENTER_OF_GRAVITY	Draws a cross at the center of gravity of the blobs. Note that to perform this operation, one feature among M_CENTER_OF_GRAVITY_X or M_CENTER_OF_GRAVITY_Y must have been calculated.
M_DRAW_FERET_MIN*	Draws the blobs' minimum Feret diameter, using an H-type line (-), centered at the blobs' position and at the maximum Feret diameter's angle.
M_DRAW_FERET_MAX*	Draws the blobs' maximum Feret diameter, using an H-type line (-), centered at the blobs' position and at the maximum Feret diameter's angle.
M_DRAW_FERET_BOX*	Draws a square box with a width and height of the maximum Feret diameter, centered at the center of the blobs' bounding box, and at the maximum Feret diameter's angle.
*Note that to perform these operations, one feature among M_BOX_X_MIN, M_BOX_X_MAX, M_BOX_Y_MIN, or M_BOX_Y_MAX, and one feature among M_FERET_MIN_DIAMETER, M_FERET_MAX_DIAMETER, M_FERET_MIN_ANGLE and M_FERET_MAX_ANGLE must have been calculated.	

The **Label** parameter specifies the label of the blob to draw and can be set to one of the following:

Label	Description
M_ALL_BLOBS	Draw all blobs. M_DEFAULT always refers to all blobs.
M_INCLUDED_BLOBS	Draw all currently included blobs.
M_EXCLUDED_BLOBS	Draw all currently excluded blobs.

The **ControlFlag** parameter is reserved for future expansion and must be set to M_DEFAULT.

See also MblobCalculate(), MblobControl(), MblobSelectFeature()

MblobFill

Synopsis Draw blobs that meet a specified fill criterion.

Format **void MblobFill(BlobResId, DestImageBufId, Criterion, Value)**

MIL_ID BlobResId;	Blob analysis result buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long Criterion;	Fill criterion
long Value;	Fill value

Description This function draws, in an image, those blobs that meet a specified fill criterion with the specified fill value. Note, only those that meet the criterion are drawn; other blobs and background pixels are not drawn.

This function is often used to remove unwanted (excluded or deleted) blobs from the identifier image (by drawing them with the background color), or to highlight included blobs in a different color. Therefore, an appropriate destination image is the blob identifier image (or a copy of it) associated with the result buffer, or another image buffer that has been cleared.

MblobCalculate() must have been called prior to using this function.

The **BlobResId** parameter specifies the identifier of the blob analysis result buffer.

The **DestImageBufId** parameter specifies the identifier of the destination image buffer. This must be a single band, packed binary, 8, or 16-bit unsigned buffer. Note, this buffer need not be the same size as the original identifier image used to calculate the blobs.

The **Criterion** parameter specifies which blobs to draw with the specified value. This parameter can be set to one of the following values:

M_INCLUDED_BLOBS	Draws all currently included blobs with the specified fill value.
M_EXCLUDED_BLOBS	Draws all currently excluded blobs with the specified fill value.
M_ALL_BLOBS	Draws all blobs with the specified fill value.

The status of the blobs (included or excluded) is taken from the blob analysis result buffer. By default, all blobs in the result buffer are included for future operations. To change the status of a blob, use **MblobSelect()**.

To fill only the blob's borders, add `M_CONTOUR` to the specified criterion. `M_CONTOUR` uses the coordinates obtained from chained pixels to draw the borders; therefore, if you have already calculated `M_CHAINS` with **MblobCalculate()**, `M_CONTOUR` will operate faster.

The **Value** parameter specifies the value with which to fill the blobs that meet the specified criterion. If the destination buffer is binary, this value must be 0 or 1.

Note This function is optimized for packed binary buffers.

See also **MblobCalculate()**, **MblobSelect()**

MblobFree

Synopsis Free the blob analysis result buffer or the feature list.

Format **void MblobFree(BlobId)**

MIL_ID BlobId;	Blob analysis result buffer identifier or feature list buffer identifier
----------------	--

Description This function deletes the specified blob analysis result buffer or feature list and releases any memory associated with it.

The **BlobId** parameter specifies the identifier of the blob analysis result buffer or feature list buffer to free. The buffer must have been successfully allocated, using **MblobAllocResult()** or **MblobAllocFeatureList()**, prior to calling this function.

See also **MblobAllocResult()**, **MblobAllocFeatureList()**

MblobGetLabel

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

Format `long MblobGetLabel(BlobResId, XPos, YPos, LabelVarPtr)`

MIL_ID BlobResId;	Blob analysis result buffer identifier
long XPos;	X coordinate of the blob
long YPos;	Y coordinate of the blob
long *LabelVarPtr;	Storage location for the label value

Description This function gets the label value of a specified blob. Label values are used, for example, to obtain calculation results for single blobs (**MblobGetResultSingle()**). Blob label values must have been generated by calling **MblobCalculate()**.

The **BlobResId** parameter specifies the identifier of the blob analysis result buffer where the labels are stored. Note, this function cannot obtain the label values of blobs that have been **deleted** from the result buffer with **MblobSelect()**.

The **XPos** and **YPos** parameters specify the coordinates of the blob.

The **LabelVarPtr** parameter specifies the address of the variable in which the label value is to be written. Since **MblobGetLabel()** also returns the label value, you can set this parameter to `M_NULL`.

Return value The returned value is the label value of the specified blob. If there is no blob at the specified location, the blob has been deleted, or if the blob's **Xpos** and **Ypos** lie outside the original identifier image, `M_NULL` is returned instead of the label value.

See also **MblobCalculate()**, **MblobGetResult()**, **MblobGetRuns()**

MblobGetNumber

Synopsis Get the number of currently included blobs.

Format **long MblobGetNumber(BlobResId, CountVarPtr)**

MIL_ID BlobResId;	Blob analysis result buffer identifier
long *CountVarPtr;	Storage location for the count

Description This function reads the number of currently included blobs from the specified blob analysis result buffer. All blobs are included unless their status is changed, using **MblobSelect()**. Included blobs will be included in future operations and result retrievals.

This function must be used to determine the number of blob results that will be returned by **MblobGetResultSingle()**.

A call to **MblobCalculate()** must have been made prior to using this function.

The **BlobResId** parameter specifies the identifier of the blob analysis result buffer.

The **CountVarPtr** parameter specifies the address of the variable in which the count is to be written. Since **MblobGetNumber()** also returns the number of selected blobs, you can set this parameter to M_NULL.

Return value The returned value is the number of selected blobs in the specified result buffer.

Example mblob.c

See also **MblobCalculate()**, **MblobGetResult()**, **MblobSelect()**

MblobGetResult

Synopsis Read feature values of the included blobs.

Format void MblobGetResult(BlobResId, Feature, TargetArrayPtr)

MIL_ID BlobResId;	Blob analysis buffer identifier
long Feature;	Type of feature for which to get results
void *TargetArrayPtr;	Array in which to return results

Description This function obtains the results for a specified feature from the blob analysis result buffer.

The **BlobResId** parameter specifies the identifier of the blob analysis result buffer from which to get results.

The **Feature** parameter specifies the feature for which results will be retrieved. See **MblobSelectFeature()**, **MblobSelectFeret()**, and **MblobSelectMoment()** for pre-defined values. The specified feature must have already been calculated with **MblobCalculate()**.

Some features may have been calculated with two results (grayscale and binary). Specifying the feature alone reads the grayscale result. Specifying the feature + M_BINARY reads the binary result. Unless a feature actually has two results, the result (whether grayscale or binary) is read simply by specifying the feature alone.

Results are normally returned in the target array as type "double". If you want results to be returned as a different type, combine the specified feature with M_TYPE_CHAR, M_TYPE_SHORT, M_TYPE_LONG, or M_TYPE_FLOAT (for example, M_AREA+M_TYPE_LONG).

The **TargetArrayPtr** parameter specifies the address of the array in which to write results. Each blob creates a separate result entry. Only results for blobs that are currently included are obtained. The size of the target array must be large enough to hold the number of currently included blobs. This number can be obtained, using **MblobGetNumber()**.

The target array must be of the correct data type ("double" by default, or whatever type you specified). Results that represent units of measure are expressed in pixels or degrees. If the pixel aspect ratio is not equal to 1, results that represent a position or length are expressed in units of "pixel height".

See also `MblobAllocResult()`, `MblobGetNumber()`, `MblobCalculate()`, `MblobSelect()`, `MblobSelectFeature()`, `MblobSelectFeret()`, `MblobSelectMoment()`

MblobGetResultSingle

Synopsis Read the feature value of a single blob.

Format `void MblobGetResultSingle(BlobResId, LabelVal, Feature, TargetVarPtr)`

MIL_ID BlobResId;	Blob analysis result buffer identifier
long LabelVal;	Label value of the blob
long Feature;	Type of feature for which to get a result
void *TargetVarPtr;	Variable in which to return result

Description This function obtains the result for a specified feature, for a specific blob, from the blob analysis result buffer. The blob for which to obtain the result is determined by its label value.

The **BlobResId** parameter specifies the identifier of the blob analysis result buffer from which to get the result.

The **LabelVal** parameter specifies the label value of the blob for which to get the result. The label value can be obtained, using **MblobGetLabel()** or **MblobGetResult()**. Note, you cannot obtain results for blobs that have been deleted from the result buffer, using **MblobSelect()**.

The **Feature** parameter specifies the feature for which results should be retrieved. The specified feature must have already been calculated with **MblobCalculate()**. See **MblobSelectFeature()**, **MblobSelectFeret()**, and **MblobSelectMoment()** for pre-defined values.

Some features may have been calculated with two results (grayscale and binary). Specifying the feature alone reads the grayscale result. Specifying the feature + M_BINARY reads the binary result. Unless a feature actually has two results, the result (whether grayscale or binary) is read simply by specifying the feature alone.

Results are normally returned in the target array as type "double". If you want results to be returned as a different type, combine the specified feature with M_TYPE_CHAR, M_TYPE_SHORT, M_TYPE_LONG, or M_TYPE_FLOAT (for example, M_AREA+M_TYPE_LONG).

The **TargetVarPtr** parameter specifies the address of the variable in which to write the result retrieved from the blob analysis result buffer.

The target variable must be of the correct data type ("double" by default, or whatever type you specified). Results that represent units of measure are expressed in pixels or degrees. If the pixel aspect ratio is not equal to 1, results that represent a position or length are expressed in units of "pixel height".

See also **MblobAllocResult()**, **MblobGetLabel()**, **MblobGetResult()**, **MblobCalculate()**, **MblobSelect()**, **MblobSelectFeature()**, **MblobSelectFeret()**, **MblobSelectMoment()**

MblobGetRuns

Synopsis Get the blob run-length encoding information.

Format `void MblobGetRuns(BlobResId, LabelVal, ArrayType, RunXPtr, RunYPtr, RunLengthPtr)`

MIL_ID BlobResId;	Blob analysis result buffer identifier
long LabelVal;	Label value of the blob
long ArrayType;	Type of the arrays in which results will be returned
void *RunXPtr;	Array in which to return the X-coordinate of each run
void *RunYPtr;	Array in which to return the Y-coordinate of each run
void *RunLengthPtr;	Array in which to return the length of each run

Description This function obtains the coordinate and length of each run (unbroken horizontal sequence of foreground pixels) in a specified blob from the blob analysis result buffer. Prior to using this function, M_NUMBER_OF_RUNS must have been added to the feature list (**MblobSelectFeature0**) and a call to **MblobCalculate0** must have been made.

The **BlobResId** parameter specifies the identifier of the blob analysis result buffer.

The **LabelVal** parameter specifies the label value of the blob for which to get run information. The label value for a blob can be obtained, using **MblobGetLabel0** or **MblobGetResult0**. You cannot obtain run-encoding information for blobs that have been deleted from the result buffer.

The **ArrayType** parameter specifies the type of the arrays in which the coordinate and length of the runs for a blob will be returned. The following array types can be used; M_TYPE_CHAR, M_TYPE_SHORT, or M_TYPE_LONG.

The **RunXPtr** parameter specifies the address of the array in which to write the X-coordinate of the start (left-most pixel) of each run in the specified blob.

The **RunYPtr** parameter specifies the address of the array in which to write the Y-coordinate of the start of each run in the specified blob.

The **RunLengthPtr** parameter specifies the address of the array in which to write the length of each run in the specified blob.

Note, if either the **RunXPtr**, **RunYPtr**, or **RunLengthPtr** parameter is set to `M_NULL`, no data will be written in that particular array.

The coordinate and length buffers must be large enough to hold information for all runs in the specified blob. The number of runs for a blob can be obtained, using **MblobGetResult()** or **MblobGetResultSingle()**. The number of runs as well as the run-length encoding results are given in raw pixel values and are not affected by the pixel aspect ratio.

See also **MblobSelectFeature()**, **MblobCalculate()**, **MblobGetLabel()**, **MblobGetResult()**, **MblobGetResultSingle()**

MblobInquire

Synopsis Inquire about a blob analysis processing mode.

Format **void MblobInquire(BlobResId, InquireType, UserVarPtr)**

MIL_ID BlobResId;	Blob analysis result buffer identifier
long InquireType;	Parameter about which to inquire
void *UserVarPtr;	Storage location for inquiry result

Description This function performs an inquiry on the specified processing mode associated with the blob analysis result buffer.

The **BlobResId** parameter specifies the identifier of the blob analysis result buffer.

The **InquireType** parameter specifies the processing mode about which to inquire. Refer to **MblobControl()** for a list of processing modes about which you can inquire. This parameter can also be set to M_OWNER_SYSTEM to inquire the identifier of the system on which the result buffer is allocated, or to M_MAX_LABEL to inquire about the maximum label value given to any blob in the result buffer. Label values for blobs are generated when a call to **MblobCalculate()** is made. The maximum label value is not necessarily the same as the total number of blobs, because some label values may not be used, depending on the blob's shape. One of the uses for obtaining the maximum label value is to determine if an 8 or 16-bit image buffer is needed for **MblobLabel()**.

The **UserVarPtr** parameter specifies the address of the variable in which the inquiry result will be written. By default, the result is written as type "double". If you want results to be written as type "long" or "double", combine the specified processing mode with M_TYPE_LONG or M_TYPE_DOUBLE, respectively (for example, M_BLOB_IDENTIFICATION+M_TYPE_LONG).

You can use **MblobControl()** to change a processing mode associated with a result buffer.

See also **MblobControl()**, **MblobCalculate()**, **MblobLabel()**

MblobLabel

Synopsis Draw a labeled image.

Format **void MblobLabel(BlobResId, DestImageBufId, Mode)**

MIL_ID BlobResId;	Blob analysis result buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long Mode;	Write mode

Description This function draws a labeled image in which each blob existing in the specified result buffer is represented with its own unique label value.

The label values are taken from the blob analysis result buffer. A call to **MblobCalculate()** must have been made to generate label values for an image. Blobs that have been deleted from the result buffer are not drawn.

The **BlobResId** parameter specifies the identifier of the blob analysis result buffer.

The **DestImageBufId** parameter specifies the identifier of the destination (labeled) image buffer. This must be a single band, 8 or 16-bit unsigned buffer.

Note, this buffer need not be the same size as the original identifier image used to calculate the blobs, but must be 16 bits deep if the maximum label value exceeds 255. To determine if a 16-bit buffer is necessary, perform an M_MAX_LABEL inquiry, using **MblobInquire()**. The number of blobs alone does not tell you the maximum label value (label values are not necessarily contiguous).

The **Mode** parameter specifies whether or not to clear the destination image buffer before drawing the labeled image into it. This parameter can be set to one of the following:

M_CLEAR	Clear the destination image buffer before placing the labeled image into it.
M_NO_CLEAR	Do not clear the destination image buffer before placing the labeled image into it (background pixels will be unchanged).

See also **MblobCalculate(), MblobFill()**

MblobReconstruct

Synopsis Reconstruct blobs (or blob holes) in an image buffer.

Format `void MblobReconstruct(SrcImageBufId, SeedImageBufId, DestImageBufId, Operation, ProcMode)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID SeedImageBufId;	Seed image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long Operation;	Type of operation to perform
long ProcMode;	Processing mode

Description This function copies (or reconstructs) blobs or blob holes from the source to the destination buffer, according to the specified operation and processing mode. By default, all non-zero pixels in the source buffer are considered to be part of a blob. Use the M_FOREGROUND_ZERO processing mode to inverse this behaviour.

The **SrcImageBufId** parameter specifies the identifier of the source image buffer.

The **SeedImageBufId** parameter specifies the identifier of the image buffer to use as a seed image. A seed image is needed to perform an M_RECONSTRUCT_FROM_SEED type of operation. For any other operation type, set this parameter to M_NULL.

The **DestImageBufId** specifies the identifier of the destination (processed blobs) image buffer.

The source, destination, and seed images must be single band, packed binary, 8, or 16-bit unsigned buffers.

The **Operation** parameter specifies the type of operation to perform. This parameter can be set to one of the following values:

Operation	Description
M_RECONSTRUCT_FROM_SEED	<p>All blobs in the source buffer that have at least one corresponding foreground seed pixel in the seed buffer are copied to the destination buffer, according to the selected processing mode. Blobs that are not seeded are replaced according to the selected processing mode.</p> <div><div><div>Binary</div><div><div>Source image</div><div>Seed image</div><div>Destination image</div></div><div><div>Source image</div><div>Seed image</div><div>Destination image</div></div></div><div><div>Grayscale</div><div><div>Source image</div><div>Seed image</div><div>Destination image</div></div><div><div>Source image</div><div>Seed image</div><div>Destination image</div></div></div><div><div>with M_FOREGROUND_ZERO</div><div><div>Source image</div><div>Seed image</div><div>Destination image</div></div><div><div>Source image</div><div>Seed image</div><div>Destination image</div></div></div></div> <p>Note that when the M_GRAYSCALE and M_FOREGROUND_ZERO processing modes are selected, blobs in the source image that are not seeded are filled with the average grayscale value of the background.</p>
M_ERASE_BORDER_BLOBS	<p>Blobs that touch the border are erased. Pixels of the erased blobs are replaced according to the selected processing mode. All blobs that do not touch the borders of the source image are copied to the destination image buffer according to the selected processing mode.</p> <div><div><div>Binary</div><div><div>Source image</div><div>Destination image</div></div><div><div>Source image</div><div>Destination image</div></div></div><div><div>Grayscale</div><div><div>Source image</div><div>Destination image</div></div><div><div>Source image</div><div>Destination image</div></div></div><div><div>with M_FOREGROUND_ZERO</div><div><div>Source image</div><div>Destination image</div></div><div><div>Source image</div><div>Destination image</div></div></div></div> <p>Note that when the M_GRAYSCALE and M_FOREGROUND_ZERO processing modes are selected, the border blobs are filled with the average grayscale pixel value of the background. This operation is similar to a "border kill".</p>

Operation	Description
M_FILL_HOLES	<p>All blobs in the source buffer are copied to the destination buffer according to the selected processing mode, and those blobs with holes are filled according to the processing mode. A hole must not touch the border of the image in order to be considered a hole.</p> <div><div><div>Binary</div><div><div>Source image</div><div>Destination image</div></div><div><div>Source image</div><div>Destination image</div></div></div><div><div>Grayscale</div><div><div>Source image</div><div>Destination image</div></div><div><div>Source image</div><div>Destination image</div></div></div><div>with M_FOREGROUND_ZERO</div></div> <p>Note that when the M_GRAYSCALE processing mode is selected, holes are filled with the average grayscale pixel value of the corresponding blob.</p>
M_EXTRACT_HOLES	<p>All holes within the blobs of the source buffer are copied to the destination buffer with their pixel values set to their corresponding blob's pixel values, according to the selected processing mode. A hole cannot touch any image border in order to be considered a hole.</p> <div><div><div>Binary</div><div><div>Source image</div><div>Destination image</div></div><div><div>Source image</div><div>Destination image</div></div></div><div><div>Grayscale</div><div><div>Source image</div><div>Destination image</div></div><div><div>Source image</div><div>Destination image</div></div></div><div>with M_FOREGROUND_ZERO</div></div> <p>Note that, in the M_GRAYSCALE processing mode, the pixel values of holes copied to the destination buffer are set to the blob's average grayscale pixel value in the source buffer. Also, when the M_GRAYSCALE and M_FOREGROUND_ZERO processing modes are selected, the blobs are filled with the average grayscale pixel value of the background.</p>

The **ProcMode** parameter specifies the processing mode to use. It can be set to any combination of the following four sets of flags. It can also be set to `M_DEFAULT`, in which case the default flag from each set is used. If no flag from a set is specified, its default flag is used.

M_BINARY (default) or M_GRAYSCALE	For M_BINARY, non-zero pixel values copied to the destination buffer will be set to the maximum value of that buffer (for example, 0xff for an 8-bit buffer). For M_GRAYSCALE, the pixel values copied to the destination buffer will be set to the corresponding pixel value in the source buffer.
M_8_CONNECTED (default) or M_4_CONNECTED	For M_8_CONNECTED, the blobs are computed on an eight connected lattice. For M_4_CONNECTED, the blobs are computed on a four connected lattice.
M_FOREGROUND_ZERO (by default not selected)	The pixel values of blobs will consist of zero values and the pixels of the background will consist of non-zero values; that is, the inverse of the usual blob pixel value definition.
M_SEED_PIXELS_ALL_IN_BLOBS (by default not selected)	Use this flag to optimize the reconstruction process only if all seed pixels have corresponding blob pixels in the source image. This condition often exists when the seed image is an eroded (see MimErode()) version of the source image.

- Note**
- Note, in general, the M_BINARY processing mode is faster.
 - This function is optimized for packed binary buffers.

See also **MimErode()**

MblobSelect

Synopsis Select blobs for calculations and result retrieval.

Format `void MblobSelect(BlobResId, Operation, Feature, Condition, CondLow, CondHigh)`

MIL_ID BlobResId;	Blob analysis result buffer identifier
long Operation;	Operation to perform on specified blobs
long Feature;	Feature to be used for selection
long Condition;	Conditional operator for selection
double CondLow;	Low compare value for the condition
double CondHigh;	High compare value for the condition

Description This function selects blobs that meet a specified criterion. These blobs will be included in or excluded from future operations (calculations or result retrieval), or deleted entirely from the result buffer.

If this function is not called at least once, all blobs are included by default. If there is more than one call to this function, the effect of the calls is cumulative unless `M_INCLUDE_ONLY` or `M_EXCLUDE_ONLY` is specified as the operation to perform.

Once a blob has been excluded, it can normally be re-included only by specifying `M_INCLUDE` or `M_INCLUDE_ONLY` in a future call to this function (with the correct criterion). However, if you change the processing mode of a result buffer (with **MblobControl()**), or use the result buffer with different images (in a call to **MblobCalculate()**), all results in the buffer are discarded and all blobs are re-included.

The **BlobResId** parameter specifies the identifier of the blob analysis result buffer to be used in the blob selection process.

The **Operation** parameter specifies the operation to perform on the specified blobs as follows:

Operation	Description
<code>M_INCLUDE</code>	Include all blobs that meet the specified condition.
<code>M_EXCLUDE</code>	Exclude all blobs that meet the specified condition.
<code>M_INCLUDE_ONLY</code>	Include only those blobs that meet the specified condition (and exclude all others).

Operation	Description
M_EXCLUDE_ONLY	Exclude only those blobs that meet the specified condition (and include all others).
M_DELETE	Delete included blobs that meet the specified condition.

Note, M_INCLUDE affects only the status of currently excluded blobs and M_EXCLUDE affects only currently included blobs. M_DELETE removes blobs permanently from the result buffer and, consequently, prevents these blobs from being re-included.

Including only (M_INCLUDE_ONLY) or excluding only (M_EXCLUDE_ONLY) those blobs that meet the specified condition does not take into consideration the present status of blobs (whether they are included or excluded), except for blobs that have been deleted (M_DELETE).

The **Feature** parameter specifies the feature to use as part of the selection criterion. See **MblobSelectFeature()**, **MblobSelectFerret()**, and **MblobSelectMoment()** for a list of features. The specified result buffer must already contain the results for the specified feature.

Some features may have been calculated with two results (grayscale and binary). Specifying the feature alone uses the grayscale result. Specifying the feature + M_BINARY uses the binary result. Unless a feature actually has two results, the result (whether grayscale or binary) is used simply by specifying the feature alone.

You can also use the **Feature** parameter to specify a group of blobs (M_ALL_BLOBS, M_INCLUDED_BLOBS or M_EXCLUDED_BLOBS) to include, exclude, or delete from future calculations and result retrieval. For example, you can delete all currently excluded blobs from the list of blobs to be operated on.

The **Condition** parameter specifies the condition for the feature selection. This parameter can be set to one of two types of condition:

- Conditions that use two limits (**CondLow** and **CondHigh**):
M_OUT_RANGE, M_IN_RANGE
- Conditions that use one limit (**CondLow**):
M_EQUAL, M_NOT_EQUAL, M_GREATER, M_LESS,
M_GREATER_OR_EQUAL, M_LESS_OR_EQUAL

When `M_OUT_RANGE` is selected, blobs with values for the specified feature less than **CondLow**, or greater than **CondHigh**, are included, excluded, or deleted from future operations on the specified result buffer.

When `M_IN_RANGE` is selected, blobs with values for the specified feature in the range **CondLow** to **CondHigh**, inclusive, are included, excluded, or deleted from future operations on the specified result buffer.

Any of the other conditions use the **CondLow** parameter to include, exclude, or delete blobs from future operations on the specified result buffer.

The **CondLow** and **CondHigh** parameters specify the upper and lower limits of the selected condition. If the condition uses only one limit, set the **CondLow** parameter to the required limit and set the **CondHigh** parameter to `M_NULL`.

Example mblob.c

See also `MblobSelectFeature()`, `MblobSelectMoment()`, `MblobSelectFeret()`, `MblobCalculate()`, `MblobGetNumber()`, `MblobGetResultSingle()`

MblobSelectFeature

Synopsis Select feature(s) to be calculated.

Format **void MblobSelectFeature(FeatureListId, Feature)**

MIL_ID FeatureListId;	Feature list identifier
long Feature;	Feature to be selected

Description This function selects the feature(s) to be calculated by **MblobCalculate()** when using the specified feature list.

Calculations for binary features are performed using the blob identifier image. Grayscale feature calculations are performed using both the blob identifier image and grayscale image. Features that have both binary and grayscale definitions are calculated using both the blob identifier image and the grayscale image (see **MblobCalculate()**).

In general, when the blob identifier image is calibrated, features are calculated in calibrated units; otherwise they are calculated in pixel units. When calculated in pixel units, the pixel aspect ratio, specified with **MblobControl()**, is taken into account horizontally. Results are returned in units of "pixel height" since the pixel width is adjusted to be equal to the pixel height. Note, all positional results are relative to the top-left pixel in the target image or the origin of the coordinate system of the calibrated image.

Also, note that you can change the number of Feret angles, for those features requiring them in calculations, using **MblobControl()**.

The **FeatureListId** parameter specifies the identifier of the feature list buffer.

The **Feature** parameter specifies the feature to add to the feature list. To add several features, you must call this function for each feature you want to add to the list (certain commonly used groups of features can be selected in a single call).

In the table that follows, features which are calculated only in pixel units are marked with the letter **p**, and those which can be calculated in either pixel or calibrated units are marked with the letters **p/c**.

Three features can be assigned as sorting keys. To specify a feature as a sorting key, you add either M_SORT#_UP or M_SORT#_DOWN to the selected feature, where the numbers 1, 2, or 3 replace the # to indicate the sorting precedence.

Note that for features that have both grayscale and binary definitions, the grayscale image is used as the sorting key by default. To override this default, you must specify the M_BINARY flag for the feature selected as a sorting key.

The following group of features do not use grayscale pixel values; they are calculated using only the blob identifier image:

Feature	Description	Units
M_LABEL_VALUE	This is the label value for each blob in an image. This is a positive integer (≥ 1) that is unique for each blob. This feature is always calculated; you do not need to select it.	p
M_AREA	This is the number of foreground pixels in a blob (holes are not counted).	p/c
M_NUMBER_OF_CHAINED_PIXELS	This is the number of chained pixels for all blobs or a specified blob.	p
M_CHAIN_INDEX	These are the indices which differentiate each chain's pixels within a specified blob. The blob's bordering chain is identified as index 1. Chained pixels that delimit holes in blobs are identified by subsequent indexes for each chain.	p
M_CHAIN_Y, M_CHAIN_X	These are the x and y coordinates of each chained pixel in the specified blob, for all chains contained within the blob. 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.	p
M_PERIMETER	This is the total length of edges in a blob (including the edges of any holes), with an allowance made for the staircase effect that is produced when diagonal edges are digitized (inside corners are counted as 1.414, rather than 2.0). A single pixel blob (area = 1) has a perimeter of 4.0.	p/c

Feature	Description	Units
M_BOX_X_MIN, M_BOX_Y_MIN, M_BOX_X_MAX, M_BOX_Y_MAX	These are the coordinates of the extreme left, top, right, and bottom pixels, respectively, of a blob.	p/c
M_FIRST_POINT_X, M_FIRST_POINT_Y	Together, these define a unique point for each object, that is always on the perimeter of the object. The Y coordinate is that of the topmost line of the object, and the X coordinate is that of the left-most pixel on that line.	p/c
M_FERET_X, M_FERET_Y	These are the dimensions of the minimum bounding box of a blob in the horizontal and vertical directions (respectively); that is, $M_BOX_X_MAX - M_BOX_X_MIN + 1$, and similarly for the Y direction.	p/c
M_FERET_MIN_DIAMETER	This is the smallest Feret diameter found after checking a certain number of angles. More angles will give a more accurate result, but will take longer to calculate. Even with the maximum number of angles (M_MAX_FERETS), this feature will not be very accurate for long thin blobs. However, you can get an accurate measure of the breadth of long thin blobs more quickly by using M_BREADTH.	p/c
M_FERET_MIN_ANGLE	This is the angle at which the minimum Feret diameter is found. The value is in degrees, with positive values indicating a counter-clockwise displacement from the positive X-axis.	p/c
M_FERET_MAX_DIAMETER	This is the largest Feret diameter found after checking a certain number of angles. More angles will give a more accurate result, but will take longer to calculate. However, the maximum Feret diameter is not very sensitive to the number of angles, and 8 usually gives an accurate result.	p/c
M_FERET_MAX_ANGLE	This is the angle at which the maximum Feret diameter is found. The value is in degrees, with positive values indicating a counter-clockwise displacement from the positive X-axis.	p/c
M_FERET_MEAN_DIAMETER	This is the average Feret diameter at all the angles checked.	p/c
M_FERET_ELONGATION	This is a measure of the shape of a blob. It is equal to $\frac{M_FERET_MAX_DIAMETER}{M_FERET_MIN_DIAMETER}$ <p>It is accurate for reasonably compact objects, but becomes less accurate for very elongated objects (because M_FERET_MIN_DIAMETER becomes less accurate). For very elongated objects, you should probably use M_ELONGATION.</p>	p

Feature	Description	Units
M_CONVEX_PERIMETER	This is an approximation of the perimeter of the convex hull of a blob. It is derived from several Feret diameters; so, a larger number of Ferets gives a more accurate result.	p/c
M_X_MIN_AT_Y_MIN, M_X_MAX_AT_Y_MAX, M_Y_MIN_AT_X_MAX , M_Y_MAX_AT_X_MIN	These values, together with the four box coordinates, give four contact points on the convex perimeter of the object.	p/c
M_COMPACTNESS	This value is a minimum for a circle (1.0) and is derived from the perimeter (p) and area (A). The more convoluted the shape, the greater the value. It is equal to $\frac{p^2}{(4\pi A)}$	p
M_NUMBER_OF_HOLES	This value is equal to the number of holes in a blob. Holes that intersect the edge of the image are not counted (they may not be holes). This value is equal to 1 - M_EULER_NUMBER and is therefore a true hole count only in M_INDIVIDUAL processing mode.	p
M_NUMBER_OF_RUNS	This value is equal to the total number of runs in a blob. A run is defined as a horizontal string of consecutive foreground pixels.	p
M_ROUGHNESS	This is a measure of how rough a blob is and is equal to M_PERIMETER / M_CONVEX_PERIMETER. A smooth convex object will have the minimum roughness of 1.0	p
M_EULER_NUMBER	This is: the number of blobs - number of holes. This value is more useful for M_WHOLE_IMAGE than for M_INDIVIDUAL processing mode.	p
M_LENGTH	This is a measure of the true length of an object, although it can only be applied to certain object types because it is derived from the perimeter (P) and area (A) assuming that P = 2(length + breadth) and A = length x breadth. It complements M_FERET_MAX_DIAMETER because it is accurate for different blob types (for example, long thin ones). Note, it is calculated much faster than the maximum Feret diameter.	p/c
M_BREADTH	This is a measure of the true breadth of an object, with the same advantages and disadvantages as M_LENGTH.	p/c

Feature	Description	Units
M_ELONGATION	This value is equal to M_LENGTH / M_BREADTH It is similar to M_FERET_ELONGATION, except that it should be used for long thin objects.	p
M_INTERCEPT_0	This is the number of times a transition from background to foreground (not vice versa) occurs in the horizontal direction for the entire blob. In other words, it is equal to the number of times the neighborhood configuration [B, F] occurs in a blob, where B is a background pixel and F is a foreground pixel.	p
M_INTERCEPT_45	This is the number of times that the neighborhood configuration $\begin{bmatrix} \cdot & F \\ B & \cdot \end{bmatrix}$ occurs in a blob, where F is a foreground pixel, B is a background pixel and a dot can be any pixel value.	p
M_INTERCEPT_90	This is the number of times that the neighborhood configuration $\begin{bmatrix} F \\ B \end{bmatrix}$ occurs in a blob.	p
M_INTERCEPT_135	This is the number of times that the neighborhood configuration $\begin{bmatrix} F & \cdot \\ \cdot & B \end{bmatrix}$ occurs in a blob.	p
The following features require grayscale pixel values, and can only be calculated if you provide a grayscale image:		
M_SUM_PIXEL	This is the sum of all pixel values in a blob.	p
M_MIN_PIXEL	This is the minimum pixel value found in a blob.	p
M_MAX_PIXEL	This is the maximum pixel value found in a blob.	p
M_MEAN_PIXEL	This is the mean pixel value in a blob. It is equal to M_SUM_PIXEL / M_AREA.	p
M_SIGMA_PIXEL	This is the standard deviation of pixel values in a blob. It is equal to $\sqrt{\frac{\sum p_i^2 - (\sum p_i)^2 / N}{N}}$ where N = number of pixels and p = pixel value.	p
M_SUM_PIXEL_SQUARED	This is the sum of the squares of each pixel value in a blob.	p

Feature	Description	Units
<p>The following features have two different definitions: a binary definition, where all pixels are considered equal, and a grayscale, where pixels are weighted by their value in the gray image (the grayscale version is much slower to calculate).</p> <p>If you don't provide a grayscale image, only the binary version can be calculated. If you do provide a grayscale image, both versions are calculated. However, if you want a feature to be calculated in one version only, you can combine the selected feature with M_BINARY or M_GRAYSCALE (for example, M_CENTER_OF_GRAVITY_X + M_BINARY).</p>		
M_CENTER_OF_GRAVITY_X	<p>This is the X position of the center of gravity of a blob. The grayscale version is $M_MOMENT_X1_Y0 / M_SUM_PIXEL$. The binary version uses M_AREA instead of M_SUM_PIXEL.</p>	p/c
M_CENTER_OF_GRAVITY_Y	<p>This is the Y position of the center of gravity of a blob. The grayscale version is $M_MOMENT_X0_Y1 / M_SUM_PIXEL$. The binary version uses M_AREA instead of M_SUM_PIXEL.</p>	p/c
M_MOMENT_X0_Y1, M_MOMENT_X1_Y0, M_MOMENT_X1_Y1, M_MOMENT_X0_Y2, M_MOMENT_X2_Y0, M_MOMENT_CENTRAL_X0_Y2, M_MOMENT_CENTRAL_X2_Y0, M_MOMENT_CENTRAL_X1_Y1	<p>Moments have the syntax M_MOMENT_Xn_Ym and are defined as</p> $\sum_i^{\infty} x_i^n y_i^m p_i$ <p>where p_i = value of a pixel (always 1 for binary moments), x_i = its X coordinate and y_i = its Y coordinate. For central moments, coordinates are relative to each blob's center of gravity. Ordinary moments use coordinates relative to the image origin (top-left corner). Calculate higher moments by calling MblobSelectMoment().</p>	p
M_AXIS_PRINCIPAL_ANGLE	<p>This is the angle at which a blob has the least moment of inertia. For elongated blobs, it is aligned with the longest axis. The result is always between -90° and +90°, measured in a counter-clockwise direction from the positive X-axis. It is calculated as:</p> $-0.5 * \text{atan} \frac{(2 * M_MOMENT_CENTRAL_X1_Y1)}{M_MOMENT_CENTRAL_X2_Y0 - M_MOMENT_CENTRAL_X0_Y2}$ <p>When the blob identifier image is calibrated this feature is calculated in calibrated units; otherwise they are calculated in pixel units</p>	p/c

Feature	Description	Units
M_AXIS_SECONDARY_ANGLE	This is the angle perpendicular to M_AXIS_PRINCIPAL_ANGLE. It is always between -90° and +90°.	p/c
The following predefined values allow you to select groups of features in a single call.		
M_BOX	Adds all 4 box features plus X and Y Ferets.	p
M_CONTACT_POINTS	Adds first point and other contact features (M_X_MIN_AT_Y_MIN, M_X_MAX_AT_Y_MAX, M_Y_MIN_AT_X_MAX and M_Y_MAX_AT_X_MIN).	p
M_CENTER_OF_GRAVITY	Adds both X and Y coordinates of the center of gravity.	p
M_ALL_FEATURES	Adds all features (except general Feret and general moment).	p
M_NO_FEATURES	Removes all features (except label value).	p
M_CHAINS	Adds all 4 chain features. Note that the chain code is computed depending on the lattice and foreground values set with MblobControl() .	
You can add the following sorting defines to a feature to specify it as a sorting key for the result retrieval		
M_SORT1_UP	When added to a feature, this define specifies the feature as the first sorting key (in ascending order).	
M_SORT1_DOWN	When added to a feature, this define specifies the feature as the first sorting key (in descending).	
M_SORT2_UP	When added to a feature, this define specifies the feature as the second sorting key (in ascending order).	
M_SORT2_DOWN	When added to a feature, this define specifies the feature as the second sorting key (in descending).	
M_SORT3_UP	When added to a feature, this define specifies the feature as the third sorting key (in ascending order).	
M_SORT3_DOWN	When added to a feature, this define specifies the feature as the third sorting key (in descending).	
M_NO_SORT	When added to a feature, this define removes the specified sorting key.	
Note that only one feature can be selected as the first, second, or third sorting key.		

Example mblob.c

See also **MblobSelectMoment()**, **MblobSelectFeret()**, **MblobCalculate()**, **MblobControl()**

MblobSelectFeret

Synopsis Add Feret angle to the feature list.

Format void MblobSelectFeret(**FeatureListId**, **Angle**)

MIL_ID FeatureListId;	Feature list identifier
double Angle;	Angle at which to calculate Feret

Description This function adds M_GENERAL_FERET at the specified angle to the feature list. The Feret diameter is then calculated at this angle, using **MblobCalculate()**. Results for this calculation can be obtained with **MblobGetResult()** or **MblobGetResultSingle()**, specifying M_GENERAL_FERET as the feature.

The **FeatureListId** parameter specifies the identifier of the feature list buffer.

The **Angle** parameter specifies the angle, in degrees, to be used for calculating the general Feret. It overrides any previous angle specified for the general Feret in this feature list. Note, the Feret diameters at 0° and 90° are calculated more efficiently with **MblobSelectFeature()** (the features are called M_FERET_X and M_FERET_Y respectively).

To select the general Feret as a sorting key for the result retrieval, add M_SORT#UP or M_SORT#DOWN to the **Angle** parameter. The numbers 1, 2, or 3 can be assigned to the number sign to indicate the sorting precedence of the feature.

See also MblobSelectFeature(), MblobCalculate(), MblobGetResult(), MblobGetResultSingle()

MblobSelectMoment

Synopsis Add specified moment to the feature list.

Format `void MblobSelectMoment(FeatureListId, MomType, XMomOrder, YMomOrder)`

MIL_ID FeatureListId;	Feature list identifier
long MomType;	Moment type
long XMomOrder;	X order of the moment
long YMomOrder;	Y order of the moment

Description Moment calculations other than those supported by **MblobSelectFeature()** must be specified with this function. This function adds the general moment with the specified parameters to the feature list. The general moment will be calculated by **MblobCalculate()**. Results for this calculation can be obtained by using **MblobGetResult()** or **MblobGetResultSingle()**, specifying M_GENERAL_MOMENT as the feature. Moments directly supported by **MblobSelectFeature()** (for example, M_MOMENT_X0_Y2) are calculated faster by selecting them through that function.

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

The **FeatureListId** parameter specifies the identifier of the feature list buffer.

The **MomType** parameter specifies the moment type, either M_CENTRAL or M_ORDINARY. If the calculation is done on a binary image, only a binary version of the result is calculated. If you provide a grayscale image, both grayscale and binary versions are calculated. However, if you want only one specific version of the results, you can combine the selected moment with M_BINARY or M_GRAYSCALE.

The **XMomOrder** and **YMomOrder** parameters specify, respectively, the X and Y order of the moment. The X and Y order of the moment must be greater than or equal to 0.

To select the general moment as a sorting key for the result retrieval, add M_SORT#UP or M_SORT#DOWN to the **MomType** parameter. The numbers 1, 2, or 3 can be assigned to the number sign to indicate the sorting precedence of the feature.

See also **MblobSelectFeature()**, **MblobCalculate()**, **MblobGetResultSingle()**, **MblobGetResultSingle()**

MbufAlloc1d

Synopsis Allocate a 1D data buffer.

Format **MIL_ID MbufAlloc1d(SystemId, SizeX, Type, Attribute, BufIdPtr)**

MIL_ID SystemId;	System identifier
long SizeX;	X dimension
long Type;	Data depth and data type
long Attribute;	Buffer attribute
MIL_ID *BufIdPtr;	Storage location for buffer identifier

Description This function allocates a one-dimensional one-band data buffer on the specified system.

After allocating a buffer, we recommend that you check if the operation was successful, using **MappGetError()** or by verifying that the buffer identifier returned is not M_NULL. When a buffer is no longer required, release it, using **MbufFree()**.

The **SystemId** parameter specifies the system on which the buffer will be allocated. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to allocate the buffer (it can be the Host system or any already allocated system).

The **SizeX** parameter specifies the buffer width in the units appropriate for the selected type of buffer attributes. For example, if the buffer has a LUT buffer attribute, specify the number of LUT entries to allocate.

The **Type** parameter specifies a combination of two values: the depth and type of the data. Express the depth in bits and give the data range as one of the following:

Data type	Description	Depth (in bits)
M_SIGNED	Signed data	8, 16, or 32
M_UNSIGNED	Unsigned data (default)	1, 8, 16, or 32
M_FLOAT	Floating point data	32

For example, when allocating a 8-bit unsigned buffer, you would set the **Type** parameter to (8 + M_UNSIGNED).

Note, you cannot allocate a 1-bit (binary) LUT buffer.

The **Attribute** parameter defines the buffer usage. The system uses this information to determine where to allocate the buffer in physical memory. For example, to allocate a LUT buffer, you should set the **Attribute** parameter to M_LUT. Set this parameter to one of the following:

Attribute	Description
M_IMAGE	Image data.
M_LUT	Lookup table.
M_KERNEL	Convolution kernel for convolution functions.
M_STRUCT_ELEMENT	Structuring element for morphology functions.
M_ARRAY	Array of data. Note that some functions take an M_ARRAY buffer rather than a user-defined array.

When allocating an image buffer (M_IMAGE), you must also specify the intended purpose of this buffer by combining M_IMAGE with one or more of the following:

Usage Specifiers	Description
M_DISP	An image buffer that can be displayed.
M_GRAB	An image buffer in which to grab data. This type of buffer is usually allocated in physically contiguous, non-paged memory.
M_PROC	An image buffer that can be processed.
M_COMPRESS	An image buffer that can hold compressed data. Note that a buffer with this attribute cannot have the M_SIGNED data type.

The maximum (total) number of grab (M_GRAB) buffers that can be allocated is restricted by the total amount of DMA memory that was specified at the time of installation.

For systems with on-board processors, the total number of M_GRAB buffers and M_PROC buffers is limited by the amount of on-board memory.

When allocating buffers for operations that require a source and destination buffer, and one of the buffers has an M_COMPRESS attribute, the data will be compressed or decompressed depending on the attributes of the destination buffer. If both the source and destination buffers have M_COMPRESS specifiers but different compression types, the data will be re-compressed according to the settings of the destination buffer.

For an M_COMPRESS type of image buffer, one of the following must be added to indicate the type of compressed data. The image buffer's data format dictates which compression type will be performed. If nothing is added, M_JPEG_LOSSY is assumed.

Compression specifiers:	Description	Supported data formats
M_JPEG_LOSSLESS	The buffer will be used to hold JPEG lossless.	1-band, 8- or 16-bit data.
M_JPEG_LOSSY	The buffer will be used to hold JPEG lossy data.	1-band 8-bit data.
M_JPEG2000_LOSSLESS	The buffer will be used to hold JPEG2000 lossless data.	1-band, 8- or 16-bit data.
M_JPEG2000_LOSSY	The buffer will be used to hold JPEG2000 lossy data.	1-band, 8- or 16-bit data.

MIL automatically selects the most appropriate internal storage format according to the specified intended usage attribute. For general processing, MIL will convert the data when the function requires a different format. If the default internal storage format is not appropriate and you want to avoid conversion during a time critical operation, you can add one of the following:

Board-dependent internal storage format specifiers:	
M_DDRAW	Force the buffer to be a DDraw surface.
M_DIB	Force the buffer to be a DIB buffer.
M_FLIP	Force the buffer to be top down (DIB).

Board-dependent location specifiers:	
M_ON_BOARD	Force the buffer in the on-board memory.
M_OFF_BOARD	Force the buffer in the Host memory.
M_NON_PAGED	Force the buffer in non-pageable memory.

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

The **BufIdPtr** parameter specifies the address of the variable in which the buffer identifier is to be written. Since the **MbufAlloc1d()** function also returns the buffer identifier, you can set this parameter to **M_NULL**. If allocation fails, **M_NULL** is written as the identifier.

Note MIL-Lite does not support JPEG 2000 compression, and requires dedicated hardware for JPEG compression. This is not a restriction under MIL.

Return value The returned value is the buffer identifier. If allocation fails, **M_NULL** is returned.

Status Current limitation:

- For **M_KERNEL** and **M_LUT** data buffers, the data type must be 8, 16, or 32-bit integer or floating point.
- For **M_STRUCT_ELEMENT** data buffers, the data type must be 32-bit integer or floating point.

See also **MbufAlloc2d()**, **MbufAllocColor()**, **MbufFree()**

MbufAlloc2d

Synopsis Allocate a 2D data buffer.

Format **MIL_ID MbufAlloc2d(SystemId, SizeX, SizeY, Type, Attribute, BufIdPtr)**

MIL_ID SystemId;	System identifier
long SizeX;	X dimension
long SizeY;	Y dimension
long Type;	Data depth and data type
long Attribute;	Buffer attributes
MIL_ID *BufIdPtr;	Storage location for buffer identifier

Description This function allocates a two-dimensional one-band data buffer on the specified system.

After allocating a buffer, we recommend that you check if the operation was successful, using **MappGetError()** or by verifying that the buffer identifier returned is not M_NULL. When a buffer is no longer required, release it, using **MbufFree()**.

The **SystemId** parameter specifies the system on which the buffer will be allocated. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to allocate the buffer (it can be the Host system or any already allocated system).

The **SizeX** and **SizeY** parameters specify the buffer width and height, respectively, in the units appropriate for the selected buffer attribute. For example, if the buffer has an image buffer attribute, specify the width and height in pixels.

The **Type** parameter specifies a combination of two values: the depth and type of the data. Express the depth in bits and give the data range as one of the following:

Data type	Description	Depth (in bits)
M_SIGNED	Signed data	8, 16, or 32
M_UNSIGNED	Unsigned data (default)	1, 8, 16, or 32
M_FLOAT	Floating point data	32

For example, when allocating a 8-bit unsigned buffer, you would set the **Type** parameter to (8 + M_UNSIGNED).

Note, you cannot allocate a 1-bit (binary) LUT buffer.

The **Attribute** parameter defines the buffer usage. The system uses this information to determine where to allocate the buffer in physical memory. This parameter should be set to one of the following:

M_IMAGE	Image data.
M_LUT	Lookup table.
M_KERNEL	Convolution kernel for convolution functions.
M_STRUCT_ELEMENT	Structuring element for morphology functions.
M_ARRAY	Array of data. Note that some functions take an M_ARRAY buffer rather than a user-defined array.

When selecting an M_IMAGE attribute, it should be set to M_IMAGE + *specifier*. For example, to allocate an image buffer that can be processed and displayed, you should set the **Attribute** parameter to M_IMAGE + M_PROC + M_DISP. The specifier can be one or more of the following:

Usage specifiers:	
M_DISP	An image buffer that can be displayed.
M_GRAB	An image buffer in which to grab data. This type of buffer is usually allocated in physically contiguous, non-paged memory.
M_PROC	An image buffer that can be processed.
M_COMPRESS	An image buffer that can hold compressed data. Note that a buffer with this attribute cannot have the M_SIGNED data type.

The maximum (total) number of grab (M_GRAB) buffers that can be allocated is restricted by the total amount of DMA memory that was specified at the time of installation.

For boards with on-board processors, the total number of M_GRAB buffers and M_PROC buffers is limited by the amount of on-board memory.

When allocating buffers for operations that require a source and destination buffer, and one of the buffers has an M_COMPRESS specifier, the data will be compressed or decompressed depending on the attributes of the destination buffer. If both the source and destination buffers have M_COMPRESS specifiers but different compression types, the data will be re-compressed according to the settings of the destination buffer.

For an M_COMPRESS type of image buffer, one of the following must be added to indicate the type of compressed data. The image buffer's data format dictates which compression type will be performed. If nothing is added, M_JPEG_LOSSY is assumed.

Compression specifiers:	Description	Supported data formats
M_JPEG_LOSSLESS	The buffer will be used to hold JPEG lossless data.	1-band, 8- or 16-bit data.
M_JPEG_LOSSLESS_INTERLACED	The buffer will be used to hold JPEG lossless data in separate fields.	1-band, 8- or 16-bit data.
M_JPEG_LOSSY	The buffer will be used to hold JPEG lossy data.	1-band 8-bit data.
M_JPEG_LOSSY_INTERLACED	The buffer will be used to hold JPEG lossy data in separate fields.	1-band 8-bit data.
M_JPEG2000_LOSSLESS	The buffer will be used to hold JPEG2000 lossless data.	1-band, 8- or 16-bit data.
M_JPEG2000_LOSSY	The buffer will be used to hold JPEG2000 lossy data.	1-band, 8- or 16-bit data.

MIL automatically selects the most appropriate internal storage format according to the specified intended usage attribute. For general processing, MIL will convert the data when the function requires a different format. If the default internal storage format is not appropriate and you want to avoid conversion during a time critical operation, you can add one of the following:

Board-dependent internal storage format specifiers:	
M_DDRAW	Force the buffer to be a DDraw surface.
M_DIB	Force the buffer to be a DIB buffer.
M_FLIP	Force the buffer to be top down (DIB).

Board-dependent location specifiers:	
M_ON_BOARD	Force the buffer in the on-board memory.
M_OFF_BOARD	Force the buffer in the Host memory.
M_NON_PAGED	Force the buffer in non-pageable memory.

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

The **BufIdPtr** parameter specifies the address of the variable in which the buffer identifier is to be written. Since the **MbufAlloc2d()** function also returns the buffer identifier, you can set this parameter to M_NULL. If allocation fails, M_NULL is written as the identifier.

Note MIL-Lite does not support JPEG 2000 compression, and requires dedicated hardware for JPEG compression. This is not a restriction under MIL.

Return value The returned value is the buffer identifier. If allocation fails, M_NULL is returned.

Status Current limitation:

- For M_KERNEL and M_LUT data buffers, the data type must be 8, 16, or 32-bit integer or floating point.
- For M_STRUCT_ELEMENT data buffers, the data type must be 32-bit integer or floating point.

See also **MbufAlloc1d()**, **MbufAllocColor()**, **MbufFree()**

MbufAllocColor

Synopsis Allocate a color data buffer.

Format **MIL_ID MbufAllocColor(SystemId, SizeBand, SizeX, SizeY, Type, Attribute, BufIdPtr)**

MIL_ID SystemId;	System identifier
long SizeBand;	Number of color bands
long SizeX;	X dimension
long SizeY;	Y dimension
long Type;	Data type and data depth per band
long Attribute;	Buffer attributes
MIL_ID *BufIdPtr;	Storage location for buffer identifier

Description This function allocates a data buffer with multiple color bands on the specified system. This type of buffer allows the representation of color images (for example, RGB).

This function creates buffers that have a two-dimensional surface for each specified color band. You can use **MbufAlloc1d()** and **MbufAlloc2d()** to create single band one- or two-dimensional data buffers, respectively.

After allocating a buffer, we recommend that you check if the operation was successful, using **MappGetError()**, or by verifying that the buffer identifier returned is not M_NULL.

When a buffer is no longer required, release it, using **MbufFree()**.

The **SystemId** parameter specifies the system on which the buffer will be allocated. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to allocate the buffer (it can be the Host system or any already allocated system).

The **SizeBand** parameter specifies the number of (x,y) surfaces (also called color bands) to allocate to the buffer. Specify one band for each color component the buffer will need to store for the image. Monochrome images require one band; RGB color images require three color bands. This parameter can be set to any non-zero integer value. However, in general, only 1- and 3-band buffers are allowed.

The **SizeX** and **SizeY** parameters specify the buffer width and height, respectively, in the units appropriate for the selected buffer attribute. For example, if the buffer has an image buffer attribute, width and height are specified in pixels.

The **Type** parameter specifies a combination of two values: data type and data depth per band. Express the depth in bits and give the data type as one of the following:

Data type	Description	Depth/band (in bits)
M_SIGNED	Signed data	8, 16, or 32
M_UNSIGNED	Unsigned data (default)	1, 8, 16, or 32
M_FLOAT	Floating point data	32

For example, when allocating an 8-bit unsigned buffer, you would set the **Type** parameter to (8 + M_UNSIGNED).

Note, you cannot allocate a 1-bit (binary) LUT buffer.

The **Attribute** parameter defines the buffer usage. The system uses this information to determine where to allocate the buffer in physical memory. This parameter should be set to M_LUT, or to M_IMAGE + *specifier*. For example, to allocate an image buffer that can be processed and displayed, you should set the **Attribute** parameter to M_IMAGE + M_PROC + M_DISP. The specifier can be one or more of the following:

Usage specifiers:	
M_DISP	An image buffer that can be displayed.
M_GRAB	An image buffer in which data can be grabbed. This type of buffer is usually allocated in physically contiguous, non-paged memory.
M_PROC	An image buffer that can be processed.
M_COMPRESS	An image buffer that can hold compressed data. Note that a buffer with this attribute cannot have the M_SIGNED data type.

The maximum (total) number of grab (M_GRAB) buffers that can be allocated is restricted by the total amount of DMA memory that was specified at the time of installation.

For boards with on-board processors, the total number of M_GRAB buffers and M_PROC buffers is limited by the amount of on-board memory.

When allocating buffers for operations that require a source and destination buffer, and one of the buffers has an `M_COMPRESS` specifier, the data will be compressed or decompressed depending on the attributes of the destination buffer. If both the source and destination buffers have `M_COMPRESS` specifiers but different compression types, the data will be re-compressed according to the settings of the destination buffer.

For an `M_COMPRESS` type of image buffer, one of the following must be added to indicate the type of compressed data. The image buffer's data format dictates which compression type will be performed. If nothing is added, `M_JPEG_LOSSY` is assumed.

Compression specifiers:	Description	Supported data formats
<code>M_JPEG_LOSSLESS</code>	The buffer will be used to hold JPEG lossless data.	1-band, 8- or 16-bit data, and 3-band data in: <code>M_RGB24</code> or <code>M_RGB48</code> .
<code>M_JPEG_LOSSLESS_INTERLACED</code>	The buffer will be used to hold JPEG lossless data in separate fields.	1-band, 8- or 16-bit data.
<code>M_JPEG_LOSSY</code>	The buffer will be used to hold JPEG lossy data.	1-band 8-bit data, and 3-band 8-bit data in: <code>M_RGB24</code> , <code>M_YUV24</code> , <code>M_YUV12</code> , <code>M_YUV9</code> , <code>M_YUV16</code> + <code>M_PLANAR</code> , or <code>M_YUV16</code> + <code>M_PACKED</code> .
<code>M_JPEG_LOSSY_INTERLACED</code>	The buffer will be used to hold JPEG lossy data in separate fields.	1-band 8-bit data, and 3-band 8-bit data in <code>M_YUV16</code> + <code>M_PACKED</code> .
<code>M_JPEG2000_LOSSLESS</code>	The buffer will be used to hold JPEG2000 lossless data.	1-band, 8- or 16-bit data, and 3-band, 8- or 16-bit data in: <code>M_RGB24+M_PLANAR</code> , <code>M_RGB48+M_PLANAR</code> , <code>M_YUV24+M_PLANAR</code>

Compression specifiers:	Description	Supported data formats
M_JPEG2000_LOSSY	The buffer will be used to hold JPEG2000 lossy data.	1-band, 8- or 16-bit data, and 3-band, 8- or 16-bit data in: M_RGB24, M_RGB48, M_YUV9, M_YUV12, M_YUV16+M_PLANAR, and M_YUV24.

MIL automatically selects the most appropriate internal storage format according to the specified intended usage attribute. For general processing, MIL will convert the data when the function requires a different format. If the default internal storage format is not appropriate and you want to avoid conversion during a time critical operation, you can add one of the following:

Internal storage format specifiers:	
M_DDRAW	Force the buffer to be a DDraw surface.
M_DIB	Force the buffer to be a DIB buffer.
M_FLIP	Force the buffer to be top down (DIB).
M_NO_FLIP	Force the buffer to be top up.

For the following specifiers, the buffer must be an 8-bit multi-band color buffer. See *MIL/MIL-Lite Board-Specific Notes* to verify which formats are supported on your board.

Note that it might be slower to use buffers that have been forced with one of these attributes. Although there is no right or wrong storage format to use, certain operations are optimized for some formats.

Internal storage format specifiers for color buffers:	
M_PACKED	Buffer bands to be packed (color buffer only).
M_PLANAR	Force the buffer bands to be planar (color buffer only).
M_RGB3 + M_PLANAR	3-bit (RGB 1:1:1) planar pixels.
M_RGB15 + M_PACKED	16-bit packed pixels (XRGB 1:5:5:5). Note that when accessing an M_RGB15+M_PACKED buffer as a 3-band 8-bit buffer, the least significant bits are set to 0.
M_RGB16 + M_PACKED	16-bit packed pixels (RGB 5:6:5). Note that when accessing an M_RGB16+M_PACKED buffer as a 3-band 8-bit buffer, the least significant bits are set to 0.

M_BGR24 + M_PACKED	24-bit (BGR) packed pixels.
M_RGB24 + M_PLANAR	24-bit (RGB 8:8:8) planar pixels.
M_BGR32 + M_PACKED	32-bit (BGR) packed pixels.
M_RGB48 + M_PLANAR	48-bit (RGB 16:16:16) planar pixels.
M_RGB96 + M_PLANAR	96-bit (RGB 32:32:32) planar pixels.
M_YUV9 + M_PLANAR	YUV9 planar standard.
M_YUV12 + M_PLANAR	YUV12 planar standard.
M_YUV16 + M_PLANAR	YUV16 planar (4:2:2) standard.
M_YUV16 + M_PACKED	YUV16 packed (4:2:2) standard.
M_YUV16_UYVY + M_PACKED	YUV16 packed (4:2:2) standard.
M_YUV16_YUYV + M_PACKED	YUV16 packed (4:2:2) standard.
M_YUV24 + M_PLANAR	YUV24 planar standard.

Location specifiers:	
M_ON_BOARD	Force the buffer in the on-board video memory.
M_OFF_BOARD	Force the buffer in the Host memory.
M_PAGED	Force the buffer in pageable memory.
M_NON_PAGED	Force the buffer in non-pageable memory.

Note that you can allocate one M_DISP+M_ON_BOARD buffer.

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

The **BuIdPtr** parameter specifies the address of the variable in which the buffer identifier is to be written. Since the **MbufAllocColor()** function also returns the buffer identifier, you can set this parameter to M_NULL. If allocation fails, M_NULL is written as the identifier.

Note MIL-Lite does not support JPEG 2000 compression, and requires dedicated hardware for JPEG compression. This is not a restriction under MIL.

Return value The returned value is the buffer identifier. If allocation fails, M_NULL is returned.

See also **MbufAlloc1d()**, **MbufAlloc2d()**, **MbufFree()**

MbufBayer

Synopsis Decode the color information of a single-band, Bayer color-encoded image.

Format **void MbufBayer(SrcImageBufId, DestImageBufId, WhiteBalanceCoefficientsID, ControlFlag)**

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
MIL_ID WhiteBalanceCoefficientsId;	White balance coefficients buffer ID
long ControlFlag;	Pattern of source data's pixels and whether to calculate white balance coefficients

Description This function converts a single-band, Bayer color-encoded image into a 1- or 3-band image.

This function can also white balance the image, if the appropriate coefficients are provided. White balancing occurs after the image has been converted. Note that you can use **MbufBayer()** to automatically determine the appropriate coefficients by adding **M_WHITE_BALANCE_CALCULATE** to the **ControlFlag** parameter when grabbing a white image.

The **SrcImageBufId** parameter specifies the identifier of the source image buffer. The source buffer must be a single-band 8-bit image buffer, and be at least 2x2 in size. The source buffer is considered unsigned.

The **DestImageBufId** parameter specifies the identifier of the destination image buffer. The destination buffer must have either 1 or 3 bands, with 8 bits per pixel, and be at least 2x2 in size. The destination buffer is considered unsigned.

To avoid internal buffer type conversions, the destination buffer should be either monochrome, or a 3-band color buffer in one of the following formats:

- **M_RGB24+M_PLANAR**
- **M_BGR32+M_PACKED**
- **M_YUV16_YUYV** (equivalent to **M_YUV16+M_PACKED**).

Destination buffers in other formats will be converted automatically to one of the formats specified above, before the Bayer-to-Color conversion is performed. After the conversion, the destination buffer is then converted back to its original format. Note that additional conversion operations will increase the processing time.

The **WhiteBalanceCoefficientsId** parameter specifies the MIL array buffer that contains the white balance coefficients (when white balancing an image), or the buffer into which the coefficients are placed (when calculating white balance coefficients). The coefficients must be in a single-band 32-bit floating-point buffer that has an M_ARRAY attribute and that has the dimensions 3x1. The first, second, and third values in the array are used for the red, green, and blue bands, or Y, U, and V bands, respectively. When this parameter is left to M_DEFAULT, no white balance coefficients are applied.

The coefficients are applied according to the format of the destination buffer:

- If the format of the destination buffer is RGB or a similar variant (such as BGR), the pixels of each band are multiplied by the corresponding value.
- If the format of the destination buffer is YUV, the pixels of the Y band are multiplied by the first value and the pixels of the U and V bands are summed with the second and third values, respectively.
- If the format of the destination buffer is 8-bit monochrome, the pixels of the image are multiplied by the first value in the array; the last two values in the array are ignored.

The results of the white balance correction are saturated, if necessary, according to the bit depth of the destination.

The **ControlFlag** identifies the Bayer pattern to use in the conversion, and specifies whether the white balance coefficients will be calculated. This parameter can be set to one of the following:

M_BAYER_GB	Use the Bayer pattern that has the top-left pixel as the green component and the next pixel as the blue component. That is, pixel [0,0] is green, pixel [1,0] is blue and pixel [0,1] is red in the source image.	<table><tr><td>G</td><td>B</td></tr><tr><td>R</td><td>G</td></tr></table>	G	B	R	G
G	B					
R	G					

M_BAYER_BG	Use the Bayer pattern that has the top-left pixel as the blue component and the next pixel as the green component. That is, pixel [0,0] is blue, pixel [1,0] is green and pixel [1,1] is red in the source image.	<table><tr><td>B</td><td>G</td></tr><tr><td>G</td><td>R</td></tr></table>	B	G	G	R
B	G					
G	R					
M_BAYER_RG	Use the Bayer pattern that has the top-left pixel as the red component and the next pixel as the green component. That is, pixel [0,0] is red, pixel [1,0] is green and pixel [1,1] is blue in the source image.	<table><tr><td>R</td><td>G</td></tr><tr><td>G</td><td>B</td></tr></table>	R	G	G	B
R	G					
G	B					
M_BAYER_GR	Use the Bayer pattern that has the top-left pixel as the green component and the next pixel as the red component. That is, pixel [0,0] is green, pixel [1,0] is red and pixel [0,1] is blue in the source image.	<table><tr><td>G</td><td>R</td></tr><tr><td>B</td><td>G</td></tr></table>	G	R	B	G
G	R					
B	G					

The control M_WHITE_BALANCE_CALCULATE can be added to the **ControlFlag** parameter to automatically determine the coefficients to be used for the white balance operation. Note that to use M_WHITE_BALANCE_CALCULATE, the source image buffer must be entirely white.

❖ If the *MdigControl*(M_GRAB_SCALE) control type has been changed 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.

MbufChildColor

Synopsis Allocate a color-band child data buffer within a color parent buffer.

Format **MIL_ID MbufChildColor(ParentBufId, Band, BufIdPtr)**

MIL_ID ParentBufId;	Parent buffer identifier
long Band;	Index of the color band
MIL_ID *BufIdPtr;	Storage location for child buffer identifier

Description This function allocates a child data buffer within the specified, previously allocated, color parent data buffer. It selects one of the color bands of the data buffer and allocates the band as a child of that buffer.

The child buffer is not allocated its own memory space; it remains part of the parent buffer. Therefore, any modification to the child buffer affects the parent and vice versa. Note, a parent buffer can have several child buffers.

A color child buffer is considered a data buffer in its own right. It can be any color band of its parent buffer, and can be used in the same circumstances as its parent buffer. A child buffer inherits its type and attributes from the parent buffer.

To allocate a child in one specific band, or specifically in all bands, use **MbufChildColor2d()** instead of **MbufChildColor()**.

When this buffer is no longer required, release it, using **MbufFree()**.

The **ParentBufId** parameter specifies the identifier of the parent buffer. The parent buffer cannot have an M_COMPRESS attribute.

The **Band** parameter specifies the index of the color band of the parent data buffer from which to allocate the child data buffer. This parameter can be set to a value from 0 to (number of bands of the parent buffer - 1). For RGB parent buffers, band 0 corresponds to the red band, band 1 corresponds to the green band, and band 2 corresponds to the blue band. The specified color band should be valid in the parent buffer.

For RGB parent buffers, **Band** can be set to: M_RED, M_GREEN, M_BLUE. For HLS parent buffers, **Band** can be set to: M_HUE, M_LUMINANCE, or M_SATURATION. For YUV parent buffers, **Band** can be set to: M_Y, M_U, or M_V.

The **BufIdPtr** parameter specifies the address of the variable in which the child buffer identifier is to be written. Since the **MbufChildColor()** function also returns the child buffer identifier, you can set this parameter to `M_NULL`. If allocation fails, `M_NULL` is written as the identifier.

Return value The returned value is the child buffer identifier. If allocation fails, `M_NULL` is returned.

See also **MbufAllocColor()**, **MbufChild2d()**, **MbufCopyColor()**, **MbufChildColor2d()**, **MbufFree()**

MbufChildColor2d

Synopsis Allocate a child data buffer within a color parent buffer.

Format **MIL_ID MbufChildColor2d(ParentBufId, Band, OffX, OffY, SizeX, SizeY, BufIdPtr)**

MIL_ID ParentBufId;	Parent buffer identifier
long Band;	Index of the color band
long OffX;	X pixel offset relative to parent buffer
long OffY;	Y pixel offset relative to parent buffer
long SizeX;	X dimension
long SizeY;	Y dimension
MIL_ID *BufIdPtr;	Storage location for child buffer identifier

Description This function allocates a child data buffer within the specified, previously allocated, color parent data buffer. It selects a two-dimensional region in one or all of the color bands of the parent data buffer and allocates the region as a child of that buffer.

The child buffer is not allocated its own memory space; it remains part of the parent buffer. Therefore, any modification to the child buffer affects the parent and vice versa. Note, a parent buffer can have several child buffers.

A color child buffer is considered a data buffer in its own right. It can be used in the same circumstances as its parent buffer. A child buffer inherits its type and attributes from the parent buffer.

When this buffer is no longer required, release it, using **MbufFree()**.

The **ParentBufId** parameter specifies the identifier of the parent buffer. The parent buffer cannot have an M_COMPRESS attribute unless the **Band** parameter is set to M_ALL_BAND.

The **Band** parameter specifies the index of the color band of the parent data buffer from which to allocate the child data buffer. This parameter can be set to a value from 0 to (number of bands of the parent buffer - 1). For RGB parent buffers, band 0 corresponds to the red band, band 1 corresponds to the green band, and band 2 corresponds to the blue band. The specified color band should be valid in the parent buffer.

For RGB parent buffers, **Band** can be set to: M_RED, M_GREEN, M_BLUE. For HLS parent buffers, **Band** can be set to: M_HUE, M_LUMINANCE, or M_SATURATION. For YUV parent buffers, **Band** can be set to: M_Y, M_U, or M_V.

❖ Note that if you are creating a child buffer in the U or V band of a YUV parent buffer, the dimensions of the child buffer must not exceed the dimensions of the U or V band of the parent.

To allocate a child buffer with the same number of bands as the parent buffer, specify M_ALL_BAND.

The **OffX** and **OffY** parameters specify the horizontal and vertical pixel offsets of the child buffer, relative to the parent buffer's top-left pixel. The offsets must be within the width and height of the parent buffer, respectively.

The **SizeX** and **SizeY** parameters specify the width and height of the child buffer, respectively.

The **BufIdPtr** parameter specifies the address of the variable in which the child buffer identifier is to be written. Since the **MbufChildColor2d()** function also returns the child buffer identifier, you can set this parameter to M_NULL. If allocation fails, M_NULL is written as the identifier.

Return value The returned value is the child buffer identifier. If allocation fails, M_NULL is returned.

See also **MbufAllocColor()**, **MbufChild1d()**, **MbufChild2d()**, **MbufChildColor()**, **MbufCopyColor2d()**, **MbufFree()**

MbufChild1d

Synopsis

Allocate a 1D child data buffer.

Format

MIL_ID MbufChild1d(ParentBufId, OffX, SizeX,BufIdPtr)

MIL_ID ParentBufId;	Parent buffer identifier
long OffX;	X pixel offset relative to parent buffer
long SizeX;	Child buffer width
MIL_ID *BufIdPtr;	Storage location for child buffer identifier

Description

This function allocates a one-dimensional child data buffer within the specified, previously allocated parent data buffer. If the parent buffer is multi-band, this function allocates a multi-band child buffer; the child is allocated within the specified one-dimensional region in each color band. To allocate a child in one specific band, or specifically in all bands, use **MbufChildColor2d()** instead of **MbufChild1d()**.

The child buffer is not allocated its own memory space; it remains part of the parent buffer. Therefore, any modification to the child buffer affects the parent and vice versa. Note, a parent buffer can have several child buffers.

A child buffer is considered a data buffer in its own right, and can be used in the same circumstances as its parent buffer. A child buffer inherits its type and attributes from the parent buffer.

When this buffer is no longer required, it can be released using **MbufFree()**.

The **ParentBufId** parameter specifies the identifier of the parent buffer.

The **OffX** parameter specifies the offset of the child buffer relative to the parent buffer's top-left pixel. The offset must be within the width of the parent buffer.

The **SizeX** parameter specifies the width of the child buffer.

The **BufIdPtr** parameter specifies the address of the variable in which the child buffer identifier is to be written. Since the **MbufChild1d()** function also returns the child buffer identifier, you can set this parameter to **M_NULL**. If allocation fails, **M_NULL** is written as the identifier.

Return value

The returned value is the child buffer identifier. If allocation fails, **M_NULL** is returned.

See also

MbufChild2d(), MbufChildColor(), MbufFree()

MbufChild2d

Synopsis Allocate a child buffer within a specific region of a parent buffer.

Format **MIL_ID MbufChild2d(ParentBufId, OffX, OffY, SizeX, SizeY, BufIdPtr)**

MIL_ID ParentBufId;	Parent buffer identifier
long OffX;	X pixel offset relative to the parent buffer
long OffY;	Y pixel offset relative to the parent buffer
long SizeX;	Child buffer width
long SizeY;	Child buffer height
MIL_ID *BufIdPtr;	Storage location for child buffer identifier

Description This function allocates a two-dimensional child buffer within a region of the specified, previously allocated data buffer. If the parent buffer is multi-band, this function allocates a multi-band child buffer; the child is allocated within the specified region in each color band. To allocate a child region in one specific band, or specifically in all bands, use **MbufChildColor2d()** instead of **MbufChild2d()**.

The child buffer is not allocated its own memory space; it remains part of the parent buffer. Any modification to the child buffer affects the parent and vice versa. Note, a parent buffer can have several child buffers.

A child buffer is considered a data buffer in its own right, and can be used in the same circumstances as its parent buffer. A child buffer inherits its type and attributes from the parent buffer.

When this buffer is no longer required, it can be released, using **MbufFree()**.

The **ParentBufId** parameter specifies the identifier of the parent buffer.

The **OffX** and **OffY** parameters specify the horizontal and vertical pixel offsets of the child buffer's top-left pixel, relative to the parent buffer's top-left pixel. The given offsets must be within the width and height of the parent buffer.

The **SizeX** and **SizeY** parameters specify the width and height of the child buffer.

The **BuIdPtr** parameter specifies the address of the variable in which the child buffer identifier is to be written. Since the **MbufChild2d()** function also returns the child buffer identifier, you can set this parameter to `M_NULL`. If allocation fails, `M_NULL` is written as the identifier.

Return value The returned value is the child buffer identifier. If allocation fails, `M_NULL` is returned.

See also **MbufChild1d()**, **MbufChildColor()**, **MbufChildColor2d()**, **MbufFree()**

MbufClear

Synopsis Clears a buffer to a specified color.

Format **void MbufClear(DestImageBufId, Color)**

MIL_ID DestImageBufId;	Destination image buffer identifier
double Color;	Color with which to clear buffer

Description This function clears the entire specified buffer to the specified color.

The **DestImageBufId** parameter specifies the identifier of the image buffer to clear.

The **Color** parameter specifies the grayscale or RGB color value with which to clear the buffer. Set this parameter as follows:

- To clear a 1-band buffer, set this parameter to any value. This value will be cast to the type of the destination buffer.
- To clear a multi-band buffer to a grayscale value, set this parameter to any value. This value will be cast to the type of the destination buffer's bands and replicated in each band.
- To clear an 8-bit 3-band buffer to an RGB color, set this parameter using the following macro:

 M_RGB888(red component, green component, blue component)

- To clear a 16-bit or 32-bit multi-band buffer to a color value, use **MgraControl()**.

See also **MgraClear()**

MbufControl

Synopsis Control specified buffer features.

Format **void MbufControl(BufId, ControlType, ControlValue)**

MIL_ID BufId;	Buffer identifier
long ControlType;	Type of buffer feature to control
double ControlValue;	Value associated with control type

Description This function allows you to control certain buffer features.

The **BufId** parameter specifies the identifier of the buffer.

The **ControlType** and **ControlValue** parameters specify the buffer feature to control and the value needed for the control. These two parameters should be set to one of the following:

ControlType	ControlValue	Description
M_ASSOCIATED_LUT	LUT buffer identifier	<p>Associate a LUT buffer with the specified image buffer for display purposes. The image buffer must be a 1-band 8-bit buffer.</p> <p>If and when the image buffer is selected to a windowed display, the required changes occur to produce the display effect of the LUT, unless the display is also associated with a custom LUT (MdispLut()). For windowed displays, MIL indirectly programs the physical output LUTs with the image's associated LUT (through the use of a Windows palette). In general, the LUT will not be used when the image is selected to an auxiliary display.</p> <p>MIL checks the target system to determine whether or not a LUT is supported. If not, an error is generated.</p> <p>To dissociate a LUT buffer from an image buffer, set ControlValue to M_DEFAULT.</p>

ControlType	ControlValue	Description
M_MODIFIED	M_DEFAULT	Signal MIL that the buffer content was modified without using MIL. This control must be used to ensure that MIL updates its internal information on the buffer. For example, if a display buffer was modified outside MIL, the display will not be updated until you use this control. Note, if only a certain region of the buffer was modified, it is more efficient to specify an appropriate child buffer as BufId .
M_WINDOW_DC_ALLOC	M_DEFAULT	Allocate a Windows display context (DC) for drawing. Determine the DC handle (HDC) using MbufInquire() with the M_WINDOW_DC inquire type. When using this control type, the buffer must be internally stored in M_DIB or M_DDRAW format, and cannot be a child buffer. The display context must be allocated and used only for a very short period of time; free it as soon as possible.
M_WINDOW_DC_FREE	M_DEFAULT	Free a Windows display DC.

For buffers with an M_IMAGE + M_COMPRESS attribute, **ControlType** and **ControlValue** can also be set to one of the following:

- ❖ If the buffer contains any data, setting one of these control types automatically deletes the data. This is because, for MIL to decompress the buffer's data, it must know the control values that were used in the compression. If you change one of these controls, MIL will be unable to decompress the data and the data is therefore irrelevant. Note that the control type M_TARGET_SIZE is the only control type that can be changed without affecting the data. See the *MIL User Guide, Chapter 23: JPEG and JPEG 2000 compression* for more information.

ControlType	ControlValue	Description
M_DECOMPOSITION_LEVEL	integer value greater than or equal to 0	Specify the number of iterations the discrete wavelet transform is passed on the image. This control type is supported with JPEG 2000 compression only (both lossy and losseless). Add M_ALL_BAND to apply the number of decomposition levels to all bands of the buffer. To specify the number of decomposition levels for a specific band, add one of the following: M_RED, M_GREEN, and M_BLUE (for RGB buffers), or M_Y, M_U, or M_V (for YUV buffers). When a band is not specified, M_ALL_BAND is assumed.
M_HUFFMAN_AC	ID of buffer with M_ARRAY attribute	Associate an AC Huffman table to the buffer. Only used for JPEG lossy compressions. If the buffer is 3-band, the same table is applied to all bands.
M_HUFFMAN_AC_LUMINANCE	ID of buffer with M_ARRAY attribute	Associate an AC Huffman table to the buffer. Only used for JPEG lossy compressions of YUV buffers. The table is applied only to the Y band.
M_HUFFMAN_AC_CHROMINANCE	ID of buffer with M_ARRAY attribute	Associate an AC Huffman table to the buffer. Only used for JPEG lossy compressions of YUV buffers. The table is applied to the U and V bands.
M_HUFFMAN_DC	ID of buffer with M_ARRAY attribute	Associate a DC Huffman table to the buffer. Only used for JPEG compression (both lossy and lossless). If the buffer is 3-band, the same table is applied to all bands.

ControlType	ControlValue	Description
M_HUFFMAN_DC_LUMINANCE	ID of buffer with M_ARRAY attribute	Associate a DC Huffman table to the buffer. Only used for JPEG compressions (both lossy and lossless) of YUV buffers. The table is applied only to the Y band. Can only be used if the compressed image buffer is of a JPEG lossy type.
M_HUFFMAN_DC_CHROMINANCE	ID of buffer with M_ARRAY attribute	Associate a DC Huffman table to the buffer. Only used for JPEG compressions (both lossy and lossless) of YUV buffers. The table is applied to the U and V bands. Can only be used if the compressed image buffer is of a lossy type.
M_PREDICTOR	0, 1 (default), or 2	For JPEG lossless compressions, use predictor #0 (no prediction), predictor #1 (the "pixel-to-the-left" predictor), or predictor #2 (the "pixel-above" predictor). If the buffer is 3-band, the same predictor is applied to all bands.
M_Q_FACTOR	integer value between 1 and 99; default value is 50	Quantization factor for lossy compressions. The higher the factor, the more the compression, but the lower the image quality. In JPEG lossy, the Q factor is applied to all bands. For JPEG 2000 lossy compressions, add M_ALL_BAND to apply the Q factor to all bands of the buffer. To specify the Q factor for a specific band add one of the following: M_RED, M_GREEN, and M_BLUE (for RGB buffers), or M_Y, M_U, or M_V (for YUV buffers). When a band is not specified, M_ALL_BAND is assumed.

ControlType	ControlValue	Description
M_Q_FACTOR_LUMINANCE	integer value between 1 and 99; default value is 50	Quantization factor for JPEG or JPEG 2000 lossy compressions of YUV images. The higher the factor, the more the compression, but the lower the image quality. The factor is applied only to the Y band.
M_Q_FACTOR_CHROMINANCE	integer value between 1 and 99; default value is 50	Quantization factor for JPEG or JPEG 2000 lossy compressions of YUV images. The higher the factor, the more the compression, but the lower the image quality. The factor is applied to the U and V bands.
M_QUANTIZATION	ID of buffer with M_ARRAY attribute	Associate a quantization table to the buffer for a JPEG compression or JPEG 2000 lossy compression. For JPEG 2000 lossy compressions, add M_ALL_BAND to associate the same quantization table to all the bands. To associate a table to a specific band, add one of the following: M_RED, M_GREEN, and M_BLUE (for RGB buffers), or M_Y, M_U, or M_V (for YUV buffers). When a band is not specified, M_ALL_BAND is assumed.
M_QUANTIZATION_LUMINANCE	ID of buffer with M_ARRAY attribute	Associate a quantization table to the buffer. Only used for JPEG or JPEG 2000 lossy compressions of YUV buffers. The table is applied only to the Y band.
M_QUANTIZATION_CHROMINANCE	ID of buffer with M_ARRAY attribute	Associate a quantization table to the buffer. Only used for JPEG or JPEG 2000 lossy compressions of YUV buffers. The table is applied to the U and V bands.

ControlType	ControlValue	Description
M_RESTART_INTERVAL	any integer value greater than 0; default value is 8	Place restart markers after every n rows of data (for JPEG lossless compressions) or after every n 8x8 blocks of data (for JPEG lossy compressions).
M_TARGET_SIZE	any integer value greater than 0	Specify the size in bytes of the buffer into which the source image is compressed. To ensure that the compressed data fits into the specified buffer size, less-significant data is discarded. This control type is supported with JPEG 2000 lossy compression only.

Note that setting the M_QUANTIZATION control type will reset the M_Q_FACTOR control type to its default value (50). If you set the M_Q_FACTOR control type after specifying a custom table with the M_QUANTIZATION control type, the custom table will be scaled accordingly.

Note The **ControlType** M_ASSOCIATED_LUT is not available with 32-bit or floating-point buffers.

MIL-Lite does not support JPEG2000 compression, and requires dedicated hardware for JPEG compression. This is not a restriction under MIL.

See also **MbufLoad()**, **MbufRestore()**, **MbufImport()**, **MbufExport()**, **MbufSave()**

MbufControlNeighborhood

Synopsis Change the value of an operation flag associated with a custom kernel or structuring element.

Format **void MbufControlNeighborhood(BufId, OperationFlag, OperationValue)**

MIL_ID BufId;	Kernel or structuring element buffer identifier
long OperationFlag;	Operation flag
long OperationValue;	Operation value

Description This function changes the value of an operation flag associated with a custom kernel or structuring element. Neighborhood operations not specifically altered by this function use the default values. After calling this function, any neighborhood operation using the specified kernel or structuring element will apply the specified change. Call this function for each operation flag you want to modify.

The **BufId** parameter specifies the identifier of the custom-kernel buffer or structuring-element buffer. You must have already allocated this buffer, using **MbufAlloc1d0** or **MbufAlloc2d0**.

The **OperationFlag** parameter specifies the action to perform on a neighborhood operation when using the specified buffer.

The **OperationValue** parameter specifies the value associated with the operation flag.

The following table lists the possible values that can be specified for each operation flag:

Operation flag	Operation value	Description
M_NORMALIZATION_FACTOR	Any numerical value	The result is normalized by the specified value (factor).
	M_DEFAULT	The result is normalized by a factor of 1.
M_ABSOLUTE_VALUE	M_ENABLE	The absolute value of the result is taken.
	M_DISABLE	The absolute value of the result is not taken.
	M_DEFAULT	Same as M_DISABLE.

Operation flag	Operation value	Description
M_SATURATION	M_ENABLE	Saturation is performed on the result. That is, a result that overflows or underflows will be set to the maximum or minimum value (respectively) that can be represented in the destination buffer.
	M_DISABLE	Saturation is not performed on the result. Therefore, the results that overflow are undefined.
	M_DEFAULT	Same as M_DISABLE.
M_OVERSCAN	M_DEFAULT	MIL-selected method to optimize speed and logic in function of both the operation required and the current processing system.
	M_MIRROR	Operations will be performed on the bordering pixels of the source buffer with overscan neighborhood pixel values which mirror the source buffer pixel values. That is, the overscan neighborhood pixel values will be a mirror copy of the source buffer's borders. <div><div><div><div>Row #</div><div>-2</div><div>-1</div><div>0</div><div>1</div><div>2</div></div><div><div><div><div>-2</div><div>-1</div><div>0</div><div>1</div><div>2</div></div><div><div><div><div>e</div><div>d</div><div>b</div><div>a</div><div>h</div></div><div><div><div><div>d</div><div>e</div><div>a</div><div>b</div><div>g</div></div><div><div><div><div>d</div><div>e</div><div>a</div><div>b</div><div>g</div></div><div><div><div><div>f</div><div>c</div><div>i</div></div></div></div></div><div><div><div><div>↑</div><div>↓</div></div><div><div><div>←</div><div>→</div></div></div></div></div><div><div>Column #</div><div>Overscan values</div><div>Image</div><div>Overscan values</div></div></div></div></div></div></div></div></div></div></div>
	M_REPLACE	Operations will be performed on the bordering pixels of the source buffer with the overscan neighborhood pixel values set to the overscan replace value.

Operation flag	Operation value	Description
M_OVERSCAN (cont)	M_TRANSPARENT	Operations will be performed on the bordering pixels of the source buffer using transparent overscan neighborhood pixel values. That is, the overscan neighborhood pixel values will be those of the parent buffer. If they are not available, a mirror type overscan is used instead
	M_DISABLE	Overscan is disabled.
M_OVERSCAN_REPLACE_VALUE		
	Any numerical value	Value of the overscan neighborhood pixels.
	M_REPLACE_MAX	The overscan neighborhood pixel values will be set to the maximum value of the source buffer.
	M_REPLACE_MIN	The overscan neighborhood pixel values will be set to the minimum value of the source buffer
	M_DEFAULT	Zero will be used as the value of the overscan neighborhood pixels.
M_OFFSET_CENTER_X	Any value from 0...(sizeX-1)	Position X of the center of the kernel or structuring element from the top-left corner.
	M_DEFAULT	The top-left pixel of the central element in a neighborhood.
M_OFFSET_CENTER_Y	Any value from 0...(sizeY-1)	Position Y of the center of the kernel or structuring element from the top-left corner.
	M_DEFAULT	The top-left pixel of the central element in a neighborhood.
M_DEFAULT	M_NULL	Default values (as indicated above for each operation flag) will be used.

For a structuring element buffer, you cannot specify a normalization factor or take the absolute value of a result.

If the saturation, normalization, and absolute value options are specified for a kernel buffer, the saturation is performed after the normalization factor and the absolute values have been applied.

The `M_OVERSCAN_REPLACE_VALUE` flag is only applicable if associating an `M_OVERSCAN` with an `M_REPLACE` value to a buffer.

Example `mconvol.c`

See also `MimConvolve()`, `MimMorphic()`, `MimRank()`, `MbufAlloc1d()`, `MbufAlloc2d()`, `MbufPut()`

MbufCopy

Synopsis Copy data from one buffer to another.

Format **void MbufCopy(SrcBufId, DestBufId)**

MIL_ID SrcBufId;	Source buffer identifier
MIL_ID DestBufId;	Destination buffer identifier

Description This function copies the specified source buffer data to the specified destination buffer. If the source and destination buffers are of different data types, MIL converts the data automatically.

If the source buffer depth is greater than that of the destination, the most significant bits are truncated when the data is copied into the destination. If the destination depth is greater than that of the source, the source data is zero or sign-extended (depending on the type of the source) when copied into the destination. If the destination is larger in size than the source, exceeding areas of the buffer are unaffected.

Note, 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()**.

When copying from a floating-point buffer to an integer buffer, the values are truncated.

If the source buffer is a 3-band YUV buffer and the destination buffer is a 1-band buffer, only the Y band (luminance) is copied. If the source buffer is a 3-band RGB buffer and the destination buffer is a 1-band buffer, only the red band is copied.

The **SrcBufId** and **DestBufId** parameters specify the identifiers of the source and destination data buffers.

Note This function is optimized for packed binary buffers.

See also **MbufCopyClip()**, **MbufCopyCond()**, **MbufCopyMask()**, **MbufCopyColor()**, **MbufCopyColor2d()**

MbufCopyClip

Synopsis Copy buffer, clipping data outside the destination buffer.

Format `void MbufCopyClip(SrcBufId, DestBufId, DestOffX, DestOffY)`

MIL_ID SrcBufId;	Source buffer identifier
MIL_ID DestBufId;	Destination buffer identifier
long DestOffX;	X pixel offset relative to destination buffer
long DestOffY;	Y pixel offset relative to destination buffer

Description This function copies the source buffer data to the destination buffer starting at the specified offset. Data outside of the destination buffer is not copied (it is clipped).

If the source buffer depth is greater than that of the destination, the most significant bits are truncated when the data is copied to the destination. If the destination depth is greater than that of the source, the source data is zero or sign-extended (depending on the type of the source) when copied to the destination.

Note, 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()**.

When copying from a floating-point buffer to an integer buffer, the values are truncated.

The **SrcBufId** and **DestBufId** parameters specify the identifiers of the source and destination data buffers.

The **DestOffX** and **DestOffY** parameters specify the horizontal and vertical pixel offsets of the destination buffer area at which to start copying data. Specify offsets relative to the top-left corner of the destination buffer (0,0). These two parameters can be set to negative values and can be specified anywhere outside the destination buffer. Data extending beyond the limits of the destination buffer is not copied (it is clipped).

Note This function is optimized for packed binary buffers.

See also **MbufCopy()**, **MbufCopyCond()**, **MbufCopyMask()**

MbufCopyColor

Synopsis Copy one or all bands of an image buffer.

Format **void MbufCopyColor(SrcBufId, DestBufId, Band)**

MIL_ID SrcBufId;	Source buffer identifier
MIL_ID DestBufId;	Destination buffer identifier
long Band;	Index of the color band to copy

Description This function copies one or all color bands of the specified source buffer to the specified destination buffer. It can also be used to insert or extract a color component from a color image.

The **SrcBufId** and **DestBufId** parameters specify the identifiers of the source and destination data buffers.

The **Band** parameter specifies the index of the color band to copy. This parameter can be set to any index from 0 to (number of bands of the buffer - 1), where band 0 is red, band 1 is green, and band 2 is blue, or to one of the following:

M_RED	Copy to/from the red color band.
M_GREEN	Copy to/from the green color band.
M_BLUE	Copy to/from the blue color band.
M_ALL_BAND	Copy all color bands.

The **Band** parameter gives the index of the color band to extract or insert. If the source is a monochrome buffer and the destination is a multi-band (color) buffer, the unique source buffer band is inserted into the specified band of the destination buffer. If the source is a multi-band buffer and the destination is a monochrome buffer, the specified source buffer band is extracted from the source buffer and written to the destination buffer. If both are multi-band buffers, the specified band(s) is copied from the source to the destination.

If the source buffer is in a HLS (hue, luminance, and saturation) format, the band can be set to: M_HUE, M_LUMINANCE, M_SATURATION, or M_ALL_BAND.

If the source buffer depth is greater than that of the destination, the most significant bits are truncated when the data is copied to the destination. If the destination depth is greater than that of the source, the source data is zero or sign-extended (depending on the type of the source) when copied to the destination. Also, the buffers must have the same number of bands if all bands are to be copied.

Note, 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.

Note This function is optimized for packed binary buffers.

See also `MbufCopy()`, `MbufCopyClip()`, `MbufCopyCond()`, `MbufCopyMask()`

MbufCopyColor2d

Synopsis Copy a two-dimensional region of one or all bands of an image buffer to another buffer.

Format **void MbufCopyColor2d(SrcBufId, DestBufId, SrcBand, SrcOffX, SrcOffY, DestBand, DestOffX, DestOffY, SizeX, SizeY)**

MIL_ID SrcBufId;	Source buffer identifier
MIL_ID DestBufId;	Destination buffer identifier
long SrcBand;	Index of the source color band to copy
long SrcOffX;	X pixel offset relative to the source parent buffer
long SrcOffY;	Y pixel offset relative to the source parent buffer
long DestBand;	Index of the destination color band
long DestOffX;	X pixel offset relative to the destination parent buffer
long DestOffY;	Y pixel offset relative to the destination parent buffer
long SizeX;	X dimension
long SizeY;	Y dimension

Description This function copies a two-dimensional region of one or all color bands of the specified source buffer to the specified color band(s) of the destination buffer. It can also be used to insert or extract a color component from a color buffer.

The **SrcBufId** and **DestBufId** parameters specify the identifiers of the source and destination data buffers.

The **SrcBand** and **DestBand** parameters specify the index of the source and destination color bands. These parameters can be set to any index from 0 to (number of bands of the buffer - 1), where band 0 is red, band 1 is green, and band 2 is blue or to one of the following:

M_RED	Copy to/from the red color band.
M_GREEN	Copy to/from the green color band.
M_BLUE	Copy to/from the blue color band.
M_ALL_BAND	Copy all color bands.

If the source is a monochrome buffer and the destination is a multi-band (color) buffer, the unique source buffer band is inserted into the specified band of the destination buffer. If the source is a multi-band buffer and the destination is a monochrome buffer, the specified source buffer band is extracted from the source buffer and written to the destination buffer. If both are multi-band buffers, the specified band(s) is copied from the source to the destination.

If the source buffer is in a HLS (hue, luminance, and saturation) format, the band can be set to: M_HUE, M_LUMINANCE, M_SATURATION, or M_ALL_BAND.

If the source buffer is a YUV buffer, the band can be set to: M_Y, M_U, M_V, or M_ALL_BAND. Note that when copying into the U or V band of a YUV buffer, the dimensions of the source buffer must not exceed the dimensions of the U or V band of the destination.

If the source buffer depth is greater than that of the destination, the most significant bits are truncated when the data is copied to the destination. If the destination depth is greater than that of the source, the source data is zero or sign-extended (depending on the type of the source) when copied to the destination. Also, the buffers must have the same number of bands if all bands are to be copied.

Note, 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.

The **SrcOffX** parameter specifies the horizontal pixel offset of the region to read relative to the source buffer starting coordinate. The offset must be within the width of the source buffer.

The **SrcOffY** parameter specifies the vertical pixel offset of the region to read relative to the source buffer starting coordinate. The offset must be within the height of the source buffer.

The **DestOffX** parameter specifies the horizontal pixel offset of the region to write relative to the destination buffer starting coordinate. The offset must be within the width of the destination buffer.

The **DestOffY** parameter specifies the vertical pixel offset of the region to write relative to the destination buffer starting coordinate. The offset must be within the height of the destination buffer.

The **SizeX** parameter specifies the width of the region to be copied, starting from the specified offset (**SrcOffX**, **DestOffX**).

The **SizeY** parameter specifies the height of the region to be copied, starting from the specified offset (**SrcOffY**, **DestOffY**).

Note This function is optimized for packed binary buffers.

See also **MbufCopy()**, **MbufCopyClip()**, **MbufCopyColor()**, **MbufCopyCond()**, **MbufCopyMask()**

MbufCopyCond

Synopsis Copy conditionally the source buffer to the destination buffer.

Format **void MbufCopyCond(SrcBufId, DestBufId, CondBufId, Condition, CondValue)**

MIL_ID SrcBufId;	Source buffer identifier
MIL_ID DestBufId;	Destination buffer identifier
MIL_ID CondBufId;	Condition buffer identifier
long Condition;	Processing condition
double CondValue;	Condition value

Description This function copies the source buffer data to the destination buffer, modifying only those pixels of the destination buffer that have a corresponding pixel in the conditional buffer that satisfies the specified condition. Other pixels are unchanged. If the source and destination buffers are of different data types, MIL converts the data automatically.

The **SrcBufId** and **DestBufId** parameters specify the identifiers of the source and destination data buffers. If the source buffer depth is greater than that of the destination, the most-significant bits are truncated when the data is copied to the destination. If the destination depth is greater than that of the source, the source data is zero or sign-extended (depending on the type of the source) when copied to the destination. For example, the data is zero-extended when copying an 8-bit unsigned buffer to a 16-bit unsigned buffer.

The **CondBufId** parameter specifies the identifier of the condition buffer.

Note that if a one-band condition buffer is used with a three-band destination buffer, the one band of the condition buffer will be used for each destination band.

The **Condition** parameter specifies the condition for which the condition buffer is tested. This parameter can be set to one of the following:

M_EQUAL	Modify destination buffer pixels corresponding to condition buffer pixels that are equal to CondValue .
M_NOT_EQUAL	Modify destination buffer pixels corresponding to condition buffer pixels that are not equal to CondValue .
M_DEFAULT	Modify destination buffer pixels corresponding to condition buffer pixels that are non-zero.

The **CondValue** parameter specifies the pixel value for the specified condition. Even though this value is of type ‘double’, it is treated as if it had the same type and depth as the condition buffer. If M_DEFAULT is used, **CondValue** is ignored. If the condition buffer is binary, this value must be 0 or 1. If the condition buffer is three bands, this value will be used for each band.

If the source buffer, destination buffer, and condition buffer are 8-bit unsigned, and the condition buffer is a 3-band buffer, then the **CondValue** parameter can also use the RGB macro:

M_RGB888(red component, green component, blue component)

This allows you to specify a color value for each band in the condition buffer.

Note This function is optimized for packed binary buffers.

See also **MbufCopy()**, **MbufCopyClip()**, **MbufCopyMask()**

MbufCopyMask

Synopsis Copy buffer with mask.

Format **void MbufCopyMask(SrcBufId, DestBufId, MaskValue)**

MIL_ID SrcBufId;	Source buffer identifier
MIL_ID DestBufId;	Destination buffer identifier
long MaskValue;	Mask value to apply to the destination buffer

Description This function copies the specified source buffer data to the specified destination buffer, modifying only the bits of the destination that have a non-zero corresponding bit in the mask.

The **SrcBufId** and **DestBufId** parameters specify the identifiers of the source and destination data buffers.

The **MaskValue** parameter specifies the mask value. Even though this value is of type 'long', it is treated as if it had the same depth as the destination buffer; the most-significant bits that are not required are ignored. If the destination buffer is binary, the value must be 0 or 1.

If the source buffer depth is greater than that of the destination, the most significant bits are truncated when the data is copied to the destination. If the destination depth is greater than that of the source, the source data is zero or sign-extended (depending on the type of the source) when copied to the destination.

Status Not available on floating-point buffers.

See also **MbufCopy()**, **MbufCopyClip()**, **MbufCopyCond()**

MbufCreate2d

Synopsis Create a two-dimensional data buffer.

Format MIL_ID MbufCreate2d(SystemId, SizeX, SizeY, Type, Attribute, ControlFlag, Pitch, DataPtr, BufIdPtr)

MIL_ID SystemId;	System identifier
long SizeX;	X dimension
long SizeY;	Y dimension
long Type;	Data type and data depth
long Attribute;	Buffer attributes
long ControlFlag;	Creation control flag
long Pitch;	Value of pitch if necessary
void *DataPtr;	Pointer to data
MIL_ID *BufIdPtr;	Storage location for buffer identifier

Description This function creates a two-dimensional data buffer that maps to a user-specified data array and associates it with a specific MIL system. **This function should be used with caution because, when using physical addresses, they provide direct manipulation of any of your PC's memory mapped devices; when using logical addresses, memory protection errors could result.** It is generally better to leave buffer allocation, data loading, and memory control to MIL (**MbufAlloc2d()**, **MbufGet2d()**, **MbufPut2d()**), since MIL might require special memory attributes (such as non-paged memory) or alignment in order to associate the buffer with a particular target system.

The appropriate memory must be allocated by the user before calling **MbufCreate2d()** and freed when no longer required, after calling **MbufFree()**.

The **SystemId** parameter specifies the MIL system with which the buffer will be associated. This parameter must be set to a valid system identifier, M_DEFAULT_HOST or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system with which to associate the buffer (it can be the Host system or any already allocated system).

The **SizeX** and **SizeY** parameters specify the buffer width and height, respectively, in the units appropriate for the selected buffer attribute. For example, if the buffer has an image buffer attribute, width and height are specified in pixels.

The **Type** parameter specifies a combination of two values: data type and data depth. Express the depth in bits and give the data range as one of the following:

Data Type	Description	Depth (in bits)
M_SIGNED	Signed data	8, 16, or 32
M_UNSIGNED	Unsigned data (default)	1, 8, 16, or 32
M_FLOAT	Floating point data	32

For example, when allocating a 8-bit unsigned buffer, you would set the **Type** parameter to (8 + M_UNSIGNED).

The **Attribute** parameter defines the buffer usage. This parameter should be set to one of the following:

M_IMAGE	Image data.
M_LUT	Lookup table.
M_KERNEL	Convolution kernel for convolution functions.
M_STRUCT_ELEMENT	Structuring element for morphology functions.
M_ARRAY	Array of data. Note that some functions take an M_ARRAY buffer rather than a user-defined array.

When selecting an M_IMAGE attribute, it should be set to M_IMAGE + *specifier*. For example, to create an image buffer that can be processed and displayed, you should set the **Attribute** parameter to M_IMAGE + M_PROC + M_DISP. The specifier can be one or more of the following:

Usage specifiers:	
M_DISP	An image buffer that can be displayed.
M_GRAB	An image buffer in which to grab data from input devices. To specify this attribute, the memory must usually be physically contiguous, non-paged memory.
M_PROC	An image buffer that can be processed.
M_COMPRESS	An image buffer that can hold compressed data. See MbufAlloc...() for a list of compression specifiers. Note that a buffer with this attribute cannot have the M_SIGNED data type.

Compressed buffers should not be used as the destination buffer of a MIL function. If a buffer with an `M_COMPRESS` specifier is used as a source buffer for an operation, the data will be decompressed depending on the attributes of the destination buffer.

You must specify the appropriate internal storage format of the buffer; MIL needs this information to manipulate the data.

Board-dependent location specifiers:	
<code>M_PAGED</code>	Buffer is in pageable memory.
<code>M_NON_PAGED</code>	Buffer is in non-pageable memory.
Board-dependent internal storage format specifiers:	
<code>M_FLIP</code>	The buffer is top down (DIB).
<code>M_NO_FLIP</code>	The buffer is top up.

The **ControlFlag** parameter specifies the physical nature of the buffer. It can be set to one of the following:

ControlFlag	Description
<code>M_DEFAULT</code>	Same as <code>+M_PITCH</code> . The pitch is the width (size X) of the buffer.
<code>M_HOST_ADDRESS +M_PITCH</code>	DataPtr is the Host address of the data buffer. The pitch is in pixels.
<code>M_HOST_ADDRESS +M_PITCH_BYTE</code>	DataPtr is the Host address. The pitch is in bytes.
<code>M_PHYSICAL_ADDRESS +M_PITCH</code>	DataPtr is the physical address of the data buffer in memory. The pitch is in pixels.
<code>M_PHYSICAL_ADDRESS +M_PITCH_BYTE</code>	DataPtr is the physical address of the data buffer. The pitch is in bytes.

The **Pitch** parameter specifies the pitch in pixels or bytes (as determined by **ControlFlag**) or `M_DEFAULT`. The pitch is the length of the buffer's memory (not data) line.

The **DataPtr** parameter is a pointer to the data array to which to map the created MIL buffer.

The **BufIdPtr** parameter specifies the address of the variable in which the buffer identifier is to be written. Since the **MbufCreate2d()** function also returns the buffer identifier, you can set this parameter to `M_NULL`. If allocation fails, `M_NULL` is written as the identifier.

Note MIL-Lite does not support JPEG 2000 compression, and requires dedicated hardware for JPEG compression. This is not a restriction under MIL.

This function is optimized for packed binary buffers.

Return value The returned value is the buffer identifier. If allocation fails, an identifier of 0 is returned.

Status Current limitation:

- For M_KERNEL data buffers, the data type must be 8-bit signed or unsigned.
- For M_STRUCT_ELEMENT data buffers, the data type must be 32-bit signed or unsigned. If signed, the range is -32768 to +32767. If unsigned, the range is 0 to +65535.

See also **MbufAlloc2d()**, **MbufGet2d()**, **MbufPut2d()**, **MbufFree()**

MbufCreateColor

Synopsis Create a color data buffer.

Format `MIL_ID MbufCreateColor(SystemId, SizeBand, SizeX, SizeY, Type, Attribute, ControlFlag, Pitch,ArrayOfDataPtr, BufIdPtr)`

<code>MIL_ID SystemId;</code>	System identifier
<code>long SizeBand;</code>	Number of color bands
<code>long SizeX;</code>	X dimension
<code>long SizeY;</code>	Y dimension
<code>long Type;</code>	Data type and data depth per band
<code>long Attribute;</code>	Buffer attributes
<code>long ControlFlag;</code>	Creation control flag
<code>long Pitch;</code>	Value of pitch, if necessary
<code>void **ArrayOfDataPtr;</code>	Array of data buffer pointers
<code>MIL_ID *BufIdPtr;</code>	Storage location for buffer identifier

Description This function creates a color data buffer that maps to a user-specified data array and associates it with a specific MIL system. **This function should be used with caution because, when using physical addresses, they provide direct manipulation of any of your PC's memory mapped devices; when using logical addresses, memory protection errors could result.** It is generally better to leave buffer allocation, data loading, and memory control to MIL (**MbufAllocColor()**, **MbufGetColor()**, **MbufPutColor()**), since MIL might require special memory attributes (such as non-paged memory) or alignment in order to associate the buffer with a particular target system. **MbufInquire()** can be used to get the pointer to a MIL allocated buffer.

The appropriate memory must be allocated by the user before calling **MbufCreateColor()** and freed when no longer required, after calling **MbufFree()**.

The **SystemId** parameter specifies the MIL system with which the buffer will be associated. This parameter must be set to a valid system identifier, `M_DEFAULT_HOST`, or `M_DEFAULT`. To use the default Host system of the current MIL application, specify `M_DEFAULT_HOST`. If you specify

M_DEFAULT, MIL will select the most appropriate system with which to associate the buffer (it can be the Host system or any already allocated system).

The **SizeBand** parameter specifies the number of (x,y) surfaces (also called color bands) that the buffer should have in order to represent the color components of an object. When acquiring or processing monochrome images, the buffer requires only one color band. For RGB color images, it requires three color bands. The possible range for this parameter is 1 to n . However, there are generally either 1 or 3 bands.

The **SizeX** and **SizeY** parameters specify the buffer width and height, respectively, in the units appropriate for the selected buffer attribute. For example, if the buffer has an image buffer attribute, width and height are specified in pixels.

The **Type** parameter specifies a combination of two values: data type and data depth per band. Express the depth in bits and give the data range as one of the following:

Data Type	Description	Depth (in bits)
M_SIGNED	Signed data	8, 16, or 32
M_UNSIGNED	Unsigned data (default)	1, 8, 16, or 32
M_FLOAT	Floating point data	32

For example, when allocating a 8-bit unsigned buffer, you would set the **Type** parameter to (8 + M_UNSIGNED).

The **Attribute** parameter specifies the buffer usage. This parameter should be set to M_LUT, or to M_IMAGE + *specifier*. For example, to create an image buffer that can be processed and displayed, you should set the **Attribute** parameter to M_IMAGE + M_PROC + M_DISP. The specifier can be one or more of the following:

Usage specifiers:	
M_DISP	An image buffer that can be displayed.
M_GRAB	An image buffer in which to grab data from input devices. To specify this attribute, the memory must usually be physically contiguous, non-paged memory.
M_PROC	An image buffer that can be processed.
M_COMPRESS	An image buffer that can hold compressed data. See MbufAllocColor() for a list of compression specifiers. Note that a buffer with this attribute cannot have the M_SIGNED data type.

Compressed buffers should not be used as the destination buffer of a MIL function. If a buffer with an M_COMPRESS specifier is used as a source buffer for an operation, the data will be decompressed depending on the attributes of the destination buffer.

You must specify the appropriate internal storage format of the buffer; MIL needs this information to manipulate the data. For example, you do not want MIL to interpret a packed data buffer as a planar.

Board-dependent location specifiers:	
M_PAGED	Buffer is in pageable memory.
M_NON_PAGED	Buffer is in non-pageable memory.

Board-dependent internal storage format specifiers:	
M_FLIP	The buffer is top down (DIB).
M_NO_FLIP	The buffer is top up.
M_PACKED	The buffer bands are packed.
M_PLANAR	The buffer bands are planar.

For the following specifiers, the buffer must be an 8-bit multi-band buffer. See *MIL/MIL-Lite Board-Specific Notes* to verify which formats are supported on your board.

Note that it might be slower to use buffers that have been forced with one of these attributes. Although there is no right or wrong storage format to use, certain operations are optimized for some formats.

Board-dependent internal storage format specifiers:	
M_RGB15+M_PACKED	16-bit packed pixels (XRGB 1:5:5:5). Note that when accessing an M_RGB15+M_PACKED buffer as a 3-band 8-bit buffer, the least significant bits are set to 0.
M_RGB16+M_PACKED	16-bit packed pixels (RGB 5:6:5). Note that when accessing an M_RGB16+M_PACKED buffer as a 3-band 8-bit buffer, the least significant bits are set to 0.
M_BGR24+M_PACKED	24-bit (BGR) packed pixels.
M_BGR32+M_PACKED	32-bit (BGR) packed pixels.
M_RGB24+M_PLANAR	24-bit (RGB) planar pixels
M_YUV9+M_PLANAR	YUV9 planar standard.
M_YUV12+M_PLANAR	YUV12 planar standard.

Board-dependent internal storage format specifiers:	
M_YUV16+M_PACKED	YUV16 packed (4:2:2) standard.
M_YUV16_UYVY+M_PACKED	YUV16 packed (4:2:2) standard.
M_YUV16_YUYV+M_PACKED	YUV16 packed (4:2:2) standard.
M_YUV24+M_PLANAR	YUV24 planar standard.

The **ControlFlag** parameter specifies the physical nature of the buffer. It can be set to one of the following:

ControlFlag	Description
M_DEFAULT	Same as +M_PITCH. The pitch is the width (size X) of the buffer.
M_HOST_ADDRESS + M_PITCH	The data pointer is the Host address of the data buffer. The pitch is in pixels.
M_HOST_ADDRESS +M_PITCH_BYTE	The data pointer is the Host address. The pitch is in bytes.
M_PHYSICAL_ADDRESS +M_PITCH	The data pointer is the physical address of the data buffer in memory. The pitch is in pixels.
M_PHYSICAL_ADDRESS +M_PITCH_BYTE	The data pointer is the physical address of the data buffer in memory. The pitch is in bytes.

The **Pitch** parameter specifies the pitch in pixels or bytes (as determined by **ControlFlag**) or M_DEFAULT. The pitch is the number of pixels or bytes (as specified by the **ControlFlag**) between the beginnings of any two adjacent lines of the buffer data. Note that when creating an M_BGR24 + M_PACKED buffer, you should use M_PITCH_BYTE instead of M_PITCH because the latter might not be able to take into account internal padding.

The **ArrayOfDataPtr** parameter is the address of an array of pointers. These pointers address the data buffers to which to map the created MIL buffer. When pointing to a planar buffer, one pointer per band must be provided. Pointers to a 3-band planar buffer must be ordered R-G-B or Y-U-V in the array. When pointing to a single-band buffer or a packed buffer, a pointer to the packed data must be provided.

The **BufIdPtr** parameter specifies the address of the variable in which the buffer identifier is to be written. Since the **MbufCreateColor()** function also returns the buffer identifier, you can set this parameter to M_NULL. If allocation fails, M_NULL is written as the identifier.

Note MIL-Lite does not support JPEG 2000 compression, and requires dedicated hardware for JPEG compression. This is not a restriction under MIL.

This function is optimized for packed binary buffers.

Return value The returned value is the buffer identifier. If allocation fails, M_NULL is returned.

See also **MbufAllocColor(), MbufGetColor(), MbufPutColor(), MbufFree()**

MbufDiskInquire

Synopsis Inquire about the buffer data in a file.

Format `long MbufDiskInquire(FileName, InquireType, UserVarPtr)`

MIL_TEXT_PTR FileName;	File name
long InquireType;	Type of information about which to inquire
void *UserVarPtr;	Storage location for inquiry result

Description This function inquires about the buffer data in the specified file on disk.

The **FileName** parameter specifies the file name. Note, an error occurs if the file does not have a known file format or the file format is not supported.

The supported file types include all the formats supported by the **MbufExport()** and **MbufExportSequence()** functions. Since a "RAW" data file does not have any information regarding size or type, you can only use **MbufDiskInquire()** to determine the file format of this type of file.

The **InquireType** parameter specifies the parameter about which to inquire. This parameter can be set to one of the following values:

InquireType	Description
M_SIZE_X	Width of the data in the file.
M_SIZE_X+M_LUT	Width of the LUT associated with the image in the file. When there is no LUT associated with the image, returns M_INVALID.
M_SIZE_Y	Height of the data in the file.
M_SIZE_BAND	Number of color bands in the file.
M_SIZE_BAND+M_LUT	Number of bands of the LUT associated with the image in the file. When there is no LUT associated with the image, returns M_INVALID.
M_TYPE	File data type and depth (size in bits + M_SIGNED, M_UNSIGNED or M_FLOAT).
M_SIZE_BIT	File data depth in bits.
M_SIGN	File data range (M_SIGNED or M_UNSIGNED).
M_ATTRIBUTE	File attribute.
M_FILE_FORMAT	MIL identifier (MIL_ID) of the file format. See MbufExport() and MbufExportSequence() for all supported file formats.

InquireType	Description
M_LUT_PRESENT	Presence of LUT data in the file. (M_YES or M_NO)
M_ASPECT_RATIO	Aspect ratio of the image in the file. (default is 1:1)
M_NUMBER_OF_IMAGES	Number of images in an *.avi file.
M_FRAME_RATE	Frame rate (number of images/second) of an *.avi file.
M_COMPRESSION_TYPE	Returns the compression type of the image in the file. Returns M_NULL if the image is not compressed (for example, in a BMP file format). See MbufAllocColor() for all possible compression formats.
For M_KERNEL and M_STRUCT_ELEMENT data buffers only (see MbufControlNeighborhood() for possible values)	
M_OVERSCAN	Overscan type. (M_TRANSPARENT, M_REPLACE, M_MIRROR or M_DISABLE)
M_OVERSCAN_REPLACE_VALUE	Overscan replace value.
M_OFFSET_CENTER_X	Offset center X coordinate.
M_OFFSET_CENTER_Y	Offset center Y coordinate.
For M_KERNEL data buffers only (see MbufControlNeighborhood() for possible values)	
M_ABSOLUTE_VALUE	Absolute value flag.
M_SATURATION	Saturation flag.
M_NORMALIZATION_FACTOR	Normalization factor.

The **UserVarPtr** parameter specifies the address of the variable in which the requested information is to be written. Since the **MbufDiskInquire()** function also returns the requested information, you can set this parameter to M_NULL.

The **UserVarPtr** parameter should be a pointer to a long. Certain exceptions apply when **InquireType** is set to one of the following:

- When M_FILE_FORMAT is specified, this parameter should be a pointer to a MIL_ID.
- When M_ASPECT_RATIO or M_FRAME_RATE is specified, this parameter should be a pointer to a double value.

Return value The returned value is the value that represents the setting for the requested information, cast to long. If the requested information is not available, M_INVALID is returned.

See also **MbufLoad()**, **MbufImport()**

MbufExport

Synopsis Export a data buffer to a file.

Format **void MbufExport(FileName, FileFormat, SrcBufId)**

MIL_TEXT_PTR FileName;	Destination file name
long FileFormat;	File format
MIL_ID SrcBufId;	Source data buffer identifier

Description This function exports a data buffer to a file, using the specified output file format.

Note, you can also save a buffer in an M_MIL file format, using **MbufSave()**. The M_MIL file format is TIFF compatible.

To export an image with a LUT (color palette), associate the LUT to the image, using **MbufControl()**. Upon export, the image is saved with its associated color palette (MIM, TIFF and BMP file formats).

If you are exporting uncompressed data to a file with an M_JPEG_xx or M_JPEG2000_xx file attribute (the **FileFormat**), this function will automatically compress the data, according to the file format. The buffer does not need an M_COMPRESS attribute. If you are exporting compressed data to an uncompressed file format, this function will automatically decompress the data.

The **FileName** parameter specifies the name of the file in which to store the data buffer. If the file already exists, it will be overwritten.

The **FileFormat** parameter specifies the file conversion format. This parameter can be set to one of the following:

FileFormat	Description
M_MIL	Save the buffer contents in MIL file format (a regular TIFF 6.0 file format with extra information included in the comment field. It uses TIFF "chunky" mode to save color images.)
M_TIFF	Save the buffer contents in TIFF file format (only available for image buffers and saved in "chunky" mode for color images). The TIFF file format that is used respects the TIFF 6.0 specification.

FileFormat	Description
M_BMP	Save the buffer contents in BMP file format. The BMP file format that is used is the standard Windows format.
M_JPEG2000_LOSSLESS	Save the buffer contents in a JPEG 2000 lossless file format. If the buffer is 3-band and does not have an M_COMPRESS attribute, the data will be stored in RGB format.
M_JPEG2000_LOSSY	Save the buffer contents in a JPEG 2000 lossy file format. If the buffer is 3-band and does not have an M_COMPRESS attribute, the data will be stored in YUV16 planar format.
M_JPEG_LOSSLESS	Save the buffer contents in a JPEG lossless file format. If the buffer is 3-band and does not have an M_COMPRESS attribute, the data will be stored in RGB format.
M_JPEG_LOSSY	Save the buffer contents in a JPEG lossy file format. If the buffer is 3-band and does not have an M_COMPRESS attribute, the data will be stored in YUV16 packed format; otherwise, it will be stored in the same color format as the buffer.
M_JPEG_LOSSLESS_INTERLACED	Save an interlaced JPEG lossless image, to a file in the same compression format. If the buffer is 3-band, the buffer will be stored in RGB format.
M_JPEG_LOSSY_INTERLACED	Save an interlaced JPEG lossy image, to a file in the same compression format. If the buffer is 3-band, the data will always be stored in YUV16 packed format.
M_JPEG_LOSSY_RGB	Save a 3-band buffer in a JPEG lossy file format and store the data in RGB format. This attribute is only applicable to uncompressed image buffers.
M_RAW	Save the buffer contents in raw file format. The contents are dumped directly (byte stream) into the file and no header is added. If the buffer is multi-band, all bands are dumped one after the other.

Note that, except for the M_MIL and M_RAW file formats, the source buffer must have an M_IMAGE attribute.

If you are saving a non 8-bit buffer in M_BMP, M_JPEG_LOSSY, M_JPEG_LOSSY_RGB, or M_JPEG_LOSSY_INTERLACED format, only the 8 least-significant bits are saved. This is because these formats are restricted to 8 bits per band. If you are saving a non 8-bit or a non 16-bit buffer in the

M_JPEG_LOSSLESS, M_JPEG_LOSSLESS_INTERLACED, M_JPEG2000_LOSSY, or M_JPEG2000_LOSSLESS format, only the 8-least significant or 16-least significant bits, respectively, are saved.

By default, most color buffers are saved in packed (chunky) format (in accordance with TIFF 6.0 specifications). Color binary buffers are saved in a 1-bit per pixel format (data is stored in a 3-band, packed binary format). When a color buffer is saved in a raw file format, its bands are saved in a planar format (one band after another). Note, however, that to save a color image in a planar, rather than a packed mode, M_PLANAR can be added to the M_MIL or M_TIFF file formats, (for example, M_TIFF+M_PLANAR).

The **SrcBufId** parameter specifies the identifier of the data buffer to save.

Note This function is optimized for packed binary buffers.

MIL-Lite does not support JPEG 2000 compression, and requires dedicated hardware for JPEG compression. This is not a restriction under MIL.

See also **MbufImport()**, **MbufSave()**, **MbufLoad()**, **MbufRestore()**, **MbufControl()**

MbufExportSequence

Synopsis Export a sequence of image buffers to an .avi file.

Format `void MbufExportSequence(FileName, FileFormat, BufArrayPtr, NumberOfImages, FrameRate, ControlFlag)`

MIL_TEXT_PTR FileName;	File name
long FileFormat;	File format
MIL_ID *BufArrayPtr;	Array of image buffer identifiers
long NumberOfImages;	Number of image buffers
double FrameRate;	Frame rate
long ControlFlag;	Control flag

Description This function exports a sequence of image buffers to an audio video interleave (*.avi) file.

The **FileName** parameter specifies the name of the file in which to export the image buffers.

The **FileFormat** parameter specifies the format of the file. This parameter can be set to one of the following:

M_AVI_MJPEG	A standard AVI format used to hold JPEG compressed sequences. When this format is specified, the image buffers will be in YUV16 packed format. In addition, image buffers will have a width that is a multiple of 16 pixels. For image buffers that have the M_JPEG_LOSSY compression type, the height will be a multiple of 8, and less than or equal to 240 pixels. For image buffers that have the M_JPEG_LOSSY_INTERLACED compression type, the height will be a multiple of 16 pixels, greater than 240 pixels. If the image buffers are not already in this format, MIL will automatically convert them appropriately. This type of sequence requires a codec to be supported by Windows Media Player.
M_AVI_DIB	An AVI format used to hold non-compressed DIB image buffers. If necessary, the image buffers will be converted to a non-compressed DIB format before exporting. This type of sequence is supported by Windows Media Player.
M_AVI_MIL	An AVI format used to hold image buffers in their MIL format. This saves images in the format in which they are sent to this function. Since the images are saved "as is", no loss is introduced in the images. This type of sequence requires a codec to be supported by Windows Media Player.
M_DEFAULT	MIL automatically decides the appropriate format.

The **BufArrayPtr** parameter specifies the address of the array containing the MIL identifiers of the image buffers to export.

The **NumberOfImages** parameter specifies the number of image buffers to export. If the supplied array is larger than this number, the remaining buffer identifiers are ignored.

The **FrameRate** parameter specifies the frame rate (number of image buffers/second) of the sequence.

The **ControlFlag** parameter specifies whether to append the image buffers to the *.avi file, if the file already exists, or overwrite the file. This parameter can be set to one of the following:

M_DEFAULT	Overwrite the file. The file will be opened, written into, and then the file will be closed.
M_OPEN	Open the AVI file for writing, and set the pointer to the beginning of the file. If M_OPEN+M_APPEND is specified, the file is opened and the file pointer is set to the end of the file. BufArrayPtr , NumberOfImages , and FrameRate should be set to M_NULL.
M_WRITE	Write the specified number of images in the files starting from the current file pointer position. After the write operation, the file pointer is left at the end of the file, ready for the next M_WRITE operation. BufArrayPtr , NumberOfImages , and FrameRate should be set to the appropriate values.
M_CLOSE	Close the AVI file. BufArrayPtr , NumberOfImages , and FrameRate should be set to M_NULL.
M_APPEND	Append the image buffers to the file. The file will be opened, the specified images will be appended, and then the file will be closed.

Note MIL-Lite does not support JPEG 2000 compression, and requires dedicated hardware for JPEG compression. This is not a restriction under MIL.

See also **MbufImportSequence()**

MbufFree

Synopsis Free a data buffer.

Format **void MbufFree(BufId)**

MIL_ID BufId;	Buffer identifier to deallocate
---------------	---------------------------------

Description This function deallocates a previously allocated data buffer. The memory reserved for the specified buffer is released.

Child buffers associated to a parent buffer must be deallocated, using **MbufFree()**, prior to deallocating the parent buffer.

The **BufId** parameter specifies the identifier of the data buffer to deallocate.

See also **MbufAlloc1d()**, **MbufAlloc2d()**, **MbufAllocColor()**, **MbufChild1d()**, **MbufChild2d()**, **MbufChildColor()**

MbufGet

Synopsis Get data from a buffer and place it in a user-supplied array.

Format **void MbufGet(SrcBufId, UserArrayPtr)**

MIL_ID SrcBufId;	Source buffer identifier
void *UserArrayPtr;	Destination user array

Description This function copies data from a specified MIL source buffer to a user-supplied array.

The **SrcBufId** parameter specifies the identifier of the source buffer.

The **UserArrayPtr** parameter specifies the address of the user array in which to copy source buffer data. Ensure that the user array is large enough to accommodate the data from the source buffer. **MbufGet()** assumes that the array is of the same data type and depth as the source buffer's bands.

Note, for multi-band buffers, **MbufGet()** behaves like **MbufGetColor(SrcBufId, M_PLANAR, M_ALL_BAND, UserArrayPtr)**. Refer to **MbufGetColor()** for more details.

Note This function is optimized for packed binary buffers.

See also **MbufGet1d()**, **MbufGet2d()**, **MbufGetColor()**, **MbufPut()**, **MbufPut1d()**, **MbufPut2d()**, **MbufPutColor()**

MbufGetColor

Synopsis Get data from one or all bands of a buffer and place it in a user-supplied array.

Format **void MbufGetColor(SrcBufId, DataFormat, Band, UserArrayPtr)**

MIL_ID SrcBufId;	Source buffer identifier
long DataFormat;	Data format of the user array
long Band;	Color band of source buffer
void *UserArrayPtr;	Destination user array

Description This function copies data from one or all color bands of a specified MIL source buffer to a user-supplied array.

The **SrcBufId** parameter specifies the identifier of the source buffer. The internal data format of the source buffer need not match the specified data format of the user-supplied array; an internal conversion will be performed if necessary. Note, however, if the formats do match the operation will be much faster.

The **DataFormat** parameter specifies the data format to use to save the data in the user array. Note that Sx and Sy denote the source width and height, respectively. This parameter must be set to one of the following values:

DataFormat	Description
M_SINGLE_BAND	Copy a single color band. The user array must be of the same type as the source buffer and have a size of Sx x Sy.
M_BGR24+M_PACKED	Copy three bands in an interleaved manner (BGRBGR). The source buffer must be a three-band, 8-bit buffer and the user array must have a size of Sx x Sy x 3 bytes (Sx x Sy x 3char).
M_BGR32+M_PACKED	Copy three bands in an interleaved manner (BGRXBGRX). The source buffer must be a three-band, 8-bit buffer and the user array must have a size of Sx x Sy x 4 bytes (Sx x Sy x long).
M_RGB15+M_PACKED	Copy three bands in an interleaved manner (RGB 5:5:5). The source buffer must be a three-band, 8-bit buffer and the user array must have a size of Sx x Sy x 2 bytes (Sx x Sy x 2 unsigned char).

DataFormat	Description
M_RGB16+M_PACKED	Copy three bands in an interleaved manner (RGB 5:6:5). The source buffer must be a three-band, 8-bit buffer and the user array must have a size of $S_x \times S_y \times 2$ bytes ($S_x \times S_y \times 2$ unsigned char).
M_PLANAR	Copy the bands one after the other (RRR...GGG...BBB...). The user array must be the same data type as the source buffer and have a size of $S_x \times S_y \times$ number of color bands of the source buffer, where S_x and S_y denote the source width and height, respectively. This format is to be used when copying from all color bands of the source buffer.

To interpret the array data as top-down (DIB), add M_FLIP to the **DataFormat** parameter. Note that for the M_PLANAR and M_SINGLE_BAND data formats, the M_FLIP flag is not supported.

The **Band** parameter specifies the index of the color band to copy. This parameter can be set to any index from 0 to $n-1$ (number of bands of the source buffer - 1), or to one of the following values:

M_RED	Copy from the red color band.
M_GREEN	Copy from the green color band.
M_BLUE	Copy from the blue color band.
M_ALL_BAND	Copy from all color bands.

If the source buffer is in a HLS (hue, luminance, and saturation) format, the band can be set to: M_HUE, M_LUMINANCE, M_SATURATION, or M_ALL_BAND.

The **UserArrayPtr** parameter specifies the address of the user array in which to copy data from the source buffer. Ensure that the user array is large enough to accommodate the data from the source buffer in the format specified.

Note This function is optimized for packed binary buffers.

See also MbufGet(), MbufGet1d(), MbufGet2d(), MbufPut(), MbufPut1d(), MbufPut2d(), MbufPutColor()

MbufGetColor2d

Synopsis Get data from a region of one or all bands of a buffer and place it in a user-supplied array.

Format **void MbufGetColor2d(SrcBufId, DataFormat, Band, OffX, OffY, SizeX, SizeY, UserArrayPtr)**

MIL_ID SrcBufId;	Source buffer identifier
long DataFormat;	Data format of the user array
long Band;	Color band of source buffer
long OffX;	X pixel offset relative to the source buffer
long OffY;	Y pixel offset relative to the source buffer
long SizeX;	Source buffer region width
long SizeY;	Source buffer region height
void *UserArrayPtr;	Destination user array

Description This function copies data from a specific region of one or all color bands of a specified MIL source buffer to a user-supplied array.

The **SrcBufId** parameter specifies the identifier of the source buffer. The internal data format of the source buffer need not match the specified data format of the user-supplied array; an internal conversion will be performed if necessary. Note however, if the formats do match the operation will be much faster.

The **DataFormat** parameter specifies the data format to use to save the data in the user array. Note that Sx and Sy denote the source width and height, respectively. This parameter must be set to one of the following values:

DataFormat	Description
M_SINGLE_BAND	Copy a single color band. The user array must be of the same type as the source buffer and have a size of Sx x Sy.
M_BGR24+M_PACKED	Copy three bands in an interleaved manner (BGRBGR). The source buffer must be a three-band, 8-bit buffer and the user array must have a size of Sx x Sy x 3 bytes (Sx x Sy x 3char).
M_BGR32+M_PACKED	Copy three bands in an interleaved manner (BGRXBGRX). The source buffer must be a three-band, 8-bit buffer and the user array must have a size of Sx x Sy x 4 bytes (Sx x Sy x long).

DataFormat	Description
M_RGB15+M_PACKED	Copy three bands in an interleaved manner (RGB 5:5:5). The source buffer must be a three-band, 8-bit buffer and the user array must have a size of $S_x \times S_y \times 2$ bytes ($S_x \times S_y \times 2$ unsigned char).
M_RGB16+M_PACKED	Copy three bands in an interleaved manner (RGB 5:6:5). The source buffer must be a three-band, 8-bit buffer and the user array must have a size of $S_x \times S_y \times 2$ bytes ($S_x \times S_y \times 2$ unsigned char).
M_PLANAR	Copy the bands one after the other (RRR...GGG...BBB...). The user array must be the same type as the source buffer and have a size of $S_x \times S_y \times$ number of color band of the source buffer. This format is to be used when copying all color bands of the source buffer.

To interpret the array data as top-down (DIB), add **M_FLIP** to the **DataFormat** parameter. Note that for the **M_PLANAR** and **M_SINGLE_BAND** data formats, the **M_FLIP** flag is not supported.

The **Band** parameter specifies the index of the color band to copy. This parameter can be set to any index from 0 to $n-1$ (number of bands of the source buffer - 1), or to one of the following values:

M_RED	Copy from the red color band.
M_GREEN	Copy from the green color band.
M_BLUE	Copy from the blue color band.
M_ALL_BAND	Copy from all color bands.

If the source buffer is in a HLS (hue, luminance, and saturation) format, the band can be set to: **M_HUE**, **M_LUMINANCE**, **M_SATURATION**, or **M_ALL_BAND**.

The **OffX** and **OffY** parameters specify the horizontal and vertical pixel offsets (relative to the top-left source buffer coordinate) of the source buffer region in which to get the data.

The **SizeX** and **SizeY** parameters specify the width and height of the source buffer region in which to get the data.

The **UserArrayPtr** parameter specifies the address of the user array in which to copy the data. Ensure that there are enough entries in the user array to receive the data of the specified source buffer region.

See also **MbufGet()**, **MbufGet1d()**, **MbufGet2d()**, **MbufPut()**, **MbufPut1d()**, **MbufPut2d()**, **MbufPutColor()**, **MbufPutColor2d()**

MbufGetHookInfo

Synopsis Get information about a hook event.

Format **long MbufGetHookInfo(BufferId, EventId, InquireType, UserVarPtr)**

MIL_ID BufferId;	Buffer identifier
MIL_ID EventId;	Identifier received by the hook-handler function
long InquireType;	Type of information which is inquired
void MPTYPE *UserVarPtr;	Storage location for requested information

Description This function allows you to get information about the event that caused the hook function to be called. **MbufGetHookInfo()** should only be called within the scope of a buffer hook-handler function (see **MbufHookFunction()**).

The **BufferId** parameter specifies the identifier of the buffer.

The **EventId** parameter is the buffer event identifier received by the hook-handler function (see **MbufHookFunction()**).

The **InquireType** parameter specifies the type of information about which to inquire. If the hook-handler function was called with a **HookType** parameter equal to M_MODIFIED_BUFFER, the supported value for **InquireType** is:

InquireType	Description
M_MODIFIED_BUFFER + M_BUFFER_ID	Writes the MIL ID of the modified buffer in the storage location pointed to by UserVarPtr.
M_MODIFIED_BUFFER + M_REGION_OFFSET_X	Writes the X-offset of the modified region (of the buffer) in the storage location pointed to by UserVarPtr.
M_MODIFIED_BUFFER + M_REGION_OFFSET_Y	Writes the Y-offset of the modified region (of the buffer) in the storage location pointed to by UserVarPtr.
M_MODIFIED_BUFFER + M_REGION_SIZE_X	Writes the width of the modified region (of the buffer) in the storage location pointed to by UserVarPtr.
M_MODIFIED_BUFFER + M_REGION_SIZE_Y	Writes the height of the modified region (of the buffer) in the storage location pointed to by UserVarPtr.

The **UserVarPtr** parameter specifies the address of the variable in which to write the requested information.

Return value Returns null (M_NULL) on success, and returns a non-null (!M_NULL) value on failure, without logging any errors in the application.

See also **MbufHookFunction()**

MbufGetLine

Synopsis Read the pixels along a specified theoretical line, count the pixels, and store them in a user-defined array.

Format **void MbufGetLine(ImageBufId, StartX, StartY, EndX, EndY, Mode, NbPixelsPtr, UserArrayPtr)**

MIL_ID ImageBufId;	Image buffer identifier
long StartX;	X start position of the line
long StartY;	Y start position of the line
long EndX;	X end position of the line
long EndY;	Y end position of the line
long Mode;	Operation mode
long *NbPixelsPtr;	Number of pixels
void *UserArrayPtr;	Destination user array

Description This function reads the series of pixels between specified coordinates (theoretical line) in a specified source image and stores the pixels in a user-defined array. The Bresenham algorithm is used to determine the theoretical line.

The **ImageBufId** parameter specifies the identifier of the source image buffer. This must be a single-band (monochrome) buffer.

The **StartX** and **StartY** parameters specify the horizontal and vertical pixel offsets of the starting position of the line, relative to the top-left pixel of the source buffer.

The **EndX** and **EndY** parameters specify the horizontal and vertical pixel offsets of the finishing position of the line, relative to the top-left pixel of the source buffer.

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

The **NbPixelsPtr** parameter specifies the address of the variable in which to write the number of pixels found along the theoretical line. You can set this parameter to M_NULL if you don't want this value to be evaluated.

The **UserArrayPtr** parameter specifies the address of the user array in which to store the pixels from the image buffer. **MbufGetLine()** assumes that the array is of the same data type as the source buffer. Ensure that the user array is large enough to accommodate the data to be stored. To determine the required size of the array, you can set this parameter to **M_NULL** and pass a non-null address to **NbPixelsPtr**. In this case, nothing is read from the image buffer.

See also **MbufPutLine()**

MbufGet1d

Synopsis Get data from a 1D area of a buffer and place it in a user-supplied array.

Format **void MbufGet1d(SrcBufId, OffX, SizeX, UserArrayPtr)**

MIL_ID SrcBufId;	Source buffer identifier
long OffX;	X offset relative to source buffer origin
long SizeX;	Width of source buffer area from which to get data
void *UserArrayPtr;	Destination user array

Description This function copies data from a specified one-dimensional area of a MIL source buffer to a user-supplied array.

Note, for multi-band buffers, this function linearly copies the data from the one-dimensional region of each band (RRR...GGG...BBB...).

The **SrcBufId** parameter specifies the identifier of the source buffer.

The **OffX** parameter specifies the horizontal offset (in pixels) of the required area, relative to the top-left pixel of the source buffer.

The **SizeX** parameter specifies the width of the required area of the source buffer.

The **UserArrayPtr** parameter specifies the address of the user array in which to copy the data from the source buffer. Ensure that the user array is large enough to accommodate the data to be copied from the source buffer. **MbufGet1d()** assumes that the array is of the same data type as the source buffer.

See also **MbufGet()**, **MbufGet2d()**, **MbufGetColor()**, **MbufPut()**, **MbufPut1d()**, **MbufPut2d()**, **MbufPutColor()**

MbufGet2d

Synopsis Get data from a 2d area of a buffer and place it in a user-supplied array.

Format **void MbufGet2d(SrcBufId, OffX, OffY, SizeX, SizeY, UserArrayPtr)**

MIL_ID SrcBufId;	Source buffer identifier
long OffX;	X pixel offset relative to source buffer region
long OffY;	Y pixel offset relative to source buffer region
long SizeX;	Width of required data area
long SizeY;	Height of required data area
void *UserArrayPtr;	Source user array

Description This function copies data from a specified two-dimensional region of a MIL source buffer to a user-supplied array.

Note, for multi-band buffers, this function linearly copies the data from the specified two-dimensional region of each band (RRR...GGG...BBB...).

The **SrcBufId** parameter specifies the identifier of the source buffer.

The **OffX** parameter specifies the horizontal offset (in pixels) of the required area, relative to the top-left pixel of the source buffer. The **OffY** parameter specifies the vertical offset.

The **SizeX** and **SizeY** parameters specify the width and height of the required area of the source buffer.

The **UserArrayPtr** parameter specifies the address of the user array in which to copy the data from the source buffer. Ensure that the user array is large enough to accommodate the data to be copied. **MbufGet2d()** assumes that the array is of the same data type as the source buffer.

See also **MbufGet(), MbufGet1d(), MbufGetColor(), MbufPut(), MbufPut1d(), MbufPut2d(), MbufPutColor()**

MbufHookFunction

Synopsis Hook a function to a buffer event.

Format **void MbufHookFunction(BufferId, HookType, HookHandlerPtr, UserDataPtr)**

MIL_ID BufferId;	Buffer identifier
long HookType;	Type of event to hook
MBUFHOOKFCTPTR HookHandlerPtr;	Pointer to the function to call when the specified buffer event occurs
void MPTYPE *UserDataPtr;	User data pointer

Description This function allows you to attach or detach a user-defined function to a specified buffer event (for example, a modification to the buffer's contents). Once a hook-handler function is defined and hooked to an event, it is automatically called when the event occurs.

You can hook more than one function to an event by making separate calls to **MbufHookFunction()** for each function that you want to hook. MIL automatically chains and keeps an internal list of all these hooked functions. When a function is hooked, this new function is added to the end of the list. When the event happens, all user-defined functions in the list will be executed in the same order that they were hooked to the event. You can also remove any function from the list; in this case, MIL preserves the order of the remaining functions in the list.

The **BufferId** parameter specifies the identifier of the buffer.

The **HookType** parameter specifies the buffer event to which to hook the function. This parameter can be set to one of the following values. Note that a hooked function must be unhooked by combining the **HookType** parameter with **M_UNHOOK**.

HookType	Description
M_MODIFIED_BUFFER	Calls the hook-handler function each time the specified buffer is modified by a MIL function.
M_UNHOOK + M_MODIFIED_BUFFER	Unhooks the specified function if hooked to an M_MODIFIED_BUFFER event.

The **HookHandlerPtr** parameter specifies the address of the function that should be called when the specified event occurs. The hook-handler function must be declared as follows:

long MFTYPE HookHandler(HookType, EventId, UserDataPtr);	
long HookType;	Type of buffer event that generated the call
MIL_ID EventId;	Event identifier to pass to MbufGetHookInfo() when inquiring about the hooked event
void MPTYPE *UserDataPtr;	User data pointer that was passed (as UserDataPtr) by MbufHookFunction()

Upon successful completion, the hook-handler function should return M_NULL. Note, MBUFHOOKFCTPTR, MFTYPE, and MPTYPE are reserved MIL predefined types for functions and data pointers.

The **UserDataPtr** parameter specifies the address of the user data that you want to make available to the hook-handler function. This address is passed to the hook-handler function, through its **UserDataPtr** parameter, when the specified event occurs. Set this parameter to M_NULL if not used.

See also **MbufGetHookInfo()**

MbufImport

Synopsis Import data from a file into a data buffer.

Format **MIL_ID MbufImport(FileName, FileFormat, Operation, SystemId, BufIdPtr)**

MIL_TEXT_PTR FileName;	Source file name
long FileFormat;	File format
long Operation;	Import operation
MIL_ID SystemId;	System identifier
MIL_ID *BufIdPtr;	Buffer identifier (returned or given)

Description This function imports data, of the specified format, from a file into a MIL data buffer. The buffer can be an existing data buffer, or an automatically allocated buffer.

Note, you can also import data using **MbufLoad()** or **MbufRestore()**; however, these functions try to determine the format from the data rather than allowing you to specify the data type.

If you are importing uncompressed data into a buffer with an M_COMPRESS attribute, this function will automatically compress it, according to the compression settings found in the buffer. If you are importing compressed data into a buffer with an M_IMAGE attribute (but not an M_COMPRESS attribute), this function will automatically decompress it. If necessary, the data in the file will be transformed to fit into the buffer. If you are not sure what type of compressed data the file contains, use M_DEFAULT as the file format rather than M_JPEG_xx or M_JPEG2000_xx; the data will be read correctly.

When a buffer is automatically allocated during a restore operation, it is allocated with the same attributes as the original buffer, with the exception of M_IMAGE buffers. In the case of an M_IMAGE type buffer, the **MbufImport()** function tries to allocate an image buffer so that it can be used for acquisition (M_GRAB), display (M_DISP), and processing (M_PROC) operations. If there is insufficient appropriate memory to allocate such a buffer, it allocates one that can be used in all of the above operations except for acquisition (M_GRAB). Note that the maximum (total) number of grab (M_GRAB) buffers that can be allocated is restricted by the total

amount of DMA memory that was specified at the time of installation. For systems with on-board processors, the total number of M_GRAB buffers and M_PROC buffers is limited by the amount of on-board memory.

When importing a compressed file into an automatically allocated buffer, the buffer will have an M_COMPRESS attribute.

When importing an image file that has been saved with an associated LUT (color palette), the LUT is also imported and associated with the resulting image buffer. You can obtain the identifier of the associated LUT, using **MbufInquire()**.

Similarly, when loading a monochrome image file that has been saved with an associated LUT (color palette) into a single-band buffer, the LUT is also imported and associated with the resulting image buffer.

❖ Note that the associated LUT will be automatically selected on the display (**MdispLut()**) if the image buffer is selected on a display and the default LUT has not been overridden by a former call to **MdispLut()**.

When loading an image file that has been saved with an associated LUT (color palette) into a 3-band 8-bit image buffer, the LUT is automatically applied to the data to generate 3-band image data. In this case, a LUT buffer is not created and, therefore, is not associated to the 3-band 8-bit buffer.

Using **MbufDiskInquire()**, you can inquire about the dimensions of a buffer saved in a file (except for RAW files) without importing it.

After restoring a buffer, we recommend that you check if the operation was successful, by using **MappGetError()**, or by verifying that the returned buffer identifier is not M_NULL.

The **FileName** parameter specifies the name of the file from which to get the data.

The **FileFormat** parameter specifies the file conversion format. This parameter can be set to one of the following:

M_MIL	Import data that is in MIL file format.
M_TIFF	Import data that is in TIFF file format (only available for image buffers). The TIFF 6.0 specification is used.
M_BMP	Import data that is in BMP file format (only available for image buffers). The standard Windows BMP format is used.
M_RAW	Import data that is in RAW file format.

M_JPEG2000_LOSSLESS	Import a JPEG 2000 lossless image.
M_JPEG2000_LOSSY	Import a JPEG 2000 lossy image.
M_JPEG_LOSSLESS	Import a JPEG lossless image.
M_JPEG_LOSSY	Import a JPEG lossy image.
M_JPEG_LOSSLESS_INTERLACED	Import a JPEG lossless image stored in two separate fields. If the buffer is 3-band, the buffer will be stored in RGB format. Only available for image buffers.
M_JPEG_LOSSY_INTERLACED	Import a JPEG lossy image stored in two separate fields. If the buffer is 3-band, the data will always be stored in YUV16 packed format. Only available for image buffers.
M_JPEG_LOSSY_RGB	Import a 3-band JPEG lossy image that is in RGB format.
M_DEFAULT	Automatically determine the file format. If the file format is not supported, its data will be treated in RAW file format.

The **Operation** parameter specifies the import operation. This parameter can be set to one of the following:

M_RESTORE	Data from the specified file is imported into an automatically allocated MIL data buffer .
M_LOAD	Data from the specified file is imported into a previously allocated MIL data buffer .

Note, you cannot restore (M_RESTORE) a RAW data file (M_RAW) because its dimensions are unknown.

The **SystemId** parameter specifies the system on which the MIL buffer will be allocated, if M_RESTORE is specified as the operation. This parameter must be given a valid system identifier or it can be set to M_DEFAULT_HOST. In the latter case, the default Host system of the current MIL application is used. You can also specify M_DEFAULT, in which case MIL selects the most appropriate system on which to allocate the buffer (either the Host system or any currently allocated system).

Set **SystemId** to M_NULL if M_LOAD is specified as the operation.

The **BufIdPtr** parameter specifies the address of the variable that either gives or receives a data buffer identifier, depending on the setting of the **Operation** parameter. When **Operation** is set to M_RESTORE, **MbufImport()** returns the buffer identifier and stores it at the

variable's specified address. Since **MbufImport()** also returns the buffer identifier, you can set this parameter to M_NULL. If allocation fails, M_NULL is written as the identifier.

When a buffer identifier is given, the buffer must be large enough in depth and dimensions to hold the data; if not, some data is clipped. For example, if the data is deeper than the buffer, the most-significant bits of the data are not written. If, however, the buffer is larger in depth or dimensions than the data, excess areas are unaffected.

Note MIL-Lite does not support JPEG 2000 compression, and requires dedicated hardware for JPEG compression. This is not a restriction under MIL.

Return value The returned value is the buffer identifier (for an M_RESTORE operation only). If allocation fails, M_NULL is returned.

Status This function supports the baseline TIFF 6.0 format for grayscale and RGB images.

See also **MbufDiskInquire(), MbufExport(), MbufSave(), MbufLoad(), MbufRestore(), MbufControl()**

MbufImportSequence

Synopsis Import a sequence of images from an *.avi file into separate image buffers.

Format **void MbufImportSequence(FileName, FileFormat, Operation, SystemId, BufArrayPtr, StartImage, NumberOfImages, ControlFlag)**

MIL_TEXT_PTR FileName;	File name
long FileFormat;	File format
long Operation;	Operation mode
MIL_ID SystemId;	Target system
MIL_ID *BufArrayPtr;	Array of image buffer identifiers
long StartImage;	Start image
long NumberOfImages;	Number of image buffers
long ControlFlag;	Control flag

Description This function imports a sequence of images from an *.avi file into separate image buffers. **MbufImportSequence()** can automatically allocate the necessary buffers or you can use previously allocated buffers. In the latter case, the **BufArrayPtr** parameter should point to an array containing the buffer identifiers. In the former case, **MbufImportSequence()** will write the identifiers of the new buffers into the array pointed to by **BufArrayPtr**.

The **FileName** parameter specifies the name of the file.

The **FileFormat** parameter specifies the format of the file. This parameter can be set to one of the following:

M_AVI_MJPG	An AVI format containing compressed images.
M_AVI_DIB	An AVI format containing non-compressed images.
M_AVI_MIL	An AVI format containing images in their MIL format.
M_DEFAULT	MIL automatically determines the file format.

The **Operation** parameter specifies whether to import the sequence into automatically allocated buffers or previously allocated buffers. This parameter can be set to one of the following:

M_LOAD	Import the sequence into previously allocated buffers.
M_RESTORE	Import the sequence into automatically allocated buffers.

The **SystemId** parameter specifies the system on which to allocate the buffers for an M_RESTORE operation. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to allocate the buffer (it can be the Host system or any already allocated system).

For an M_LOAD operation, set the **SystemId** parameter to M_NULL.

The **BufArrayPtr** parameter specifies the address of the array containing the buffer identifiers (for an M_LOAD operation) or the address of the array in which to store the new buffer identifiers (for an M_RESTORE operation).

For an M_LOAD operation, the destination buffers should be large enough to hold the imported images. If you are importing compressed images into buffers with only an M_IMAGE specifier, the images will be automatically decompressed. If you are importing decompressed images into buffers with an M_IMAGE+M_COMPRESS specifier, the images will be automatically compressed.

For an M_RESTORE operation, the destination buffers will be allocated with an appropriate size and type to hold the images. For example, if you are importing compressed images, the destination buffers will have an M_IMAGE+M_COMPRESS specifier. If an M_RESTORE operation fails, zero will be written for the buffer identifiers.

The **StartImage** parameter specifies the first image in the sequence to import. Images start at 0.

The **NumberOfImages** parameter specifies the number of images, starting at **StartImage**, to import. The array pointed to by **BufArrayPtr** should be at least as big as this number. Note that you can inquire about the number of images in an *.avi file using **MbufDiskInquire()**.

The **ControlFlag** parameter specifies the function's control flag. This parameter must be set to one of the following:

ControlFlag	Description
M_DEFAULT	Open the AVI file, read the specified images, and then close the file.
M_OPEN	Open the AVI file for reading, and set the pointer to the first image. BufArrayPtr , NumberOfImages , and StartImage should be set to M_NULL.

ControlFlag	Description
M_READ	Read the specified images in the AVI file, starting at the specified StartImage position. To read the image at the current read position, set StartImage to M_DEFAULT. After the read operation, the file pointer is left at the position of the next image, ready for the next M_READ operation.
M_CLOSE	Close the AVI file after reading, and (re)set the pointer position to the first image. BufArrayPtr , NumberOfImages , and FrameRate should be set to M_NULL.

Note MIL-Lite does not support JPEG 2000 compression, and requires dedicated hardware for JPEG compression. This is not a restriction under MIL.

See also **MbufDiskInquire()**, **MbufExportSequence()**

MbufInquire

Synopsis Inquire about a data buffer parameter setting.

Format `long MbufInquire(BufId, InquireType, UserVarPtr)`

MIL_ID BufId;	Source buffer identifier
long InquireType;	Type of information about which to inquire
void *UserVarPtr;	Storage location for requested information

Description This function inquires about a specified MIL buffer parameter setting. This function is useful, for example, to check the size of a buffer restored from disk.

The **BufId** parameter specifies the identifier of the source buffer.

The **InquireType** parameter specifies the buffer parameter setting about which to inquire. This parameter can be set to one of the following values:

InquireType	Description
M_SIZE_X	Width of the buffer.
M_SIZE_Y	Height of the buffer.
M_SIZE_BAND	Number of buffer color bands.
M_SIZE_BIT	Depth per band, in bits.
M_SIZE_BYTE	Size of the buffer, in bytes. If the buffer is of type M_JPEG2000_LOSSY, this value might be smaller than the value of M_TARGET_SIZE if the data could be compressed into a size smaller than the target.
M_TYPE	Buffer data type and depth (size in bits + M_SIGNED, M_UNSIGNED, or M_FLOAT).
M_SIGN	Buffer range (M_SIGNED or M_UNSIGNED).
M_ATTRIBUTE	Buffer attribute.
M_OWNER_SYSTEM	Identifier of the system on which the buffer has been allocated.
M_OWNER_SYSTEM_TYPE	Type of system on which the buffer was allocated.

InquireType	Description
M_PITCH*	The number of pixels between the beginnings of any two adjacent lines of the buffer data.
M_PITCH_BYTE*	The number of bytes between the beginnings of any two adjacent lines of the buffer data.
*Note: when inquiring the pitch of an M_BGR24 + M_PACKED buffer, you should use M_PITCH_BYTE instead of M_PITCH because the latter might not be able to take into account internal padding.	
M_HOST_ADDRESS	Host pointer to the buffer or M_NULL. If a planar, 3-band buffer is being used, M_NULL will be returned. However, the Host address can be determined by allocating a child buffer for the required band and then using M_HOST_ADDRESS to determine its Host address. If available, this pointer can be used to directly access the data of a MIL buffer with the Host CPU.
M_PHYSICAL_ADDRESS	Physical address of the buffer or M_NULL. Available only for a non-paged buffer mapped to the Host. This type of buffer is used mostly used for access by bus masters other than the Host CPU.
M_PARENT_ID	Identifier of parent buffer. (returns same as BufId if no parent buffer)
M_PARENT_OFFSET_X	X offset relative to the parent buffer.
M_PARENT_OFFSET_Y	Y offset relative to the parent buffer.
M_PARENT_OFFSET_BAND	Band offset relative to the parent buffer.
M_ANCESTOR_ID	MIL identifier of the ancestor buffer (returns same as BufId if no ancestor buffer). An ancestor buffer is a buffer from which other buffers originated. It must have been allocated with MbufAlloc1d() , MbufAlloc2d() , or MbufAllocColor() and does not have a parent buffer.
M_ANCESTOR_OFFSET_X	X offset relative to the ancestor buffer.
M_ANCESTOR_OFFSET_Y	Y offset relative to the ancestor buffer.
M_ANCESTOR_OFFSET_BAND	Band offset relative to the ancestor buffer.
M_ANCESTOR_OFFSET_BIT	Bit offset relative to the ancestor buffer.

InquireType	Description
M_MODIFICATION_COUNT	<p>Returns the current value of the modification counter of the image buffer. The modification counter is initialized to a number that is unique to the image buffer and is given its own unique range. If the image buffer is freed, this number will not be reassigned to a new image buffer. This number is incremented by one each time the image buffer is modified.</p> <p>If the image buffer is accessed externally, for example, when using MbufCreateColor() or MbufCreate2d(), MbufControl() with M_MODIFIED must be called to indicate that the image buffer's contents have been modified. Calling this function will increment the counter. This feature is useful for optimization. For example, you can avoid repeating certain computations (for example, analysis computations) if you know that the image buffer has not been modified. In this case, inquire the count before the first computation in the sequence of computations, and then inquire it again before repeating the same sequence. If no modifications have been made to the image buffer, you can avoid repeating the sequence unnecessarily.</p>
M_ASSOCIATED_LUT	Identifier of the LUT buffer associated with the image buffer. (returns M_DEFAULT if no LUT)
M_NATIVE_ID	The native identifier (handle) of the buffer. This identifier can be used when operating in the system native library.
M_WINDOW_DDRAW_SURFACE	Pointer (LPDIRECTDRAWSURFACE) to the DirectDraw surface associated with the MIL buffer (if any) or M_NULL.
M_WINDOW_DIB_HEADER	Pointer (LPBITMAPINFO) to the header of the DIB associated with the MIL buffer (if any) or M_NULL.
M_WINDOW_DC	Windows display context handle (HDC) (MbufControl()) or M_NULL. This inquire type must be used with the MbufControl() M_WINDOW_DC_ALLOC control type.

InquireType	Description
M_FORMAT	This setting accesses information about the buffer format. See MbufAlloc...() for all possible return values. Note, it is also possible to extract the internal format of the buffer by adding the M_INTERNAL_FORMAT mask to the resulting M_FORMAT value.
For M_KERNEL and M_STRUCT_ELEMENT data buffers only (see MbufControlNeighborhood() for possible values):	
M_OVERSCAN	Overscan type.
M_OVERSCAN_REPLACE_VALUE	Overscan replace value.
M_OFFSET_CENTER_X	Offset center X coordinate.
M_OFFSET_CENTER_Y	Offset center Y coordinate.
For M_KERNEL data buffers only (see MbufControlNeighborhood() for possible values):	
M_ABSOLUTE_VALUE	Absolute value flag.
M_SATURATION	Saturation flag.
M_NORMALIZATION_FACTOR	Normalization factor.
For M_IMAGE+M_COMPRESS image buffers (see MbufAlloc...() for possible values):	
M_COMPRESSION_TYPE	Type of compression. See MbufAlloc...() for possible values.
M_SIZE_BYTE	Size of compressed buffer in bytes. The buffer size will be zero if the buffer has not been initialized with data.
For M_IMAGE+M_COMPRESS image buffers with compression type set to M_JPEG_LOSSY , M_JPEG_LOSSY_INTERLACED , or M_JPEG2000_LOSSY :	
M_Q_FACTOR	Quantization factor for both JPEG 2000 lossy and JPEG lossy compressions. For JPEG 2000 lossy, the Q factor for a specified band can be inquired by adding M_RED , M_GREEN , and M_BLUE , or M_Y , M_U , or M_V . See MbufControl() for more details. Note that for 3-band buffers, only the quantization factor associated with the first band is returned.
M_Q_FACTOR_LUMINANCE	Quantization factor of the Y band for both JPEG 2000 lossy and JPEG lossy compressions of a YUV image buffer.

InquireType	Description
M_Q_FACTOR_CHROMINANCE	Quantization factor of the U and V bands for both JPEG 2000 lossy and JPEG lossy compressions of a YUV image buffer.
M_QUANTIZATION	Identifier of the array buffer containing the quantization table (for a JPEG lossy or JPEG 2000 lossy compression) which is associated with the image buffer. Note that for 3-band buffers, only the identifier of the array buffer associated with the first band is returned.
M_QUANTIZATION_LUMINANCE	Identifier of the array buffer containing the quantization table which is associated with the Y band of a YUV image buffer for both JPEG 2000 lossy and JPEG lossy compressions.
M_QUANTIZATION_CHROMINANCE	Identifier of the array buffer containing the quantization table which is associated with the U and V bands of a YUV image buffer for both JPEG 2000 lossy and JPEG lossy compressions.
For M_IMAGE+M_COMPRESS image buffers with compression type set to M_JPEG2000_LOSSY or M_JPEG2000_LOSSLESS:	
M_DECOMPOSITION_LEVEL	The number of iterations the discrete wavelet transform is passed on the image (for single-band images) or on the first band of the image (for 3-band images).
M_NUMBER_SUBBAND	The number of sub-bands rendered from the discrete wavelet transform passed on an image (for 1-band images) or on the first band of a 3-band image.
For M_IMAGE+M_COMPRESS image buffers with compression type set to M_JPEG2000_LOSSY:	
M_TARGET_SIZE	The requested size of the compressed buffer in bytes. Returns M_DEFAULT if no size has been requested. This inquire type is supported for JPEG 2000 lossy only.
For M_IMAGE+M_COMPRESS image buffers with compression type set to M_JPEG_LOSSY, M_JPEG_LOSSY_INTERLACED, M_JPEG_LOSSLESS, or M_JPEG_LOSSLESS_INTERLACED:	
M_HUFFMAN_DC	Identifier of the array buffer containing the DC Huffman table which is associated with the image buffer. For 3-band buffers, only the identifier of the array buffer associated with the first band is returned.

InquireType	Description
M_HUFFMAN_DC_LUMINANCE	Identifier of the array buffer containing the DC Huffman table which is associated with the Y band of a YUV image buffer.
M_HUFFMAN_DC_CHROMINANCE	Identifier of the array buffer containing the DC Huffman table which is associated with the U and V bands of a YUV image buffer.
For M_IMAGE+M_COMPRESS image buffers with compression type set to M_JPEG_LOSSLESS, or M_JPEG_LOSSLESS_INTERLACED:	
M_PREDICTOR	Type of predictor. This inquire type is supported for JPEG lossless compressions only.
M_RESTART_INTERVAL	Number of lines between restart markers (for JPEG lossless compressions) or number of 8x8 blocks of data between restart markers (for JPEG lossy compressions).
For M_IMAGE+M_COMPRESS image buffers with compression type set to M_JPEG_LOSSY or M_JPEG_LOSSY_INTERLACED:	
M_HUFFMAN_AC	Identifier of the array buffer containing the AC Huffman table which is associated with the image buffer. For 3-band buffers, only the identifier of the array buffer associated with the first band is returned.
M_HUFFMAN_AC_LUMINANCE	Identifier of the array buffer containing the AC Huffman table which is associated with the Y band of a YUV image buffer.
M_HUFFMAN_AC_CHROMINANCE	Identifier of the array buffer containing the AC Huffman table which is associated with the U and V bands of a YUV image buffer.

To extract the internal format of the buffer, use the M_INTERNAL_FORMAT mask to isolate it from the other flags. For example:

```
BufferFormat=MbufInquire(BufId, M_FORMAT, 0);
if ((BufferFormat&M_INTERNAL_FORMAT)==M_BGR24)
{
    ...
}
```

The **UserVarPtr** parameter specifies the address of the variable in which the requested information is to be written. The variable must be of type long, except when the **InquireType** is set to one of the following:

- M_ASSOCIATED_LUT
- M_PARENT_ID
- M_OWNER_SYSTEM
- M_ANCESTOR_ID
- M_HUFFMAN...
- M_QUANTIZATION...

In which case, the **UserVarPtr** parameter requires a pointer to a MIL_ID.

Since the **MbufInquire()** function also returns the requested information, you can set this parameter to M_NULL.

Note MIL-Lite does not support JPEG2000 compression, and requires dedicated hardware for JPEG compression. This is not a restriction under MIL.

Return value The returned value is the value that represents the setting of the requested MIL buffer attribute, cast as long, otherwise M_ERROR is returned.

MbufLoad

Synopsis Load data from a file into a data buffer.

Format **void MbufLoad(FileName, BufId)**

MIL_TEXT_PTR FileName;	Source file name
MIL_ID BufId;	Destination buffer identifier

Description This function loads data from a file into a previously allocated data buffer. The function detects the file format from the data.

Note, you can perform the same operation as **MbufLoad()** using **MbufImport()**, which uses the specified file format to open the file instead of trying to determine the format from the data.

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

The **BufId** parameter specifies the identifier of the destination buffer. This buffer must be big enough in depth and dimensions to hold the data; if not, some data is clipped. For example, if the data is deeper than the buffer, the most-significant bits of the data are truncated when loaded into the buffer. If the buffer depth is greater than that of the data, the data is zero or sign-extended (depending on the data type) when loaded into the buffer. If the buffer is larger in size than the data, exceeding areas of the buffer are unaffected.

When loading an image file that was saved with an associated LUT (color palette), the LUT is also loaded and associated with the destination image buffer. You can obtain the identifier of the associated LUT, using **MbufInquire()**.

Note MIL-Lite does not support JPEG 2000 compression, and requires dedicated hardware for JPEG compression. This is not a restriction under MIL.

See also **MbufImport()**, **MbufExport()**, **MbufSave()**, **MbufRestore()**, **MbufInquire()**, **MbufControl()**

MbufPut

Synopsis Put data from a user-supplied array into a data buffer.

Format **void MbufPut(DestBufId, UserArrayPtr)**

MIL_ID DestBufId;	Destination buffer identifier
void *UserArrayPtr;	Source user array

Description This function copies data from a user-supplied array to a specified MIL destination buffer.

The **DestBufId** parameter specifies the identifier of the destination buffer.

The **UserArrayPtr** parameter specifies the address of the user array from which to copy data into the destination buffer. Ensure that there are enough entries in the user array to fill the destination buffer. **MbufPut()** assumes that the array is of the same data type and depth as the destination buffer's bands.

Note, for multi-band buffers, **MbufPut()** behaves like **MbufPutColor(DestBufId, M_PLANAR, M_ALL_BAND, UserArrayPtr)**. See **MbufPutColor()** for more details.

Example mconvol.c

See also **MbufPut1d()**, **MbufPut2d()**, **MbufPutColor()**, **MbufGet()**, **MbufGet1d()**, **MbufGet2d()**, **MbufGetColor()**

MbufPutColor

Synopsis Put data from a user-supplied array into one or all bands of a data buffer.

Format **void MbufPutColor(DestBufId, DataFormat, Band, UserArrayPtr)**

MIL_ID DestBufId;	Destination buffer identifier
long DataFormat;	Data format of source user array
long Band;	Color band in destination buffer
void *UserArrayPtr;	Source user array

Description This function copies data from a user-supplied array to one or all bands of a specified MIL destination buffer.

The **DestBufId** parameter specifies the identifier of the destination buffer. The internal data format of the destination buffer need not match the specified data format of the user-supplied array; an internal conversion will be performed if necessary. Note, however, if the formats do match the operation will be much faster.

The **DataFormat** parameter specifies the data format of the user-supplied array; this information is required to properly copy the data. Note that Dx and Dy denote the destination width and height, respectively. This parameter must be set to one of the following values:

DataFormat	Description
M_SINGLE_BAND	Copy to a single color band. The user array must be of the same type as the destination buffer and have a size of Dx x Dy.
M_BGR24+M_PACKED	Copy to three bands in an interleaved manner (BGRBGR). The destination buffer must be a three-band, 8-bit buffer and the user array must have a size of Dx x Dy x 3 bytes (Dx x Dy x 3char).
M_BGR32+M_PACKED	Copy to three bands in an interleaved manner (BGRXBGRX). The destination buffer must be a three-band, 8-bit buffer and the user array must have a size of Dx x Dy x 4 bytes (Dx x Dy x long).
M_RGB15+M_PACKED	Copy to three bands in an interleaved manner (RGB 5:5:5). The destination buffer must be a three-band, 8-bit buffer and the user array must have a size of Dx x Dy x 2 bytes (Dx x Dy x 2 unsigned char).

DataFormat	Description
M_RGB16+M_PACKED	Copy to three bands in an interleaved manner (RGB 5:6:5). The destination buffer must be a three-band, 8-bit buffer and the user array must have a size of Dx x Dy x 2 bytes (Dx x Dy x 2 unsigned char).
M_PLANAR	Copy the bands one after the other (RRR...GGG...BBB...). The user array must be the same type as the destination buffer and have a size of Dx x Dy x number of color band of the destination buffer. This format is to be used when copying to all color bands of the destination buffer.

To interpret the array data as top-down (DIB), add M_FLIP to the **DataFormat** parameter. Note that for the M_PLANAR and M_SINGLE_BAND data formats, the M_FLIP flag is not supported.

The **Band** parameter specifies the index of the color band in which to copy. This parameter can be set to any index from 0 to (number of bands of the destination buffer - 1) or to one of the following values:

M_RED	Copy to the red color band.
M_GREEN	Copy to the green color band.
M_BLUE	Copy to the blue color band.
M_ALL_BAND	Copy to all color bands.

If the source buffer is in a HLS (hue, luminance, and saturation) format, the band can be set to: M_HUE, M_LUMINANCE, M_SATURATION, or M_ALL_BAND.

The **UserArrayPtr** parameter specifies the address of the user array from which to copy data into the destination buffer. Ensure that there are enough entries in the user array to fill the color band of the destination buffer.

See also MbufPut(), MbufPut1d(), MbufPut2d(), MbufGet(), MbufGet1d(), MbufGet2d(), MbufGetColor()

MbufPutColor2d

Synopsis Put data from a user-supplied array into a region of one or all bands of a data buffer.

Format **void MbufPutColor2d(DestBufId, DataFormat, Band, OffX, OffY, SizeX, SizeY, UserArrayPtr)**

MIL_ID DestBufId;	Destination buffer identifier
long DataFormat;	Data format of source user array
long Band;	Color band in destination buffer
long OffX;	X pixel offset relative to the parent buffer
long OffY;	Y pixel offset relative to the parent buffer
long SizeX;	Destination buffer region width
long SizeY;	Destination buffer region height
void *UserArrayPtr;	Source user array

Description This function copies data from a user-supplied array to a specified region in one or all bands of a specified MIL destination buffer.

The **DestBufId** parameter specifies the identifier of the destination buffer. The internal data format of the destination buffer need not match the specified data format of the user-supplied array; an internal conversion will be performed if necessary. Note, however, if the formats do match the operation will be much faster.

The **DataFormat** parameter specifies the data format of the user-supplied array; this information is required to properly copy the data. Note that Dx and Dy denote the destination width and height, respectively. This parameter must be set to one of the following values:

DataFormat	Description
M_SINGLE_BAND	Copy to a single color band. The user array must be of the same type as the destination buffer and have a size of Dx x Dy.
M_BGR24+M_PACKED	Copy to three bands in an interleaved manner (BGRBGR). The destination buffer must be a three-band, 8-bit buffer and the user array must have a size of Dx x Dy x 3 bytes (Dx x Dy x 3char).
M_BGR32+M_PACKED	Copy to three bands in an interleaved manner (BGRXBGRX). The destination buffer must be a three-band, 8-bit buffer and the user array must have a size of Dx x Dy x 4 bytes (Dx x Dy x long).

DataFormat	Description
M_RGB15+M_PACKED	Copy to three bands in an interleaved manner (RGB 5:5:5). The destination buffer must be a three-band, 8-bit buffer and the user array must have a size of Dx x Dy x 2 bytes (Dx x Dy x 2 unsigned char).
M_RGB16+M_PACKED	Copy to three bands in an interleaved manner (RGB 5:6:5). The destination buffer must be a three-band, 8-bit buffer and the user array must have a size of Dx x Dy x 2 bytes (Dx x Dy x 2 unsigned char).
M_PLANAR	Copy the bands one after the other (RRR...GGG...BBB...). The user array must be the same type as the destination buffer and have a size of Dx x Dy x number of color band of the destination buffer. This format is to be used when copying to all color bands (M_ALL_BAND) of the destination buffer.

To interpret the array data as top-down (DIB), add M_FLIP to the **DataFormat** parameter. Note that for the M_PLANAR and M_SINGLE_BAND data formats, the M_FLIP flag is not supported.

The **Band** parameter specifies the index of the color band in which to copy. This parameter can be set to any index from 0 to (number of bands of the destination buffer - 1), or to one of the following values:

M_RED	Copy to the red color band.
M_GREEN	Copy to the green color band.
M_BLUE	Copy to the blue color band.
M_ALL_BAND	Copy to all color bands.

If the source buffer is in a HLS (hue, luminance, and saturation) format, the band can be set to: M_HUE, M_LUMINANCE, M_SATURATION, or M_ALL_BAND.

The **OffX** and **OffY** parameters specify the horizontal and vertical pixel offsets of the destination buffer region in which to put the data, relative to the destination buffer's top-left pixel.

The **SizeX** and **SizeY** parameters specify the width and height of the destination buffer region in which to put the data.

The **UserArrayPtr** parameter specifies the address of the user array from which to copy data into the destination buffer. Ensure that there are enough entries in the user array to fill the specified region of the destination buffer.

See also MbufPut(), MbufPut1d(), MbufPut2d(), MbufGet(), MbufGet1d(), MbufGet2d(), MbufGetColor(), MbufGetColor2d()

MbufPutLine

Synopsis Write a specified series of pixels along a specified theoretical line.

Format **void MbufPutLine(ImageBufId, StartX, StartY, EndX, EndY, Mode, NbPixelsPtr, UserArrayPtr)**

MIL_ID ImageBufId;	Image buffer identifier
long StartX;	X start position on the line
long StartY;	Y start position on the line
long EndX;	X end position on the line
long EndY;	Y end position on the line
long Mode;	Operation mode
long *NbPixelsPtr	Number of pixels
void *UserArrayPtr;	Source user array

Description This function reads a series of pixels from a user-defined array and writes them to the specified image, along the theoretical line defined by specified coordinates. The Bresenham algorithm is used to determine the theoretical line.

The **ImageBufId** parameter specifies the identifier of the destination image buffer. This must be a single-band (monochrome) buffer.

The **StartX** and **StartY** parameters specify the horizontal and vertical pixel offsets of the starting position of the line, relative to the top-left pixel of the source buffer.

The **EndX** and **EndY** parameters specify the horizontal and vertical pixel offsets of the finishing position on the line, relative to the top-left pixel of the source buffer.

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

The **NbPixelsPtr** parameter specifies the address of the variable in which to write the number of pixels found along the theoretical line. You can set this parameter to M_NULL if you don't want this value to be evaluated.

The **UserArrayPtr** parameter specifies the address of the user array containing the pixels to insert in the image buffer. **MbufPutLine()** assumes that the array is of the same data type as the destination buffer. Ensure that the user array contains all the pixels to be inserted. To determine the number of pixel values required, you can set this parameter to `M_NULL` and pass a non-null address to **NbPixelsPtr**. In this case, nothing is written to the image buffer.

See also **MbufGetLine()**

MbufPut1d

Synopsis Put data from a user-supplied array into a 1D area of a buffer.

Format **void MbufPut1d(DestBufId, OffX, SizeX, UserArrayPtr)**

MIL_ID DestBufId;	Destination buffer identifier
long OffX;	X pixel offset relative to destination buffer origin
long SizeX;	Width of destination buffer area in which to put data
void *UserArrayPtr;	Source user array

Description This function copies data from a user-supplied array to a one-dimensional area of the specified MIL destination buffer.

The **DestBufId** parameter specifies the identifier of the destination buffer.

The **OffX** parameter specifies the horizontal offset of the destination buffer area in which to put data, relative to the destination buffer's top-left pixel.

The **SizeX** parameter specifies the width of the destination buffer area in which to copy the data (starting from the specified offset **OffX**).

The **UserArrayPtr** parameter specifies the address of the user array from which to copy data into the destination buffer. Ensure that there are enough entries in the user array to fill the specified destination buffer area.

MbufPut1d() assumes that the array is of the same data type as the destination buffer.

Note, for multi-band buffers, **MbufPut1d()** behaves like

MbufPutColor(DestBufId, M_PLANAR, M_ALL_BAND, UserArrayPtr), but puts the data in the specified one-dimensional region. Refer to **MbufPutColor()** for more details.

See also **MbufPut()**, **MbufPut2d()**, **MbufPutColor()**, **MbufGet()**, **MbufGet1d()**, **MbufGet2d()**, **MbufGetColor()**

MbufPut2d

Synopsis Put data from a user-supplied array into a 2d area of a buffer.

Format **void MbufPut2d(DestBufId, OffX, OffY, SizeX, SizeY, UserArrayPtr)**

MIL_ID DestBufId;	Destination buffer identifier
long OffX;	X pixel offset relative to destination buffer origin
long OffY;	Y pixel offset relative to the destination buffer origin
long SizeX;	Width of destination buffer area in which to put data
long SizeY;	Height of destination buffer area in which to put data
void *UserArrayPtr;	Source user array

Description This function copies data from a user-supplied array to a two-dimensional area of the specified MIL destination buffer.

The **DestBufId** parameter specifies the identifier of the destination buffer.

The **OffX** and **OffY** parameters specify the horizontal and vertical pixel offsets of the destination buffer area in which to put the data, relative to the destination buffer's top-left pixel.

The **SizeX** and **SizeY** parameters specify the width and height of the destination buffer area in which to copy the data (starting from the specified offsets **OffX** and **OffY**).

The **UserArrayPtr** parameter specifies the address of the user array from which to copy data into the destination buffer. Ensure that there are enough entries in the user array to fill the specified destination buffer area.

MbufPut2d() assumes that the array is of the same data type as the destination buffer.

Note, for multi-band buffers, **MbufPut2d()** behaves like

MbufPutColor(DestBufId, M_PLANAR, M_ALL_BAND, UserArrayPtr), but puts the data in the specified two-dimensional region. Refer to **MbufPutColor()** for more details.

See also **MbufPut()**, **MbufPut1d()**, **MbufPutColor()**, **MbufGet()**, **MbufGet1d()**, **MbufGet2d()**, **MbufGetColor()**

MbufRestore

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

Format **MIL_ID MbufRestore(FileName, SystemId, BufIdPtr)**

MIL_TEXT_PTR FileName;	Source file name
MIL_ID SystemId;	System identifier
MIL_ID *BufIdPtr;	Storage location for MIL buffer identifier

Description This function restores the data from the specified file and loads it into an automatically allocated buffer. It tries to detect the file format from the data. If the file is in a M_MIL file format, the buffer is allocated with the same attributes as the original buffer, with the exception of M_IMAGE buffers.

In the case of an M_IMAGE type buffer, the **MbufRestore()** function tries to allocate the buffer so that it can be used for acquisition (M_GRAB), display (M_DISP), and processing (M_PROC) operations. If there is insufficient appropriate memory to allocate such a buffer, it allocates one that can be used in all of the above operations except for acquisition (M_GRAB). Note that the maximum (total) number of grab (M_GRAB) buffers that can be allocated is restricted by the total amount of DMA memory that was specified at the time of installation. For systems with on-board processors, the total number of M_GRAB buffers and M_PROC buffers is limited by the amount of on-board memory.

When restoring an image file that was saved with an associated LUT (color palette), the LUT is also restored and associated with the restored image buffer. You can obtain the identifier of the associated LUT, using **MbufInquire()**.

After restoring a buffer, we recommend that you check that the operation was successful by using **MappGetError()** or by checking that the buffer identifier returned is not M_NULL.

Note, you can perform the same operation as **MbufRestore()** by using **MbufImport()**, which uses the specified file format to restore the data instead of trying to determine the format from the data.

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

The **SystemId** parameter specifies the system on which the MIL buffer will be allocated. This parameter must be given a valid system identifier or can be set to `M_DEFAULT_HOST`. In the latter case, the default Host system of the current MIL application is used. You can also specify `M_DEFAULT`, in which case MIL selects the most appropriate system on which to allocate the buffer (either the Host system or any currently allocated system).

The **BufIdPtr** parameter specifies the address of the variable in which the buffer identifier is to be written. Since the **MbufRestore()** function also returns the buffer identifier, you can set this parameter to `M_NULL`. If allocation fails, `M_NULL` is written as the identifier.

Note MIL-Lite does not support JPEG 2000 compression, and requires dedicated hardware for JPEG compression. This is not a restriction under MIL.

Return value The returned value is the buffer identifier. If allocation fails, `M_NULL` is returned.

See also **MbufLoad()**, **MbufSave()**, **MbufExport()**, **MbufImport()**, **MbufInquire()**, **MbufControl()**

MbufSave

Synopsis Save a data buffer in a file, using the MIL output file format.

Format **void MbufSave(FileName, BufId)**

MIL_TEXT_PTR FileName;	Destination file name
MIL_ID BufId;	Source buffer

Description This function saves a previously allocated data buffer in a file, using the MIL output file format (a regular TIFF file format with extra information included in the comment field). The buffer attributes and data type are also saved in the file.

When saving an image buffer (M_IMAGE) that has an associated LUT buffer (color palette), the content of the LUT is also saved with the image.

Note, you can perform the same operation as **MbufSave()** by using **MbufExport()** with its **FileFormatBufId** parameter set to M_MIL.

The **FileName** parameter specifies the name of the file in which to save the data buffer. If this file already exists, it will be overwritten.

The **BufId** parameter specifies the identifier of the data buffer to save.

Note This function is optimized for packed binary buffers.

See also **MbufLoad(), MbufRestore(), MbufExport(), MbufImport(), MbufControl()**

McalAlloc

Synopsis Allocate a calibration object.

Format `MIL_ID McalAlloc(Mode, ModeFlag, CalibrationIdPtr)`

<code>long Mode;</code>	Operation mode
<code>long ModeFlag;</code>	Operation flag
<code>MIL_ID *CalibrationIdPtr;</code>	Address of calibration object identifier

Description This function allocates a calibration object. Use **McalGrid()** or **McalList()** to define the pixel-to-world mapping for the calibration object.

The **Mode** parameter specifies the default calibration mode. It can be set to:

<code>M_LINEAR_INTERPOLATION</code>	Piecewise linear interpolation.
<code>M_PERSPECTIVE_TRANSFORMATION</code>	Perspective transformation.
<code>M_DEFAULT</code>	Same as <code>M_LINEAR_INTERPOLATION</code> .

The **ModeFlag** parameter specifies the function's operation flag. This parameter must be set to `M_DEFAULT`.

The **CalibrationIdPtr** parameter specifies the address in which to return the identifier of the calibration object. Since **McalAlloc()** also returns the identifier, you can set this parameter to `M_NULL`.

Return value The returned value is the identifier of the calibration object if the operation was successful; `M_NULL` if the operation failed.

Example `mcplib.c`

See also **McalGrid()**, **McalList()**

McalAssociate

Synopsis Associate/disassociate a calibration object to/from an image or digitizer.

Format **void McalAssociate(CalibrationId, ImageOrDigitizerId, ControlFlag)**

MIL_ID CalibrationId;	Calibration object identifier
MIL_ID ImageOrDigitizerId;	Image or digitizer identifier
long ControlFlag;	Control flag

Description This function associates a calibration object to an image or digitizer. It can also be used to disassociate a calibration object from an image or digitizer. Note that you do not have to first disassociate a calibration object from an image or digitizer in order to associate a different calibration object to it.

The calibration object gets associated to the image, not to the buffer containing the image. This implies that if you copy or process the image, the operation will copy the image's calibration settings to the destination image.

When you grab an image with a calibrated digitizer, the calibration object currently associated to the digitizer gets associated to the grabbed image.

When you associate a calibration object to an image (either with a call to **McalAssociate()** or a grab with a calibrated digitizer), the image receives a copy of the calibration object's current relative coordinate system and current relative camera position and a reference to the calibration object for all other settings. When you associate a calibration object to a digitizer, the digitizer only receives a reference to the calibration object.

When a calibrated image is used within a MIL module, results from this module can be returned in pixel units or in real-world units; use the calibration object's **M_OUTPUT_COORDINATE_SYSTEM** control to specify your preference. This control is set by **McalControl()**. By default, this control is set to return results in real-world units.

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

The **CalibrationId** parameter specifies the identifier of the calibration object you want to associate. To disassociate a calibration object from an image or digitizer, set this parameter to **M_NULL**.

The **ImageOrDigitizerId** parameter specifies the identifier of the image or digitizer.

The **ControlFlag** parameter specifies the function's control flag. This parameter must be set to M_DEFAULT.

Example mcalib.c

McalControl

Synopsis Control a calibration object parameter setting.

Format **void McalControl(CalibrationId, ControlType, ControlValue)**

MIL_ID CalibrationId;	Calibration object identifier
long ControlType;	Setting to change
double ControlValue;	New value

Description This function changes a setting of a calibration object.

The **CalibrationId** parameter specifies the identifier of the calibration object.

The **ControlType** parameter specifies the setting to change, while the **ControlValue** parameter specifies its new value. The table below lists those settings that can be changed and their allowable values.

ControlType	ControlValue	Meaning
M_CAMERA_POSITION_X	any value	Relative X position of the camera, in world units. (default = origin)
M_CAMERA_POSITION_Y	any value	Relative Y position of the camera in world units. (default = origin)
M_CAMERA_POSITION_Z	M_NULL	Relative Z position of the camera in world units. (default = origin)
M_OUTPUT_COORDINATE_SYSTEM	M_WORLD or M_PIXEL	Coordinate system in which to return results from operations on calibrated images. The default setting is M_WORLD.
M_FOREGROUND_VALUE	M_FOREGROUND_WHITE, M_FOREGROUND_BLACK, or M_DEFAULT	Whether the grid's circles, used with McalGrid() , are lighter or darker than the background. M_DEFAULT automatically determines the appropriate setting.
M_TRANSFORM_CACHE	M_ENABLE or M_DISABLE	Whether to enable or disable a cache used to accelerate the McalTransformImage() function. The default value is M_ENABLE. Disabling the cache saves memory but slows down subsequent calls to McalTransformImage() .

Example `mcalib.c`

McalFree

Synopsis Free a calibration object.

Format **void McalFree(CalibrationId)**

MIL_ID CalibrationId;	Calibration object identifier
-----------------------	-------------------------------

Description This function frees a calibration object.

The **CalibrationId** parameter specifies the identifier of the calibration object.

McalGrid

Synopsis Calibrate your imaging setup using a grid.

Format **void McalGrid(CalibrationId, SrcImageBufId, GridOffsetX, GridOffsetY, GridOffsetZ, RowNumber, ColumnNumber, RowSpacing, ColumnSpacing, Mode, ModeFlag)**

MIL_ID CalibrationId;	Calibration object identifier
MIL_ID SrcImageBufId;	Grid image identifier
double GridOffsetX;	X offset
double GridOffsetY;	Y offset
double GridOffsetZ;	Z offset
long RowNumber;	Number of rows
long ColumnNumber;	Number of columns
double RowSpacing;	Spacing between rows
double ColumnSpacing;	Spacing between columns
long Mode;	Operation mode
long ModeFlag;	Operation flag

Description This function uses an image of a user-defined grid of circles and the world description of this grid to calibrate your imaging setup. The mapping is stored with the specified calibration object.

To create an accurate (sub-pixel) mapping, your physical grid should meet the following guidelines (at the working resolution):

- The radius of the grid's circles should range between 6 and 10 pixels.
- The center-to-center distance between the grid's circles should range from 18 to 32 pixels (22 pixels recommended).
- The minimum distance between the edges of the circles should be 6 pixels.
- The grid should be large enough to cover the area of the image from which you want real-world results.
- The grid image should have high contrast.

The **CalibrationId** parameter specifies the identifier of the calibration object.

The **SrcImageBufId** parameter specifies the identifier of the image containing the grid. This image must be 8- or 16-bit unsigned, with 1 or 3 bands.

The **GridOffsetX**, **GridOffsetY**, and **GridOffsetZ** parameters specify the X, Y, and Z offset, respectively, from the top-left circle to the origin of the real-world coordinate system. **GridOffsetZ** must be set to M_NULL.

The **RowNumber** and **ColumnNumber** parameters specify the number of rows and columns, respectively, in the calibration grid. The minimum number of rows or columns is 2.

The **RowSpacing** and **ColumnSpacing** parameters specify the number of world units between rows and columns, respectively.

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

M_DEFAULT	Use the mode selected at allocation of the calibration object.
M_LINEAR_INTERPOLATION	Use piecewise linear interpolation.
M_PERSPECTIVE_TRANSFORMATION	Use perspective transformation.

The **ModeFlag** parameter specifies the type of grid used. It can be set to:

M_CIRCLE_GRID	Grid of circles.
M_DEFAULT	Same as M_CIRCLE_GRID.

To specify the orientation of the Y-axis, you can add one of the following to the **ModeFlag** parameter.

M_Y_AXIS_UP	The positive Y-axis is orientated 90° counter-clockwise with respect to the positive X-axis.
M_Y_AXIS_DOWN	The positive Y-axis is orientated 90° clockwise with respect to the positive X-axis (default).

Example mcalib.c

McallInquire

Synopsis Inquire about a calibration object setting or about the calibration object associated to an image or digitizer.

Format **long McallInquire(CalibrationOrMilId, InquireType, UserVarPtr)**

MIL_ID CalibrationOrMilId;	Calibration object, image, or digitizer identifier
long InquireType;	Setting to inquire about
void *UserVarPtr;	Address of return value

Description This function inquires about a setting of a calibration object. It can also be used to inquire about the calibration object associated to an image or digitizer.

The **CalibrationOrMilId** parameter specifies the identifier of the calibration object, image, or digitizer.

The **InquireType** parameter specifies the setting about which to inquire. The setting for **InquireType** depends on whether you are inquiring about a calibration object, calibrated image, or calibrated digitizer.

For a calibration object, **InquireType** can be set to:

InquireType	Description
M_CAMERA_POSITION_X	Relative X position of the camera.
M_CAMERA_POSITION_Y	Relative Y position of the camera.
M_CAMERA_POSITION_Z	Relative Z position of the camera.
M_OUTPUT_COORDINATE_SYSTEM	Coordinate system used to return results.
M_RELATIVE_ORIGIN_X	X-coordinate of the origin of the relative coordinate system.
M_RELATIVE_ORIGIN_Y	Y-coordinate of the origin of the relative coordinate system.
M_RELATIVE_ORIGIN_Z	Z-coordinate of the origin of the relative coordinate system.
M_RELATIVE_ORIGIN_ANGLE	Angle, in degrees, at which the relative coordinate system is orientated.
M_TRANSFORM_CACHE	Whether the cache used to accelerate, McalTransformImage0 is enabled or disabled.
M_PIXEL_SIZE_X	Average pixel size in the X direction.
M_PIXEL_SIZE_Y	Average pixel size in the Y direction.

M_ASPECT_RATIO	Average aspect ratio.
M_CALIBRATION_SUCCESSFUL	Whether the last call to McalGrid() or McalList() was successful (M_TRUE or M_FALSE).
M_NUMBER_OF_CALIBRATION_POINTS	Number of calibration points found by McalGrid() or passed to McalList() .
M_CALIBRATION_MODE	Calibration mode: M_LINEAR_INTERPOLATION or M_PERSPECTIVE_TRANSFORMATION.
M_CALIBRATION_IMAGE_POINTS_X	X coordinates of the image points.*
M_CALIBRATION_IMAGE_POINTS_Y	Y coordinates of the image points.*
M_CALIBRATION_WORLD_POINTS_X	X coordinates of the real-world points.*
M_CALIBRATION_WORLD_POINTS_Y	Y coordinates of the real-world points.*
* These settings can be used to determine whether the image points were correctly mapped to their real-world points. The user array should be large enough to hold the total number of points. You can inquire about the number of points using M_NUMBER_OF_CALIBRATION_POINTS.	
The following settings only apply if McalGrid() was used to perform the calibration.	
M_GRID_ORIGIN_X	X-coordinate of the position within the absolute coordinate system associated to the circle in the top-left corner of the grid image.
M_GRID_ORIGIN_Y	Y-coordinate of the position within the absolute coordinate system associated to the circle in the top-left corner of the grid image.
M_GRID_ORIGIN_Z	Z-coordinate of the position within the absolute coordinate system associated to the circle in the top-left corner of the grid image.
M_ROW_NUMBER	Number of rows in the calibration grid.
M_COLUMN_NUMBER	Number of columns in the calibration grid.
M_ROW_SPACING	Number of world units between rows.
M_COLUMN_SPACING	Number of world units between columns.
M_FOREGROUND_VALUE	Whether the grid circles are lighter or darker than the background.
The following settings only apply to an M_PERSPECTIVE_TRANSFORMATION calibration.	
M_AVERAGE_PIXEL_ERROR	Average calibration error, in pixels.
M_AVERAGE_WORLD_ERROR	Average calibration error, in world units.
M_MAXIMUM_PIXEL_ERROR	Maximum calibration error, in pixels.
M_MAXIMUM_WORLD_ERROR	Maximum calibration error, in world units.

For a calibrated image or digitizer, **InquireType** can be set to:

InquireType	Description
M_ASSOCIATED_CALIBRATION	Identifier of its associated calibration object.
M_CORRECTION_STATE	Whether the image has been physically corrected using McalTransformImage() (M_TRUE or M_FALSE).

The **UserVarPtr** parameter specifies the address in which to return the value of the inquired setting. By default, the value is returned as type double. To have it returned as a different type, add one of the following to the **InquireType** parameter.

M_TYPE_CHAR	Return as type char.
M_TYPE_SHORT	Return as type short.
M_TYPE_LONG	Return as type long.
M_TYPE_FLOAT	Return as type float.
M_TYPE_DOUBLE	Return as type double (default).

Since **McalInquire()** also returns the value of the inquired setting, you can set **UserVarPtr** to M_NULL.

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

McalList

Synopsis Calibrate your imaging setup using a list of coordinates.

Format `void McalList(CalibrationId, XPixArray, YPixArray, XWorldArray, YWorldArray, ZWorld, NumPoint, Mode, ModeFlag)`

MIL_ID CalibrationId;	Calibration object identifier
double *XPixArray;	X pixel coordinates
double *YPixArray;	Y pixel coordinates
double *XWorldArray;	X world coordinates
double *YWorldArray;	Y world coordinates
double *ZWorld;	Z world coordinate
long NumPoint;	Number of coordinates
long Mode;	Operation mode
long ModeFlag;	Operation flag

Description This function uses a list of pixel coordinates and their associated world coordinates to calibrate your imaging setup. The mapping is stored with the specified calibration object.

The **CalibrationId** parameter specifies the identifier of the calibration object.

The **XPixArray** and **YPixArray** parameters specify the addresses of the arrays containing the X and Y pixel coordinates.

The **XWorldArray** and **YWorldArray** parameters specify the addresses of the arrays containing the X and Y world coordinates.

The **ZWorld** parameter specifies the Z world coordinate. This parameter must be set to M_NULL.

The **NumPoint** parameter specifies the number of coordinates in the supplied arrays. For a piecewise linear interpolation calibration mode, the minimum number of coordinates is 3. For a perspective transformation calibration mode, the minimum number of coordinates is 4. Note, the specified pixel coordinates should cover the area of the image from which you want real-world coordinates (the working area).

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

M_DEFAULT	Use the mode selected at allocation of the calibration object.
M_LINEAR_INTERPOLATION	Use piecewise linear interpolation.
M_PERSPECTIVE_TRANSFORMATION	Use perspective transformation.

The **ModeFlag** parameter specifies the function's operation flag. This parameter must be set to M_DEFAULT.

McalRelativeOrigin

Synopsis Change the origin and/or orientation of a relative coordinate system.

Format `void McalRelativeOrigin(CalibrationId, XOffset, YOffset, ZOffset, AngularOffset, ControlFlag)`

MIL_ID CalibrationId;	Calibration object identifier
double XOffset;	X offset
double YOffset;	Y offset
double ZOffset;	Z offset
double AngularOffset;	Angular offset
long ControlFlag;	Control flag

Description This function changes the origin and/or orientation of a calibration object's relative coordinate system.

Note that real-world positional results are returned in the relative coordinate system.

The **CalibrationId** parameter specifies the identifier of the calibration object.

The **XOffset**, **YOffset**, and **ZOffset** parameters specify the X, Y, and Z offsets, respectively, from the absolute coordinate system origin to the relative coordinate system origin. The **ZOffset** parameter must be set to M_NULL.

The **AngularOffset** parameter specifies the angle, in degrees, at which to orient the relative coordinate system. This angle is taken at a counter-clockwise direction relative to the X-axis of the absolute coordinate system.

Note that setting the **XOffset**, **YOffset**, **ZOffset**, and **AngularOffset** parameters to 0 has the effect of resetting the relative coordinate system to that of the absolute coordinate system.

The **ControlFlag** parameter specifies the function's control flag. This parameter must be set to M_DEFAULT.

McalRestore

Synopsis Restore a calibration object from a file.

Format **MIL_ID McalRestore(FileName, ControlFlag, CalibrationIdPtr)**

MIL_TEXT_PTR FileName;	File name
long ControlFlag;	Control flag
MIL_ID *CalibrationIdPtr;	Address of calibration object identifier

Description This function restores a calibration object from a file and assigns it an identifier.

The **FileName** parameter specifies the name of the file containing the calibration object.

The **ControlFlag** parameter specifies the function’s control flag. This parameter must be set to M_DEFAULT.

The **CalibrationIdPtr** parameter specifies the address in which to return the identifier of the calibration object. Since **McalRestore()** also returns the identifier, you can set this parameter to M_NULL.

Return value The returned value is the identifier of the calibration object if the operation was successful; M_NULL if the operation failed.

McalSave

Synopsis Save a calibration object to a file.

Format `void McalSave(FileName, CalibrationId, ControlFlag)`

MIL_TEXT_PTR FileName;	File name
MIL_ID CalibrationId;	Calibration object identifier
long ControlFlag;	Control flag

Description This function saves a calibration object to a file.

The **FileName** parameter specifies the name of the file in which to save the calibration object. If the file contains any data, it is overwritten.

The **CalibrationId** parameter specifies the calibration object to save.

The **ControlFlag** parameter specifies the function's control flag. This parameter must be set to M_DEFAULT.

McalTransformCoordinate

Synopsis

Convert coordinates between their world and pixel values.

```
Format void McalTransformCoordinate(CalibrationOrMilId,  
TransformType, X, Y, ResXPtr,  
ResYPtr)
```

MIL_ID CalibrationOrMilId;	Calibration object or image identifier
long TransformType;	Type of conversion
double X;	X coordinates of input
double Y;	Y coordinate of input
double *ResXPtr;	Address of output's X coordinate
double *ResYPtr;	Address of output's Y coordinate

Description This function converts a pair of coordinates from their pixel value to their world value (or vice versa). The conversion can be performed according to a calibration object, calibrated image, or corrected image.

Note that, if you changed the origin and/or orientation of the relative coordinate system (using **McalRelativeOrigin()**), world coordinates will be returned, or assumed to be given, in this relative coordinate system.

The **CalibrationOrMilId** parameter specifies the identifier of the calibration object, calibrated image, or corrected image.

Note that, to convert coordinates from a child image, you must set the **CalibrationOrMilId** parameter to the identifier of the corresponding child buffer.

The **TransformType** parameter specifies whether to perform a pixel-to-world or world-to-pixel conversion. It can be set to:

TransformType	Description
M_PIXEL_TO_WORLD	Convert from pixel to world.
M_WORLD_TO_PIXEL	Convert from world to pixel.

The **X** and **Y** parameters specify the X and Y coordinates, respectively, of the input.

The **ResXPtr** and **ResYPtr** parameters specify the addresses in which to place the X and Y coordinates, respectively, of the output.

See also `McalTransformResult()`, `McalTransformCoordinateList()`

McalTransformCoordinateList

Synopsis Convert a list of coordinates between their world and pixel values.

Format `void McalTransformCoordinateList(CalibrationOrMilId, TransformType, NumPoints, SrcXPtr, SrcYPtr, ResXPtr, ResYPtr)`

MIL_ID CalibrationOrMilId;	Calibration object or image identifier
long TransformType;	Type of conversion
double NumPoints;	Number of points in coordinate list
double *SrcXPtr;	Array of X source coordinates
double *SrcYPtr;	Array of Y source coordinates
double *ResXPtr;	Array of X destination coordinates
double *ResYPtr;	Array of Y destination coordinates

Description This function converts a list of coordinates from their pixel value to their world value (or vice versa). The conversion can be performed according to a calibration object, calibrated image, or corrected image.

Note that, if you changed the origin and/or orientation of the relative coordinate system (using **McalRelativeOrigin()**), world coordinates will be returned, or assumed to be given, in this relative coordinate system.

The **CalibrationOrMilId** parameter specifies the identifier of the calibration object, calibrated image, or corrected image. When an identifier of an image buffer is specified, the transformation uses the calibration object associated with this image.

Note that, to convert coordinates from a child image, you must set the **CalibrationOrMilId** parameter to the identifier of the corresponding child buffer.

The **TransformType** parameter specifies whether to perform a pixel-to-world or world-to-pixel conversion. It can be set to:

TransType	Description
M_PIXEL_TO_WORLD	Convert from pixel to world.
M_WORLD_TO_PIXEL	Convert from world to pixel.

The **NumPoints** parameter specifies the number of points in the coordinate list.

The **SrcXPtr** and **SrcYPtr** parameters specify addresses of the arrays of X- and Y-coordinates, respectively, of the input.

The **ResXPtr** and **ResYPtr** parameters specify the addresses of the arrays in which to place the X- and Y-coordinates, respectively, of the output.

See also **McalTransformCoordinate()**

McalTransformImage

Synopsis Physically transform an image to remove any distortions.

Format `void McalTransformImage(SrcImageBufId, DestImageBufId, CalibrationId, InterpolationMode, OperationType, ControlFlag)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
MIL_ID CalibrationId;	Calibration object identifier
long InterpolationMode;	Interpolation mode
long OperationType;	Operation type
long ControlFlag;	Control flag

Description This function removes distortions in an image by physically transforming it according to a specified calibration object.

The image is transformed such that:

- It has an aspect ratio of 1.
- Its relative coordinate system is aligned with its image coordinate system.
- All of its pixels fit in the destination image, in accordance with the above two points.

Note that an image can only be transformed once. If you pass a transformed image to **McalTransformCoordinateList()**, it will just be copied to the destination image.

The **SrcImageBufId** parameter specifies the identifier of the source image buffer.

The **DestImageBufId** parameter specifies the identifier of the destination image buffer. This buffer should be at least as large as the source image, so that it has at least the same number of pixels to represent all the information in the source image.

The **CalibrationId** parameter specifies the identifier of the calibration object.

The **InterpolationMode** parameter specifies the type of interpolation to perform when associating destination pixels to source points. It can be set to:

M_DEFAULT	Same as M_NEAREST_NEIGHBOR.
M_NEAREST_NEIGHBOR	Nearest-neighbor interpolation.
M_BILINEAR	Bilinear interpolation.
M_BICUBIC	Bicubic interpolation.

To specify how to determine the value of a destination pixel when its associated point falls outside the source buffer, you can add one of the following defines to the **InterpolationMode** parameter. The default is M_OVERSCAN_ENABLE.

M_OVERSCAN_ENABLE	If the associated point falls outside the source buffer, use pixels from the source buffer's ancestor buffer. If the source buffer is not a child buffer or if the associated point falls outside the ancestor buffer, leave the destination pixel as is.
M_OVERSCAN_DISABLE	If the associated point falls outside the source buffer, leave the destination pixel as is.
M_OVERSCAN_CLEAR	If the associated point falls outside the source buffer, set the destination pixel to 0.

The **OperationType** parameter specifies the function's operation type. This parameter must be set to M_DEFAULT.

The **ControlFlag** parameter specifies the function's control flag. This parameter must be set to M_DEFAULT.

Example `mcplib.c`

McalTransformResult

Synopsis Convert a result between its world and pixel value.

Format `void McalTransformResult(CalibrationOrMilId, TransformType, ResultType, Result, ResResult)`

MIL_ID CalibrationOrMilId;	Calibration object or buffer identifier
long TransformType;	Type of transform
long ResultType;	Type of result
double Result;	Input value
double *ResResult;	Output value

Description This function converts a specific result (a length, area, or angle) from its pixel-to-world value or from its world-to-pixel value. The conversion can be performed according to a calibration object, calibrated image, or corrected image. However, since this function uses the average pixel size to perform the conversion, results will be more accurate if you use a corrected image.

The **CalibrationOrMilId** parameter specifies the identifier of the calibration object, calibrated image, or corrected image.

The **TransformType** parameter specifies whether to perform a pixel-to-world or world-to-pixel conversion. It can be set to:

M_PIXEL_TO_WORLD	Convert from pixel to world.
M_WORLD_TO_PIXEL	Convert from world to pixel.

The **ResultType** parameter specifies the type of result the given input value represents. It can be set to:

M_LENGTH	The given value represents a length (for example, the perimeter of an object).
M_LENGTH_X	The given value represents a length in the X direction only.
M_LENGTH_Y	The given value represents a length in the Y direction only.
M_AREA	The given value represents an area.
M_ANGLE	The given value represents an angle.

The **Result** parameter specifies the input value.

The **ResResult** parameter specifies the address in which to place the output value.

See also `McalTransformCoordinateList()`, `McalTransformCoordinate()`

McodeAlloc

Synopsis Allocate a code object.

Format **MIL_ID McodeAlloc(SystemId, CodeType, ControlFlag, CodeIdPtr)**

MIL_ID SystemId;	System identifier
long CodeType;	Type of code
long ControlFlag;	Control flag
MIL_ID *CodeIdPtr;	Storage location for code identifier

Description This function allocates a code object. A code object is used to read or write a specific type of code. The control settings of the code object can be set using **McodeControl()**.

The **SystemId** parameter specifies the system on which to allocate the object. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to allocate the object (it can be the Host or any already allocated system).

The **CodeType** parameter specifies the type of code to read or write. It can be set to one of the following.

M_BC412	BC412.
M_CODABAR	Codabar.
M_CODE39	Code 39.
M_CODE128	Code 128.
M_DATAMATRIX	DataMatrix code (2D).
M_EAN13	EAN-13 code.
M_INTERLEAVED25	Interleaved code.
M_MAXICODE	Maxicode (2D).
M_PDF417	PDF417 (2D).

The **ControlFlag** parameter specifies the function's control flag. This parameter must be set to M_DEFAULT.

The **CodeIdPtr** parameter specifies the address in which to return the identifier of the code object. Since the function also returns the identifier, this parameter can be set to M_NULL.

Return value The returned value is the identifier of the code object if the allocation was successful; M_NULL if allocation failed.

Example mcode.c

See also **McodeFree()**

McodeControl

Synopsis Control a code object.

Format `void McodeControl(CodeId, ControlType, ControlValue)`

MIL_ID CodeId;	Code identifier
long ControlType;	Setting to change
double ControlValue;	New value

Description This function changes a setting of a specified code object.

The **CodeId** parameter specifies the identifier of the code object.

The **ControlType** parameter specifies the setting to change, while the **ControlValue** parameter specifies its new value. The table below lists those settings that can be changed and their allowable values. The **R** and **W** columns indicate with an “X” whether the control types apply to **McodeRead()** (**R**) and/or **McodeWrite()** (**W**) operations.

ControlType	ControlValue	Description	R	W
M_CELL_NUMBER_X	1... n or M_ANY	Number of cells in the X direction of a 2-D code. M_ANY (the default setting) searches for code with any number of cells. Note that for the PDF417 code type M_CELL_NUMBER_X is equal to $17c + 35$, where c is the number of columns, and 35 represents the number of cells required for the start and stop characters.	X	X
M_CELL_NUMBER_Y	1... n or M_ANY	Number of cells in the Y direction of a 2-D code. M_ANY (the default setting) searches for code with any number of cells.	X	X
The number of cells is only used for 2-dimensional symbology (M_DATAMATRIX and M_PDF417 type codes), not for bar codes. The number of cells is not required for Maxicode because it is a fixed size.				

ControlType	ControlValue	Description	R	W
M_CELL_SIZE_MIN	1... n or M_DEFAULT	Minimum cell size, in pixels. For a read operation, M_DEFAULT is equivalent to 1. When reading a Maxicode code type, M_DEFAULT is equivalent to 10. For a write operation, M_DEFAULT causes the code to be resized so as to just fit into the source image of the operation.	X	X
M_CELL_SIZE_MAX	1...n or M_DEFAULT	Maximum cell size, in pixels. M_DEFAULT automatically selects an appropriate value.	X	
M_DOT_SPACING	Distance, in pixels, between 2 dots in a matrix code composed of dots. M_DEFAULT is equivalent to no spacing.		X	
M_ENCODING	<p>Type of encoding scheme. The possible values depend on the code type:</p> <ul style="list-style-type: none"> ■ M_DATAMATRIX: M_ENC_NUM, M_ENC_ALPHA, M_ENC_ALPHANUM, M_ENC_ALPHANUM_PUNC, M_ENC_ASCII, M_ENC_ISO8, or M_ANY (M_ANY can be used to automatically detect the encoding scheme for read operations). ■ M_EAN13: M_ENC_NUM only ■ M_CODE39: M_ENC_STANDARD or M_ENC_ASCII ■ M_INTERLEAVED25: M_ENC_NUM only ■ M_CODE128: M_ENC_ASCII only ■ M_CODABAR: M_ENC_STANDARD only. ■ M_BC412: M_ENC_STANDARD only. ■ M_PDF417: M_ENC_STANDARD only. ■ M_MAXICODE: M_ENC_MODE2, M_ENC_MODE3, M_ENC_MODE4, M_ENC_MODE5, and M_ENC_MODE6. 		X	X

ControlType	ControlValue	Description	R	W
M_ERROR_CORRECTION	Type of error correction. The possible values depend on the code type: <ul style="list-style-type: none"> ■ M_DATAMATRIX: M_ECC_NONE, M_ECC_050, M_ECC_080, M_ECC_100, M_ECC_140, M_ECC_200, or M_ANY (M_ANY can be used to automatically detect the error correction type for read operations). ■ M_EAN13: M_ECC_CHECK_DIGIT only ■ M_CODE39: M_ECC_NONE or M_ECC_CHECK_DIGIT ■ M_INTERLEAVED25: M_ECC_NONE or M_ECC_CHECK_DIGIT ■ M_CODE128: M_ECC_CHECK_DIGIT only ■ M_CODABAR: M_ECC_NONE only ■ M_BC412: M_ECC_NONE only ■ M_PDF417: M_ECC_REED_SOLOMON_1 through M_ECC_REED_SOLOMON_8 type error correction, or M_ANY (M_ANY can be used to automatically detect the error correction type for both read and write operations). ■ M_MAXICODE: M_ECC_REED_SOLOMON only. 		X	X
M_FOREGROUND_VALUE	M_FOREGROUND_WHITE or M_FOREGROUND_BLACK	Foreground color of the code. The default setting is M_FOREGROUND_BLACK.	X	X
M_SEARCH_ANGLE	0.0 – 360.0 or M_DEFAULT	Search at the specified angle. M_DEFAULT is equivalent to 0.0.	X	
M_SEARCH_ANGLE_DELTA_NEG	0.0 – 180.0 or M_DEFAULT	Negative angle range of the search. M_DEFAULT is equivalent to 5.0. *	X	
M_SEARCH_ANGLE_DELTA_POS	0.0 – 180.0 or M_DEFAULT	Positive angle range of the search. M_DEFAULT is equivalent to 5.0. *	X	
* The search is performed between the range of angles defined by: (M_SEARCH_ANGLE - M_SEARCH_ANGLE_DELTA_NEG) to (M_SEARCH_ANGLE + M_SEARCH_ANGLE_DELTA_POS) For the PDF417, Maxicode, and Data Matrix codes, the angular range does not affect the speed of the operation.				

ControlType	ControlValue	Description	R	W
M_SPEED	M_VERY_LOW, M_LOW, M_MEDIUM, M_HIGH, or M_VERY_HIGH	Search speed. The faster the search speed, the less robust the read operation. The default setting is M_MEDIUM.	X	
M_STRING_SIZE	any value or M_ANY	Size of the string for which to search. M_ANY searches for any size.	X	
M_THRESHOLD	any value or M_DEFAULT	Threshold value used to internally binarize the source image of a read operation. M_DEFAULT automatically selects the best threshold value. Note that, if the background is both darker and lighter than the code, a simple binarization will not separate the code from the background. In this case, you should process the image before performing the read operation so that the background is either darker or lighter than the code.	X	
M_USE_PRESEARCH	M_ENABLE or M_DISABLE	Determines whether localization algorithm is performed prior to the decoding step (McodeRead()). This flag is only supported for 2D codes. The default setting is M_DISABLE.	X	

Example `mcode.c`

See also `McodeInquire()`

McodeFree

Synopsis Free a code object.

Format **void McodeFree(CodeId)**

MIL_ID CodeId;	Code identifier
----------------	-----------------

Description This function frees a code object.

The **CodeId** parameter specifies the identifier of the code object.

McodeGetResult

Synopsis Get a result from a read or write operation.

Format **void McodeGetResult(CodeId, ResultType, ResultPtr)**

MIL_ID CodeId;	Code identifier
long ResultType;	Type of result
void *ResultPtr;	Address of array

Description This function retrieves a result from a read or write operation.

McodeGetResult() in combination with **McodeWrite()** can also be used to determine the required image buffer size to encode a given string. First call **McodeWrite()** with **ImageBufId** set to M_NULL and then call **McodeGetResult()** to retrieve the minimum width (M_WRITE_SIZE_X) and height (M_WRITE_SIZE_Y) required for the image.

The **CodeId** parameter specifies the identifier of the code object used in the read or write operation.

The **ResultType** parameter specifies the type of result to retrieve. It can be set to:

M_ANGLE	Angle at which the code was read.
M_CELL_SIZE	Cell size of the code that was read.
M_CELL_NUMBER_X	Number of cells in the X direction of a 2-D code.
M_CELL_NUMBER_Y	Number of cells in the Y direction of a 2-D code.
M_ENCODING	Type of encoding. Possible return values are those listed by McodeControl() , as well as M_ENC_UNKNOWN (unknown encoding type).
M_ERROR_CORRECTION	Type of error correction. Possible return values are those listed by McodeControl() , as well as M_ECC_UNKNOWN (unknown error correction type).
M_SCORE	Confidence score of a read operation.
M_STATUS	Status of a read/write operation. For a read operation, possible return values are M_STATUS_OK, M_STATUS_CRC_FAILED, M_STATUS_ECC_UNKNOWN, M_STATUS_ENC_UNKNOWN or M_STATUS_NOT_FOUND. For a write operation, possible return values are M_STATUS_WRITE_OK or M_STATUS_WRITE_FAILED.

M_STRING	Decoded string. (You can determine the size of the required result array using M_STRING_SIZE). The flag M_ESCAPE_SEQUENCE can be added to this result type to return the string showing the ASCII codes for non-printing characters.
M_STRING_SIZE	Length of the decoded string. The flag M_ESCAPE_SEQUENCE can be added to this result type to return the length of the string, including all ASCII characters used to encode the non-printing characters in the count.
M_THRESHOLD	Threshold value used to internally binarize the source image of a read operation.
M_WRITE_SIZE_X	Minimum width required for the destination image of a write operation.
M_WRITE_SIZE_Y	Minimum height required for the destination image of a write operation.

The **ResultPtr** parameter specifies the address of the array in which to place the specified result. By default, the result is returned as type double. To have it returned as type long, add M_TYPE_LONG to the **ResultType** parameter.

Example mcode.c

McodeInquire

Synopsis Inquire about a code object setting.

Format **void McodeInquire(CodeId, InquireType, UserVarPtr)**

MIL_ID CodeId;	Code object identifier
long InquireType;	Setting to inquire about
void *UserVarPtr;	Address of return value

Description This function inquires about a setting of a specified code object.

The **CodeId** parameter specifies the identifier of the code object.

The **InquireType** parameter specifies the setting about which to inquire. It can be set to one of the settings listed below. See **McodeControl()** or **McodeAlloc()** for the possible return values.

M_CELL_NUMBER_X	Number of cells in the X direction of a 2D code.
M_CELL_NUMBER_Y	Number of cells in the Y direction of a 2D code.
M_CELL_SIZE_MIN	Minimum cell size, in pixels.
M_CELL_SIZE_MAX	Maximum cell size, in pixels.
M_CODE_TYPE	Type of code.
M_DOT_SPACING	Distance, in pixels, between 2 dots.
M_ENCODING	Type of encoding.
M_ERROR_CORRECTION	Type of error correction.
M_FOREGROUND_VALUE	Foreground color of the code.
M_SEARCH_ANGLE	Angle at which to search.
M_SEARCH_ANGLE_DELTA_POS	Positive angle range of the search.
M_SEARCH_ANGLE_DELTA_NEG	Negative angle range of the search.
M_SPEED	Search speed.
M_STRING_SIZE	Size of string for which to search.
M_THRESHOLD	Threshold value used to internally binarize the source image of a read operation.
M_USE_PRESEARCH	Whether or not the pre-search (localization) algorithm is applied.

The **UserVarPtr** parameter specifies the address in which to return the value of the inquired setting. By default, the value is returned as type double. To have it returned as type long, add `M_TYPE_LONG` to **UserVarPtr**.

See also `McodeControl()`

McodeRead

Synopsis Read a specific type of code in an image.

Format **void McodeRead(CodeId, ImageBufId, ControlFlag)**

MIL_ID CodeId;	Code object identifier
MIL_ID ImageBufId;	Source image
long ControlFlag;	Control flag

Description This function searches for a specific type of code in an image. The control settings of the specified code object determine how to perform the operation. Retrieve results using **McodeGetResult()**.

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

Before performing a read operation, certain controls might have to be set in **McodeControl()**, specifically:

M_FOREGROUND_VALUE	This control is essential for all code types, and the image will not be decoded if the foreground value is not correctly set.
M_ENCODING	For code types where M_ANY is not supported.
M_ERROR_CORRECTION	For code types where M_ANY is not supported.
M_CELL_NUMBER_X	For 2D code types only, improves the robustness of the operation.
M_CELL_NUMBER_Y	For 2D code types only, improves the robustness of the operation.
M_CELL_SIZE_MIN*	To improve the robustness of read operations for some code types, especially Data Matrix and Maxicode.
M_CELL_SIZE_MAX*	To improve the robustness of read operations for some code types, especially Data Matrix and Maxicode.
M_STRING_SIZE	For M_BC412, the string size must be specified.
M_SEARCH_ANGLE M_SEARCH_ANGLE_DELTA_NEG M_SEARCH_ANGLE_DELTA_POS	If the code to read is not within 5 degrees of the horizontal axis, and/or if it has bearer bars.

*Note that MIL might have difficulty reading codes if the cell size is less than 2 pixels, even if the size is specified.

The **CodeId** parameter specifies the identifier of the code object.

The **ImageBufId** parameter specifies the image buffer in which to search. This buffer must be 8-bit unsigned.

The **ControlFlag** parameter specifies the function's control flag. This parameter must be set to M_DEFAULT.

Example mcode.c

See also **McodeGetResult()**, **McodeControl()**

McodeRestore

Synopsis Restore a code object previously saved to a file.

Format MIL_ID McodeRestore(FileName, SystemId, ControlFlag, CodeIdPtr)

MIL_TEXT_PTR FileName;	Name for the input file
MIL_ID SystemId;	System identifier
long ControlFlag;	Control flag
MIL_ID* CodeIdPtr;	Storage location for code identifier

Description This function restores a code object that has been saved to a filename with **McodeSave()**. All of the code object's controls are restored and the identifier to the newly allocated code object is returned.

The **FileName** parameter specifies the name of the file in which the code object was saved.

The **SystemId** parameter specifies the system on which to allocate the object. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to allocate the object (it can be the Host or any already allocated system).

The **ControlFlag** parameter specifies the function's control flag. This parameter must be set to M_DEFAULT.

The **CodeIdPtr** parameter specifies the address in which to return the identifier of the code object. Since the function also returns the identifier, this parameter can be set to M_NULL.

See also McodeSave()

McodeSave

Synopsis Save the specified code object in a file.

Format **void McodeSave(FileName, CodeId, ControlFlag)**

MIL_TEXT_PTR FileName;	Name for the output file
MIL_ID CodeId;	Code object identifier
long ControlFlag;	Control flag

Description This function saves a code object in a file so that it can be reloaded with **McodeRestore()**.

The **FileName** parameter specifies the name of the file in which to save the code object. If this file already exists, it will be overwritten.

The **CodeId** parameter specifies the identifier of the data buffer to save.

The **ControlFlag** parameter specifies the function's control flag. This parameter must be set to M_DEFAULT.

See also **McodeRestore()**

McodeWrite

Synopsis Encode an ASCII string.

Format **void McodeWrite(CodeId, ImageBufId, String, ControlFlag)**

MIL_ID CodeId;	Code object identifier
MIL_ID ImageBufId;	Destination image
MIL_TEXT_PTR String;	Null terminated string
long ControlFlag;	Control flag

Description This function encodes a null-terminated string into an image. Retrieve results using **McodeGetResult()**.

The control settings of the specified code object determine how to perform the operation. Before performing a write operation, however, certain controls might have to be set in **McodeControl()**, specifically:

M_CELL_NUMBER_X	For 2D code types only, improves the robustness of the operation.
M_CELL_NUMBER_Y	For 2D code types only, improves the robustness of the operation.
M_CELL_SIZE_MIN	Specifies the cell size, in pixels, of a unit of the code. If this setting is set to M_DEFAULT, the code will be resized so as to just fit into the destination image.
M_ENCODING	For code types where M_ANY is not supported.
M_ERROR_CORRECTION	For code types where M_ANY is not supported.
M_FOREGROUND_VALUE	This control is essential for all code types.

Results of a write operation can be retrieved using **McodeGetResult()**.

The **CodeId** parameter specifies the identifier of the code object.

The **ImageBufId** parameter specifies the image buffer in which to write the string. This buffer must be 8-bit unsigned. In addition, it should be large enough to hold the encoded string. For a given code object and string, you can inquire about the minimum buffer size required by first calling **McodeWrite()** with **ImageBufId** set to M_NULL and then calling **McodeGetResult()** to retrieve the minimum width (M_WRITE_SIZE_X) and height (M_WRITE_SIZE_Y) required for the image.

The **String** parameter specifies the address of the string. Note the following restrictions for certain code types:

- The encoded string for a Codabar code type must be numeric, but can contain the following ASCII marks: -, \$, :, /, ., and +. In addition, the string must start and end with any of the following characters: a, b, c, or d.
- The Interleaved 2 of 5 code type requires a string with an even number of characters.

The **ControlFlag** parameter specifies the function's control flag. This parameter can be set to `M_ESCAPE_SEQUENCE` if you want to encode the string using ASCII codes for non-printable characters; otherwise, set this parameter to `M_DEFAULT`. You must use a code type that supports non-printing characters. When typing the string to encode, the non-printing character's ASCII code must be in hexadecimal format, preceded by `\x`, for example, `\x0D` for ASCII 13, which is the carriage return.

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

See also `McodeGetResult()`

MdigAlloc

Synopsis Allocate a digitizer.

Format **MIL_ID MdigAlloc(SystemId, DigNum, DataFormat, InitFlag, DigIdPtr)**

MIL_ID SystemId;	System identifier
long DigNum;	Digitizer number
MIL_TEXT_PTR DataFormat;	DCF that corresponds to data format of the input device
long InitFlag;	Initialization flag
MIL_ID *DigIdPtr;	Storage location for digitizer identifier

Description This function allocates a digitizer on the specified system so that it can be used by subsequent MIL digitizer functions.

A digitizer on the target system must be allocated in order to acquire data from an input device.

Upon execution of this command, MIL ensures that the digitizer is present before allocating it and generates an error if it is not.

The default input channel is determined by the selected DCF (generally, M_CH0). Some digitizers have multiple input channels. You can switch to another channel using **MdigChannel()**.

When you have completely finished using a digitizer, you should free it, using **MdigFree()**.

The **SystemId** parameter specifies the identifier of the system on which the digitizer will be allocated. This parameter must be given a valid system identifier.

The **DigNum** parameter specifies the number (or rank) of the digitizer that is required. This parameter can be set to one of the following:

M_DEFAULT	Default digitizer (the same as M_DEV0).
M_DEV0	The first digitizer on the specified system.
...	The n th digitizer on the specified system.
M_DEV15	The sixteenth digitizer on the specified system.

The **DataFormat** parameter specifies the name of the digitizer configuration format (DCF) for your input device. Depending on the target system, different DCFs are supported. See the *MIL/MIL-Lite Board Specific Notes* for the valid values. To use the DCF specified in the *milsetup.h* file, set this parameter to `M_CAMERA_SETUP`.

The **InitFlag** parameter specifies the type of initialization you want to perform on the digitizer. This parameter should be set to `M_DEFAULT`.

The **DigIdPtr** parameter specifies the address of the variable in which the digitizer identifier is to be written. Since the **MdigAlloc()** function also returns the digitizer identifier, you can set this parameter to `M_NULL`. If allocation fails, `M_NULL` is written as the identifier.

Return value The returned value is the digitizer identifier. If allocation fails, `M_NULL` is returned.

See also **MdigFree()**, **MappAllocDefault()**

MdigChannel

Synopsis Select the active input channel of a digitizer.

Format **void MdigChannel(DigId, Channel)**

MIL_ID DigId;	Digitizer identifier
long Channel;	Input channel

Description This function selects the active input channel (if any) for the specified digitizer. If the digitizer does not have the specified channel, an error is generated and the last selected channel remains effective. The default channel is the one specified in the data format selected upon digitizer allocation, using **MdigAlloc()**.

The **DigId** parameter specifies the identifier of the digitizer.

The **Channel** parameter specifies the channel on which the digitizer is to input data (signal and sync). This parameter can be set to one of the following values, depending on the number of channels available for the specified digitizer's data format.

M_DEFAULT	Corresponds to the default channel for the specified digitizer data format or M_CH0.
M_CH0	Channel 0
M_CH1	Channel 1
M_CH2	Channel 2
M_CH3	Channel 3
M_RGB	RGB input source (if present). The RGB signal is on channels 0, 1, and 2. The sync is on channel 3. This selection can be used only for RGB input.

If your digitizer has only one channel that supports the selected data format, **Channel** can only be set to M_DEFAULT.

To select a sync channel only, add M_SYNC to the required channel (M_CH...) parameter (for example, M_CH0+M_SYNC).

To select a signal channel only, add M_SIGNAL to the required channel (M_CH...) parameter (for example, M_CH0+M_SIGNAL).

See also **MdigAlloc()**

MdigControl

Synopsis Control the specified digitizer feature.

Format `void MdigControl(DigId, ControlType, ControlValue)`

MIL_ID DigId;	Digitizer identifier
long ControlType;	Control Type
double ControlValue;	Control value

Description This function allows you to control various digitizer settings.

The **DigId** parameter specifies the identifier of the digitizer.

The **ControlType** and **ControlValue** parameters specify, respectively, the digitizer feature to control and the value to assign to the digitizer feature.

ControlType	Description & ControlValue	
M_GRAB_DIRECTION_X	Set the horizontal grab direction:	
	M_REVERSE	Flip the grabbed image horizontally.
	M_FORWARD	Grab normally in the horizontal direction.
	M_DEFAULT	Same as M_FORWARD.
M_GRAB_DIRECTION_Y	Set the vertical grab direction:	
	M_REVERSE	Flip the grabbed image vertically.
	M_FORWARD	Grab normally in the vertical direction.
	M_DEFAULT	Same as M_FORWARD.

ControlType	Description & ControlValue	
M_GRAB_SCALE	Control the vertical and horizontal scaling factor when grabbing data with MdigGrab() or MdigGrabContinuous() .	
	Values of 0.25, 0.5, and 1.0 are typically supported	The ControlValue specifies the scaling factor (reduction or enlargement). For example, if ControlValue is set to 0.5, the source image height and width are reduced by a factor of two.
	M_FILL_DESTINATION	The scaling factor is calculated to fill the destination buffer, if the hardware supports it.
	M_FILL_DISPLAY	The scaling factor is 1, but during a continuous grab operation with the buffer selected on the display, the grab is scaled to fit the size of the display, if the hardware supports it. Therefore, this only affects the copy of the destination buffer in display memory.
M_GRAB_SCALE_X	Control the horizontal scaling factor when grabbing data with MdigGrab() or MdigGrabContinuous() .	
	Values of 0.25, 0.5, and 1.0 are typically supported	The ControlValue specifies the scaling factor (reduction or enlargement).
	M_FILL_DESTINATION	The scaling factor is calculated to fill the width of the destination buffer, if the hardware supports it.
	M_FILL_DISPLAY	The scaling factor is 1, but during a continuous grab operation with the buffer selected on the display, the grab width is scaled to fit the size of the display, if the hardware supports it. Therefore, this only affects the copy of the destination buffer in display memory.

ControlType	Description & ControlValue	
M_GRAB_SCALE_Y	Control the vertical scaling factor when grabbing data with MdigGrab() or MdigGrabContinuous() .	
	Values of 0.25, 0.5, and 1.0 are typically supported	The ControlValue specifies the scaling factor (reduction or enlargement).
	M_FILL_DESTINATION	The scaling factor is calculated to fill the height of the destination buffer, if the hardware supports it.
	M_FILL_DISPLAY	The scaling factor is 1, but during a continuous grab operation with the buffer selected on the display, the grab height is scaled to fit the size of the display if the hardware supports it. Therefore, this only affects the copy of the destination buffer in display memory.
M_GRAB_TIMEOUT	Set the maximum time to wait for a frame before generating an error.	
	M_DEFAULT	Determined by the frame period.
	M_INFINITE	Wait indefinitely. This is recommended only for triggered cameras.
	value in msec	Specify time for wait.
M_GRAB_WINDOW_RANGE	Limit the range of pixel values between 10 and 245: M_ENABLE or M_DISABLE.	
M_SOURCE_OFFSET_X	Set the X offset of the input signal capture window.	
M_SOURCE_OFFSET_Y	Set the Y offset of the input signal capture window.	
M_SOURCE_SIZE_X	Set the width of the input signal capture window.	
M_SOURCE_SIZE_Y	Set the height of the input signal capture window.	

ControlType	Description & ControlValue	
M_GRAB_MODE	Control the synchronization when grabbing data with MdigGrab() .	
	M_SYNCHRONOUS (default)	Synchronize your application with the end of a grab operation (that is, wait until a grab has finished before returning from the grab command).
	M_ASYNCHRONOUS	Do not synchronize your application with the end of a grab operation, but return immediately after initiating the start of a grab. This allows other operations to be performed while waiting for MdigGrab() to be executed. However, only one MdigGrab() command can be queued; a call to another MdigGrab() before the current grab has finished will cause your application to wait until the current grab has finished. Note, in this mode, you can use MdigGrabWait() to force your application to wait until a grab that is in progress has finished.
	M_ASYNCHRONOUS_QUEUED	Do not synchronize your application with the end of a grab operation, but return immediately after initiating the start of the grab. Queue the grab on-board if another grab is issued before the first one has finished. This allows other operations to be performed while waiting for the next MdigGrab() to be executed, but in this case more than one MdigGrab() command can be queued. See <i>MIL/MIL-Lite Board Specific Notes</i> for exceptions.

ControlType	Description & ControlValue	
M_GRAB_FIELD_NUM	Control the number of fields to grab when grabbing data with MdigGrab() . This control type can only be set to 1 or 2, and should only be used for interlaced video. When set to 1, each field is treated like a frame and the following digitizer hooks are aligned with the field: M_GRAB_FRAME_START, M_GRAB_END, and M_GRAB_FRAME_END. To achieve 60 fps in NTSC or 50 fps in PAL, control type M_GRAB_START_MODE must be set to M_FIELD_START.	
M_GRAB_START_MODE	Set the grab start mode to odd, even or any field: M_FIELD_START_ODD, M_FIELD_START_EVEN (M_DEFAULT), or M_FIELD_START.	
M_GRAB_HALT_ON_NEXT_FIELD	Stop grabbing at the end of the current field, rather than at the end of the frame. M_ENABLE, M_DISABLE or M_DEFAULT (same as M_DISABLE).	
M_GRAB_TRIGGER_SOURCE	Set the source of the grab trigger.	
	M_NULL	The trigger is inactive.
	M_DEFAULT	Same as DCF <i>file</i> (if any) or M_NULL.
	M_SOFTWARE	Use software trigger.
	M_HARDWARE_PORT0	Use hardware trigger connected to port 0 (the most common connection for analog). See the <i>MIL/MIL-Lite Board Specific Notes</i> manual.
	M_HARDWARE_PORT1	Use hardware trigger connected to port 1 (the most common connection for digital). See the <i>MIL/MIL-Lite Board Specific Notes</i> manual.
	M_HARDWARE_PORT_CAMERA	Use hardware trigger connected to the same port as the selected camera (MIL-determined). See the <i>MIL/MIL-Lite Board Specific Notes</i> manual.
	M_TIMER1	Trigger on Timer1 signal.
	M_TIMER2	Trigger on Timer2 signal.

ControlType	Description & ControlValue	
M_GRAB_TRIGGER_MODE	Set the hardware trigger activation mode.	
	M_EDGE_RISING	Low to high signal variation (valid with exposure).
	M_EDGE_FALLING	High to low signal variation (valid with exposure).
	M_LEVEL_LOW	Minimum signal level (not valid with exposure).
	M_LEVEL_HIGH	Maximum signal level (not valid with exposure).
	M_DEFAULT	The trigger mode in the DCF file or, if none, M_EDGE_RISING.
M_GRAB_TRIGGER	Set the grab trigger detection state.	
	M_ENABLE	Enable trigger detection.
	M_DISABLE	Disable trigger detection.
	M_DEFAULT	The trigger state from the DCF file or, if none, M_DISABLE.
	M_ACTIVATE	Start the grab immediately (for software trigger). An asynchronous or continuous grab must be in progress.
M_GRAB_EXPOSURE_BYPASS (If the board supports exposures; See Matrox Board Specific Notes)	Activate the manual or automatic exposure model (see <i>Grabbing with triggers</i> in the <i>Matrox Imaging Library User Guide</i>):	
	M_ENABLE	Manual exposure model.
	M_DISABLE	Automatic exposure model.
	M_DEFAULT	Same as M_DISABLE.
For the following M_GRAB_EXPOSURE... control types, you can add M_TIMER1 or M_TIMER2 in manual exposure mode, to control the different on-board exposure timers. When omitted, Timer1 is assumed.		
M_GRAB_EXPOSURE (If the board supports exposures; See Matrox Board Specific Notes)	When using a software trigger source, use this control type to activate the specified grab exposure timer. When using a non-software trigger source, enable or disable the specified grab exposure timer. Note, the M_GRAB_EXPOSURE control type has no effect when grabbing using the automatic exposure model.	
	M_ACTIVATE	Activate a software trigger for the specified exposure timer.
	M_ENABLE	Enable exposure timer.
	M_DISABLE	Disable exposure timer.
	M_DEFAULT	same as .dcf (non-software trigger source).

ControlType	Description & ControlValue	
M_GRAB_EXPOSURE_TIME (If the board supports exposures; See Matrox Board Specific Notes)	<p>Set the time (in nsec) for the active portion of the exposure signal (that is, the exposure time). M_DEFAULT has the same effect as the setting in the digitizer's DCF.</p> <p>When using the automatic exposure model, if a single timer cannot generate the required exposure time, MIL automatically sets up connections with the second timer to generate the requested exposure time length. If ControlValue is set to 0, exposure is disabled and the grab is performed immediately.</p> <p>Note, an error is returned if the specified exposure time cannot be generated.</p>	
M_GRAB_EXPOSURE_MODE (If the board supports exposures; See MIL/MIL-Lite Board Specific Notes)	Set the exposure signal's polarity:	
	M_LEVEL_HIGH	
	M_LEVEL_LOW	
	M_DEFAULT	Same as DCF.
M_GRAB_EXPOSURE_TIME_DELAY (If the board supports exposures; See MIL/MIL-Lite Specific Notes)	<p>Set the delay (in nsec) between the trigger and the start of exposure. If M_DEFAULT, same value as DCF.</p> <p>Note, an error is returned if the specified delay cannot be generated.</p>	
M_GRAB_EXPOSURE_TRIGGER_MODE (If the board supports exposures; See MIL/MIL-Lite Board Specific Notes)	Set the trigger activation mode for specified timer.	
	M_DEFAULT	Same as the .dcf file.
	M_EDGE_RISING	Low-to-high signal variation.
	M_EDGE_FALLING	High-to-low signal variation.

ControlType	Description & ControlValue	
M_GRAB_EXPOSURE_SOURCE (If the board supports exposures; See MIL/MIL-Lite Board Specific Notes)	Select the trigger source for the specified exposure timer if the hardware supports it. The M_GRAB_EXPOSURE_SOURCE control type has no effect when grabbing using the automatic exposure model.	
	M_DEFAULT	Same as the .dcf file.
	M_NULL	Disable specified exposure timer. This has no effect when grabbing using automatic exposure model.
	M_SOFTWARE	Use software trigger. The exposure signal is generated when MdigControl() with M_GRAB_EXPOSURE + M_TIMER n and M_ACTIVATE is called.
	M_HARDWARE_PORT0	Connect hardware trigger to port 0. See the <i>MIL/MIL-Lite Board Specific Notes</i> manual.
	M_HARDWARE_PORT1	Connect hardware trigger to port 1. See the <i>MIL/MIL-Lite Board Specific Notes</i> manual.
	M_HARDWARE_PORT2	Connect hardware trigger to port 2. See the <i>MIL/MIL-Lite Board Specific Notes</i> manual.
	M_VSYNC	Use vertical sync signal.
	M_HSYNC	Use horizontal sync signal.
	M_TIMER1	Use exposure signal generated by Timer1. Use only if setting trigger source for Timer2.
	M_TIMER2	Use exposure signal generated by Timer2. Use only if setting trigger source for Timer1.
	M_CONTINUOUS	No actual trigger. Run selected exposure timer in periodic mode. Automatically reset timer after each exposure signal is output. Exposure signal loops between delay and active mode.

Note If using a software trigger, setting `M_GRAB_TRIGGER` to `M_ACTIVATE` starts a grab immediately; if using a hardware trigger, setting `M_GRAB_TRIGGER` to `M_DISABLE` temporarily stops a continuous grab.

See also `MdigGrab()`, `MdigGrabContinuous()`, `MdigGrabWait()`

MdigFocus

Synopsis Adjust a camera's lens motor to a position which provides optimum focus.

Format **void MdigFocus(DigId, DestImageBufId, FocusImageRegionBufId, FocusHookPtr, UserDataPtr, MinPosition, StartPosition, MaxPosition, MaxPositionVariation, ProcMode, ResultPtr)**

MIL_ID DigId;	Digitizer ID
MIL_ID DestImageBufId;	Destination buffer ID
MIL_ID FocusImageRegionBufId;	Child buffer ID
MFOCUSHOOKFCTPTR FocusHookPtr;	Pointer to hook function
void *UserDataPtr;	User data
long MinPosition;	Minimum position
long StartPosition;	Starting position
long MaxPosition;	Maximum position
long MaxPositionVariation;	Positional increment
long ProcMode;	Processing mode
long *ResultPtr;	Address of optimum position or focus indicator

Description This function adjusts the lens motor of the camera, attached to the specified digitizer, to a position which produces optimum focus. **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, then grabbing and analyzing another image. The process repeats until the optimum focus position is found.

A specified search strategy determines how the position of the lens motor is updated (in which direction and by how much) between grabs. These strategies are described in your user guide.

By default, before an image is analyzed, it is subsampled and filtered.

The **DigId** parameter specifies the identifier of the digitizer. If you want the hook function to perform the grab, as well as move the lens motor, set this parameter to M_NULL.

The **DestImageBufId** parameter specifies the identifier of the buffer in which to place the grabbed images. This buffer should be of an appropriate type to hold the grabbed images. This buffer cannot be signed 3-band or 32-bit.

The **FocusImageRegionBufId** parameter specifies the identifier of a child buffer of the destination buffer. Analysis of each grabbed image is limited to the region specified by this buffer. If you want the entire image analyzed, set this parameter to M_DEFAULT.

The **FocusHookPtr** parameter specifies the address of the user-defined function to call before each grab. This function must be declared as follows:

```
long MFTYPE FocusHook(HookType, Position, UserDataPtr)
```

long HookType;	Indicates whether the optimum focus position was found or whether a new position needs to be analyzed:	
	M_CHANGE	Change the focus position to the value specified by the Position parameter.
	M_ON_FOCUS	The optimum focus position was found and is specified by the Position parameter.
long Position;	Focus position	
void *UserDataPtr;	Pointer to data	

The **UserDataPtr** parameter specifies the address of the data that you want to make available to the user-defined function. If your function does not require data, set this parameter to M_DEFAULT.

The **MinPosition** parameter specifies the minimum focus position of the search. Specify the position in lens motor steps. M_DEFAULT is equal to 0.

The **StartPosition** parameter specifies the starting focus position of the search. Specify the position in lens motor steps. M_DEFAULT is equal to the value specified by **MinPosition**.

The **MaxPosition** parameter specifies the maximum focus position of the search. Specify the position in lens motor steps. M_DEFAULT is equal to 255.

The **MaxPositionVariation** parameter specifies the positional increment to use in a smart scan strategy. M_DEFAULT is equal to 1/16 of the value specified by **MaxPosition**.

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

M_BISECTION	Use a bisection search strategy.
M_REFOCUS	Use a refocus search strategy. The default number of positions used to verify a peak is 2 but can be changed by adding the number to M_REFOCUS.
M_SCAN_ALL	Use a scan_all search strategy.
M_SMART_SCAN	Use a smart_scan search strategy. The default number of positions used to verify a peak is 2 but can be changed by adding a number to M_SMART_SCAN.
M_EVALUATE	Return the focus indicator value for the image passed to DestImageBufId . To evaluate only a region of this image, pass a valid child buffer identifier of the image to FocusImageRegionBufId . To grab an image at the current lens position into DestImageBufId and evaluate this image, pass a valid digitizer identifier to DigId .
M_DEFAULT	Same as M_SMART_SCAN.

To skip the subsampling and/or filtering of each image grabbed by **MdigFocus()**, add M_NO_SUBSAMPLING and/or M_NO_FILTER to the **ProcMode** parameter.

The **ResultPtr** parameter specifies the address in which to return the optimum focus position or the focus indicator value, depending on the mode of operation.

MdigFree

Synopsis Free a digitizer.

Format **void MdigFree(DigId)**

MIL_ID DigId;	Digitizer identifier
---------------	----------------------

Description This function deallocates a digitizer previously allocated with **MdigAlloc()**.

The **DigId** parameter specifies the identifier of the digitizer.

See also **MdigAlloc()**

MdigGrab

Synopsis Grab data from an input device into a buffer.

Format **void MdigGrab(DigId, DestImageBufId)**

MIL_ID DigId;	Digitizer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier

Description This function uses the specified digitizer to acquire data from an input device (generally a camera) and stores this data in the destination image buffer.

When grabbing in color, all bands will be filled simultaneously. Note, the destination image buffer must have the same number of color bands (in general three) as the digitizer.

When acquiring data from a line-scan type of input device, each line of the destination image buffer is filled from top to bottom or a single line is grabbed, depending on the data format specification passed to **MdigAlloc()**. The operation will only end when the entire buffer has been filled.

When acquiring data from an interlaced camera, both the odd and even fields are grabbed.

You can use **MdigGrabContinuous()** to grab multiple frames of data.

The **DigId** parameter specifies the identifier of the digitizer.

The **DestImageBufId** parameter specifies the identifier of the destination image buffer.

Example Mgrab.c

See also **MdigGrabContinuous(), MdigControl()**

MdigGrabContinuous

Synopsis Grab data continuously from an input device.

Format `void MdigGrabContinuous(DigId, DestImageBufId)`

MIL_ID DigId;	Digitizer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier

Description This function uses the specified digitizer to continuously acquire frames of data from the specified input device (generally a camera) and stores this data in the destination image buffer, until **MdigHalt()** is called.

When acquiring data from a line-scan type of input device, each line of the destination image buffer is filled from top to bottom or a single line is grabbed, depending on the data format specification passed to **MdigAlloc()**. The operation will only end when the entire buffer has been filled.

When grabbing in color, the destination image buffer must have the same number of color bands (in general three) as the digitizer; all bands will be filled simultaneously.

The **DigId** parameter specifies the identifier of the digitizer.

The **DestImageBufId** parameter specifies the identifier of the destination image buffer.

Examples `mdispovr.c, mwindisp.c, mfocus.c, mdbproc.c, mgrabhk.c, mgrabseq.c, msubtrac.c`

See also **MdigHalt()**, **MdigGrab()**, **MdigControl()**

MdigGrabWait

Synopsis Wait for the end of the grab in progress.

Format **void MdigGrabWait(DigId, Flag)**

MIL_ID DigId;	Digitizer identifier
long Flag;	Digitizer flag

Description This function allows you to temporarily override a grab mode of M_ASYNCHRONOUS on the specified digitizer (see **MdigControl()**). Using this function allows your application to wait for the grab in progress to end, before continuing.

The **DigId** parameter specifies the identifier of the digitizer.

The **Flag** parameter specifies the digitizer flag to set. This parameter must be set to one of the following:

M_GRAB_END	Wait for the end of the current grab.
M_GRAB_NEXT_FRAME	Wait for the end of the current frame grab.
M_GRAB_NEXT_FIELD	Wait for the end of the current field grab.

The M_GRAB_END flag should not be used when grabbing data with **MdigGrabContinuous()**.

Some of these flags are not supported on all platforms.

See also **MdigControl()**, **MdigGrab()**

MdigHalt

Synopsis Halt a continuous grab from an input device.

Format void MdigHalt(DigId)

MIL_ID DigId;	Digitizer identifier
---------------	----------------------

Description This function stops the specified digitizer from grabbing data. It should be used when performing a continuous grab with **MdigGrabContinuous()**.

This function will wait for the end of the current frame before returning, to ensure the last frame is always valid. To override this, use **MdigControl()** with M_GRAB_HALT_ON_NEXT_FIELD set to M_ENABLE.

The **DigId** parameter specifies the identifier of the digitizer.

Examples
mdispovr.c, **MdigGrabContinuous()**, **MdigControl()**
mwindisp.c
See also

MdigHookFunction

Synopsis Hook a function to a digitizer event.

Format **void MdigHookFunction(DigId, HookType,
 HookHandlerPtr, UserDataPtr)**

MIL_ID DigId;	Digitizer identifier
long HookType;	Type of event to hook
MDIGHOOKFCTPTR HookHandlerPtr;	Pointer to hook function
void *UserDataPtr	User data pointer

Description This function allows you to attach or detach a user-defined function to a specified digitizer event. Once a hook-handler function is defined and hooked to an event, it is automatically called when the event occurs.

Note that functions hooked to an event execute on a distinct thread. This permits the functions to run asynchronously from the operation that fired the event and from functions hooked to other events. Although there is a small queue to permit a certain amount of overlap, hooked functions should not take longer to execute than the period in which two of their associated events can occur. You cannot determine the instance of the event that fired the function, and even if this were possible, this information would generally not be very useful because, for example, you could miss a grab. Typically, a hooked function performs the minimum number of operations required and, if necessary, performs longer processes by launching other threads.

You can hook more than one function to an event by making separate calls to **MdigHookFunction()** for each function that you want to hook. MIL automatically chains and keeps an internal list of all these hooked functions. When a function is hooked, this new function is added to the end of the list. When the event happens, all user-defined functions in the list will be executed in the same order that they were hooked to the event. You can also remove any function from the list; in this case, MIL preserves the order of the remaining functions in the list.

The **DigId** parameter specifies the identifier of the digitizer.

The **HookType** parameter specifies the event type. This parameter can be set to one of the values in the following tables. Note that a hooked function must be unhooked by combining the **HookType** parameter with M_UNHOOK.

Hook Type	Description
M_GRAB_START	Hook to the start of each grab.
M_GRAB_END	Hook to the end of each grab.
M_GRAB_FRAME_START	Hook to the start of grabbed frames.
M_GRAB_FRAME_END	Hook to the end of grabbed frames.
M_GRAB_FIELD_END	Hook to the end of grabbed fields.
M_GRAB_FIELD_END_ODD	Hook to the end of grabbed odd fields.
M_GRAB_FIELD_END_EVEN	Hook to the end of grabbed even fields.

When grabbing continuously, the M_GRAB_START event is fired only when the first frame is grabbed. To be notified at the start of each frame, use the M_GRAB_FRAME_START event.

When grabbing continuously, the M_GRAB_END event is fired only when the last frame is grabbed. To be notified at the end of each frame, use the M_GRAB_FRAME_END event.

When a camera is connected, whether a grab is occurring or not, the **HookType** parameter can be set to one of the following:

M_FRAME_START	Hook to the start of the incoming signal's frames.
M_FIELD_START	Hook to the start of the incoming signal's fields.
M_FIELD_START_ODD	Hook to the start of the incoming signal's odd fields.
M_FIELD_START_EVEN	Hook to the start of the incoming signal's even fields.
Note: These parameters are not supported on all systems. See <i>MIL/MIL-Lite Board-Specific Notes</i> to verify if these parameters are supported on your board.	

Starting or stopping a grab does not affect the occurrence of the above four events.

The **HookHandlerPtr** parameter specifies the address of the function that should be called when an event occurs.

The hook-handler function, pointed to by **HookHandlerPtr**, must be declared as follows:

long MFTYPE HookHandler(HookType, EventId, UserDataPtr);	
long HookType;	Type of event hooked
MIL_ID EventId;	Event identifier (currently set to null)
void MPTYPE *UserDataPtr;	User data pointer

Upon successful completion, the hook-handler function should return M_NULL. Note, MDIGHOOKFCTPTR and MPTYPE are reserved MIL predefined types for function and data pointers.

The **UserDataPtr** parameter specifies the address of the user data that you want to make available to the hook-handler function. This address is passed to the hook-handler function, through its **UserDataPtr** parameter, when the specified event occurs. Set this parameter to M_NULL if not used.

Return value The original prototype of this function has been kept for backwards compatibility. However, because of the current chaining method, the function always returns null.

Examples mgrabhk.c

See also **MdigControl()**

MdigInquire

Synopsis Inquire about a digitizer parameter setting.

Format `long MdigInquire(DigId, InquireType, UserVarPtr)`

MIL_ID DigId;	Digitizer identifier
long InquireType;	Type of information to inquire
void *UserVarPtr;	Storage location for inquired information

Description This function inquires about the specified digitizer parameter setting.

The **DigId** parameter specifies the identifier of the digitizer.

The **InquireType** parameter specifies the digitizer parameter about which to inquire. This parameter can be set to one of the following values:

InquireType	Description
M_OWNER_SYSTEM	The MIL identifier (MIL_ID) of the system on which the digitizer has been allocated (MdigAlloc()).
M_NATIVE_ID	The native identifier of the digitizer (if any).
M_NUMBER	Digitizer rank in the system (MdigAlloc()).
M_FORMAT	Digitizer data format (MdigAlloc()).
M_FORMAT_SIZE	Number of characters in the digitizer data format string.
M_INIT_FLAG	Digitizer initialization flag (MdigAlloc()).
M_CHANNEL	Current channel of the digitizer (MdigChannel()).
M_CHANNEL+M_SYNC	Current synchronization channel of the digitizer (MdigChannel()).
M_CHANNEL+M_SIGNAL	Current signal channel of the digitizer (MdigChannel()).
M_CHANNEL_NUM	Number of available channels of the device (MdigChannel()).
M_LUT_ID	MIL identifier (MIL_ID) of the LUT associated with the digitizer (MdigLut()).
M_BLACK_REF	Digitizer black reference level (MdigReference()).
M_WHITE_REF	Digitizer white reference level (MdigReference()).
M_HUE_REF	Digitizer hue reference level (MdigReference()).
M_SATURATION_REF	Digitizer saturation reference level (MdigReference()).

InquireType	Description
M_BRIGHTNESS_REF	Digitizer brightness reference level (MdigReference()).
M_COLOR_MODE See the <i>Matrox Board Specific Notes</i> to determine which mode applies to your particular board.	Monochrome or color input: M_MONOCHROME M_RGB, M_MONO8_VIA_RGB M_COMPOSITE, M_EXTERNAL_CHROMINANCE
M_CONTRAST_REF	Digitizer contrast reference level (MdigReference()).
M_GRAB_SCALE_X	Digitizer horizontal and vertical scaling factor (MdigControl()).
M_GRAB_SCALE_X	Digitizer horizontal scaling factor (MdigControl()).
M_GRAB_SCALE_Y	Digitizer vertical scaling factor (MdigControl()).
M_GRAB_MODE	Grab synchronization (M_SYNCHRONOUS, M_ASYNCHRONOUS, or M_ASYNCHRONOUS_QUEUED.) (MdigControl()).
M_GRAB_FIELD_NUM	Number of fields grabbed when MdigGrab() is called. (MdigControl()).
M_GRAB_START_MODE	Type of field on which to grab.
M_GRAB_HALT_ON_NEXT_FIELD	Whether to stop grabbing as soon as possible, whether the last frame is valid or not (MdigControl()).
M_GRAB_TRIGGER_SOURCE	Grab trigger source (MdigControl()).
M_GRAB_TRIGGER_MODE	Hardware trigger activation mode (MdigControl()).
M_GRAB_TRIGGER	Grab trigger state (M_ENABLE, M_DISABLE, M_START_GRAB or M_DEFAULT (same as .dcf, if any, or M_DISABLE) (MdigControl()).
M_GRAB_WINDOW_RANGE	State of limiting the range of the grabbed pixel values: M_ENABLE or M_DISABLE.
M_SIZE_X	Digitizer input width.
M_SIZE_Y	Digitizer input height.
M_SIZE_BAND	Number of input color bands of the digitizer.
M_SIZE_BAND_LUT	Number of input color bands of the input LUT (if any) associated with the digitizer.
M_SIZE_BIT	Number of bits of the digitizer.
M_SIGN	Digitizer data range (M_SIGNED or M_UNSIGNED).
M_TYPE	Digitizer data type (number of bits + M_SIGNED or M_UNSIGNED).

InquireType	Description
M_SOURCE_SIZE_X	Width of the input-signal capture window.
M_SOURCE_SIZE_Y	Height of the input-signal capture window.
M_SOURCE_OFFSET_X	X offset of the input-signal capture window.
M_SOURCE_OFFSET_Y	Y offset of the input signal capture window.
M_SCAN_MODE	Scan mode (M_INTERLACE, M_PROGRESSIVE, or M_LINESCAN).
M_INPUT_MODE	Analog or digital input (M_ANALOG or M_DIGITAL).
M_GRAB_EXPOSURE_BYPASS (If the board supports exposures; See <i>Matrox Board Specific Notes</i>)	The exposure model that is activated (manual or automatic).
For the following M_GRAB_EXPOSURE... inquire types, you can add M_TIMER1 or M_TIMER2 in manual exposure mode, to control the different on-board exposure timers. When omitted, Timer1 is assumed.	
M_GRAB_EXPOSURE (If the board supports exposures; See <i>Matrox Board Specific Notes</i>)	Exposure timer state for non-software trigger source: M_ENABLE or M_DISABLE.
M_GRAB_EXPOSURE_MODE (If the board supports exposures; See <i>Matrox Board Specific Notes</i>)	Exposure signal's polarity: M_LEVEL_HIGH or M_LEVEL_LOW.
M_GRAB_EXPOSURE_SOURCE (If the board supports exposures; See <i>Matrox Board Specific Notes</i>)	The trigger source for the specified exposure timer if the hardware supports it.
M_GRAB_EXPOSURE_TIME (If the board supports exposures; See <i>Matrox Board Specific Notes</i>)	Time (in nsec) for the active portion of the exposure signal (that is, the exposure time). M_DEFAULT has the same effect as the setting in the digitizer's DCF.
M_GRAB_EXPOSURE_TIME_DELAY (If the board supports exposures; See <i>Matrox Board Specific Notes</i>)	The delay (in nsec) between the trigger and the start of exposure.
M_GRAB_EXPOSURE_TRIGGER_MODE (If the board supports exposures; See <i>Matrox Board Specific Notes</i>)	Trigger activation mode for specified timer: M_EDGE_RISING or M_EDGE_FALLING.

You can inquire about the reference level on a specific input channel by adding one of the following predefined values to M_BLACK_REF and M_WHITE_REF.

M_CH0_REF	Inquire about reference level on channel 0 (default).
M_CH1_REF	Inquire about reference level on channel 1.
M_CH2_REF	Inquire about reference level on channel 2.
M_CH3_REF	Inquire about reference level on channel 3.

For example M_BLACK_REF+M_CH1_REF.

The **UserVarPtr** parameter specifies the address of the variable in which the requested information is to be written. Since the **MdigInquire()** function also returns the requested information, you can set this parameter to **M_NULL**.

The **UserVarPtr** parameter should be a pointer to a long, except when **InquireType** is set to one of the following:

- **M_OWNER_SYSTEM** and **M_LUT_ID**, in which case it should be a pointer to a **MIL_ID**.
- **M_FORMAT**, in which case it should be a pointer to an array of **MIL_TEXT_CHAR**.
- **M_GRAB_SCALE_X** and **M_GRAB_SCALE_Y**, in which case it should be a pointer to a double.

Return value Except for the **M_FORMAT** inquire type, the returned value is the setting of the requested digitizer attribute, cast to long. For the **M_FORMAT** inquire type, the returned value is **M_NULL**.

See also **MdigAlloc()**, **MdigChannel()**, **MdigControl()**, **MdigReference()**

MdigLut

Synopsis Copy a LUT buffer to a digitizer LUT.

Format `void MdigLut(DigId, LutBufId)`

MIL_ID DigId;	Digitizer identifier
MIL_ID LutBufId;	LUT buffer identifier

Description This function copies a LUT buffer to the specified digitizer LUT. MIL uses the data format of the digitizer to determine whether a LUT is supported. If it is not, an error is generated.

The **DigId** parameter specifies the identifier of the digitizer.

The **LutBufId** parameter specifies the identifier of a previously allocated LUT buffer (with an M_LUT attribute).

The number of entries in the LUT buffer must match those of the digitizer, but the pixel depth of the LUT buffer and the digitizer can be different. A 1-band LUT can be copied to a 1-band digitizer LUT, or a 3-band digitizer LUT. If the LUT buffer has a single color band, its data is copied to the LUTs of each of the digitizer's color bands; this will pseudo-color the grabbed image. A three-band LUT can also be copied to a 1-band digitizer LUT or a 3-band digitizer LUT. If the LUT buffer has three color bands, each band copied to the single band digitizer LUT. You can set this parameter to M_DEFAULT to associate the default pass-through LUT (or transparent LUT) with the digitizer.

See also `MdigAlloc()`, `MbufAlloc1d()`

MdigReference

Synopsis Select digitization reference level.

Format **void MdigReference(DigId, ReferenceType, ReferenceLevel)**

MIL_ID DigId;	Digitizer identifier
long ReferenceType;	Reference type
long ReferenceLevel;	Reference level

Description This function sets (if available) the reference levels used to digitize the analog signal received from an input device (generally a camera). This function is specific to analog input devices. Depending on the type of digitizer and input signal, some reference types are not applicable.

The **DigId** parameter specifies the identifier of the digitizer on which to set the reference level. An error is generated if the specified digitizer does not support the type of programmable digitization reference levels specified.

The **ReferenceType** parameter specifies the reference level type to adjust for the specified digitizer. This parameter can be set to one of the following:

M_BLACK_REF	Set the input signal's digitization black reference level (0).
M_WHITE_REF	Set the input signal's digitization white reference level (eg: 0xff for 8-bit digitization).
M_BRIGHTNESS_REF	Set the brightness level for composite input signals.
M_CONTRAST_REF	Set the contrast level for composite input signals.
M_HUE_REF	Set the hue level for composite input signals.
M_SATURATION_REF	Set the saturation level for composite input signals.

On many digitizers, when using RGB or multi-tap input and setting **ReferenceType** to M_BLACK_REF or M_WHITE_REF, you can control the reference level of a specific input channel by combining it with one of the following:

M_CH0_REF	Set the reference level on input channel 0.
M_CH1_REF	Set the reference level on input channel 1.
M_CH2_REF	Set the reference level on input channel 2.
M_CH3_REF	Set the reference level on input channel 3.
M_ALL_REF	Set the reference level on all input channels. (This is the default setting).

The **ReferenceLevel** parameter specifies the level of reference. This parameter can be set to a value between M_MIN_LEVEL and M_MAX_LEVEL, inclusive. The value may be expressed as an integer within this range, or as M_MIN_LEVEL + n or M_MAX_LEVEL - n. If you set this parameter to M_DEFAULT, the reference levels are set to the default levels for the specified digitizer data format.

To calculate the value to pass to *MdigReference()*, use the following equation with the appropriate voltages specified in the *MIL Board-specific notes* for your particular board. The smallest voltage increment supported by your

$$\text{Value to pass to } MdigReference() = \left(\frac{\text{Voltage needed} - \text{minimum voltage}}{\text{maximum voltage} - \text{minimum voltage}} \right) (M_MAX_LEVEL - M_MIN_LEVEL)$$

board can differ such that consecutive reference-level settings might produce the same result.

Note, some digitizers might take a few milliseconds before the reference level stabilizes.

See also MdigAlloc()

MdispAlloc

Synopsis Allocate a display.

Format **MIL_ID MdispAlloc(SystemId, DispNum, DispFormat, InitFlag, DisplayIdPtr)**

MIL_ID SystemId;	System identifier
long DispNum;	Display number
MIL_TEXT_PTR DispFormat;	Display format name or file name
long InitFlag;	Initialization flag
MIL_ID *DisplayIdPtr;	Storage location for the display identifier

Description This function allocates a display on the specified system so that it can be used by subsequent MIL display functions.

A display must be allocated to display an image buffer. Note that the buffer and the display should be allocated on the same system.

When you have completely finished using a display, you should free it, using **MdispFree()**.

The **SystemId** parameter specifies the system on which the display is allocated. This parameter must be given a valid system identifier.

The **DispNum** parameter specifies the number (or rank) of the display. This parameter should always be set to M_DEFAULT since MIL will find the best device to use when displaying an image in a windowed display or in an auxiliary display. If your imaging board has a display section, and it is available, MIL will typically use it for display purposes.

The **DispFormat** parameter specifies the display format or the name of the file (*.vcf) in which the display format is to be found. For windowed displays, **DispFormat** must be set to "M_DEFAULT". For auxiliary displays, **DispFormat** can be set to a string that specifies the required video output format; note that when you are allocating an auxiliary display, MIL does not impose any restrictions on this format. See the *MIL/MIL-Lite Board-Specific Notes* manual for the formats supported by your board. Note that for auxiliary displays, **DispFormat** can also be set to "M_DEFAULT", which indicates that MIL should use the default display format specified in the *MIL/MIL-Lite Board-Specific Notes* manual.

The **InitFlag** parameter specifies your display's type. This parameter can be set to one of the following:

M_WINDOWED	The image buffer selected for display purposes is presented in its own window on the Windows desktop screen(s). The display window is tracked and updated with the image buffer selected for display; that is, if the window moves or is occluded, the window is updated with the image buffer accordingly. For each system that has been allocated, you can allocate and select up to a maximum of 64 windowed displays.
M_AUXILIARY	The image buffer selected for display has no window associated with it and is presented at the top-left corner on the auxiliary screen, which is any output device that is not part of the Windows desktop. You are responsible for moving and tracking this type of display, if required. Note that if you are using the second CRT controller provided with the Matrox Millennium G400, G450, or G550 for your auxiliary screen, your display driver's DualHead mode must be disabled, otherwise both the display driver and the MIL driver will attempt to access the second CRT controller. In addition, the second CRT controller provided with the G400 does not support encoded video formats, but the G450 and G550 do.
M_DEFAULT	Typically, the default display type is M_WINDOWED. However, a Matrox imaging board might be dedicated for MIL auxiliary display, which can make the default M_AUXILIARY. For more information, see the board's installation and hardware reference manual.

For windowed displays, when using a 256-color Windows display resolution, you can control the Windows display function that MIL uses for display by adding one of the following to **InitFlag**. To independently control the display of 8-bit and 3-band 8-bit images, add both an M_DISPLAY_8... and M_DISPLAY_24... display initialization to **InitFlag**.

Display initialization	Description
M_DISPLAY_ENHANCED M_DISPLAY_8_ENHANCED M_DISPLAY_24_ENHANCED (default)	When using an enhanced initialization, the MIL display calls the Microsoft Video for Windows DrawDIBDraw() function to display image buffers. This function's use of dithering particularly improves the display of 3-band 8-bit images under a 256-color display resolution. Note, with enhanced initializations, the actual display color values are selected, on a best-match basis, from the logical palette's available display colors. Therefore, effects such as those of an inverse LUT are not possible. This is the default display initialization for an 8-bit 3-band image buffer.

Display initialization	Description
M_DISPLAY_BASIC M_DISPLAY_8_BASIC (default) M_DISPLAY_24_BASIC	When using a basic with optimization initialization, the MIL display calls the Windows API StretchDIBits() , StretchBlt() , or DirectDrawBlt() function to display image buffers. When 8-bit images are displayed, the pixel values are used, as much as possible, to index the physical LUTs. When 3-band 8-bit images are displayed in a 256-color display resolution, the display uses an algorithm optimized for speed. This algorithm converts 24 bits to 8 bits by taking the most-significant bits of each component: 3 bits each are taken from the red and green components, and 2 bits from the blue. This produces an 8-bit DIB with 3:3:2 RGB values for display; it is these values that are used to address the physical LUTs. This is the best possible combination when you are not aware of the color content of the image buffer.
M_DISPLAY_WINDOWS M_DISPLAY_24_WINDOWS	When using a basic without optimization initialization, the MIL display calls the Windows API StretchDIBits() , StretchBlt() or DirectDrawBlt() function to display image buffers; however no optimization for speed is done when displaying a 3-band 8-bit image in a 256-color display resolution. This can result in slow performance. This display initialization is a combination of M_DISPLAY_8_BASIC and M_DISPLAY_24_WINDOWS.

You can add one of these values to the **InitFlag** to control the Windows zoom type that MIL uses for the display:

Zoom initialization	Description
M_ZOOM_ENHANCED	When using an enhanced initialization, the DrawDIBDraw() function is called to perform a zoom. Although zooming might be a little slower than using the basic initialization option, it does not alter the dithering quality, providing a better quality zoom. This option is the default and is only available when M_DISPLAY_XXX_ENHANCED is used. When adding a zoom initialization type, the default is M_ZOOM_ENHANCED . If you select only M_DISPLAY_ENHANCED , M_ZOOM_ENHANCED is assumed.
M_ZOOM_BASIC	When using a basic initialization, Windows (Windows API functions) is called to perform a zoom. Note, if M_DISPLAY_XXX_ENHANCED is used, this zoom might alter the quality of the DrawDIBDraw() dithering.

The **DisplayIdPtr** parameter specifies the address of the variable in which to write the display identifier. Since the **MdispAlloc()** function also returns the display identifier, you can set this parameter to `M_NULL`. If allocation fails, `M_NULL` is written as the identifier.

Return value The returned value is the display identifier. If allocation fails, `M_NULL` is returned.

See also **MdispControl()**, **MdispFree()**, **MappAllocDefault()**

MdispControl

Synopsis Control the MIL display.

Format **void MdispControl(DisplayId, ControlType, ControlValue)**

MIL_ID DisplayId;	Display identifier
long ControlType;	Window feature to change
long ControlValue;	Value of the window feature

Description This function allows you to control the specified MIL display; it does this by setting the state of the display's individual features.

The **DisplayId** parameter specifies the identifier of the target display.

The **ControlType** and **ControlValue** parameters specify the display feature to modify and the new value to assign to the feature, respectively. The control types for windowed displays can control the default MIL or user-specified window of a display (**MdispSelect()** or **MdispSelectWindow()**).

The corresponding combinations for the **ControlType** and **ControlValue** parameters are:

ControlType	Description and ControlValue	
Unless otherwise stated, the following controls are only available with windowed displays.		
M_DESKTOP_CHANGE	Allow the update of the Windows desktop: M_ENABLE or M_DISABLE. Note: M_DISABLE (stop desktop update) should be used carefully and only for short periods of time otherwise undesirable results can occur.	
M_THREAD_PRIORITY	Thread priority.	
	Range:	Priority class:
	1 - 6	Idle.
	7 - 10	Normal.
	11 - 15	High.
	16, 22-26, 31	Real-time.
M_VIEW_BIT_SHIFT	The number of bits by which to shift when M_VIEW_MODE is set to M_BIT_SHIFT. Should be set to the number of significant bits in the buffer minus 8. For example, if a 16-bit buffer contains data grabbed from a 10-bit digitizer, a shift of 2 should be used.	

ControlType	Description and ControlValue	
M_VIEW_MODE	Controls how a buffer gets remapped to the display; especially useful when displaying a non 8-bit buffer.	
	M_BIT_SHIFT	Bit-shift the pixel values of the buffer by the specified number of bits upon updating the display. Specify the number of bits with M_VIEW_BIT_SHIFT.
	M_AUTO_SCALE	The pixel values are remapped to the display such that the minimum and maximum values in the image (not the full range of the buffer) are set to 0 and 255, respectively. If the image buffer contains a single value, its corresponding displayed value is determined by linearly re-mapping the full range of the buffer; for example, (0 to 64K) to (0 through 255). This control value is only available when using a windowed display.
	M_MULTI_BYTES	Display each byte of the buffer in separate display pixels. In other words, each pixel of a 16-bit buffer will occupy two consecutive display pixels. Each pixel of a 32-bit buffer will occupy four consecutive display pixels. This mode is primarily useful when grabbing from a multi-tap camera.
	M_DEFAULT	MIL automatically selects the appropriate mode, depending on the buffer depth: 1-bit M_BIT_SHIFT (0 shift), 8-bit M_BIT_SHIFT (0 shift), 16-bit M_AUTO_SCALE, 32-bit M_AUTO_SCALE, 32-bit float M_AUTO_SCALE
M_WINDOW_BUF_WRITE	Allow direct access (destructive annotation) to the copy of the buffer stored in the frame buffer, after an MdispSelect() operation: M_ENABLE or M_DISABLE (default). If enabled, the MIL identifier of this buffer can be inquired, using MdispInquire() . If disabled, the buffer is invalid.	
M_WINDOW_COLOR	Force a window update to fill with a constant background color rather than with the selected buffer: M_ENABLE or M_DISABLE.	
M_WINDOW_COLOR_CHANGE	Set a background color, in Windows' COLORREF format. It is used when M_WINDOW_COLOR is enabled.	
M_WINDOW_INITIAL_POSITION_X	Set the window client area's initial left-most X-coordinate.	
M_WINDOW_INITIAL_POSITION_Y	Set the window client area's initial topmost Y-coordinate.	

ControlType	Description and ControlValue
M_WINDOW_KEYBOARD_USE	Activate the keys associated with the display window: M_ENABLE (default) or M_DISABLE. <i>The default key usage is:</i>
	+ Increase the X and Y zoom factors.
	- Decrease the X and Y zoom factors.
	Pg-up Scroll the buffer up to the previous display section.
	Pg-dn Scroll the buffer down to the next display section.
	Up arrow Scroll the buffer up to the previous line.
	Dn arrow Scroll the buffer down to the next line.
	Left arrow Pan the buffer left by one pixel.
	Right arrow Pan the buffer right by one pixel.
	Ctrl Up arrow Scroll the buffer up to the previous display section.
M_WINDOW_MAXBUTTON	Make the window's maximize button visible: M_ENABLE or M_DISABLE.
M_WINDOW_MENU_BAR	Make the window's menu bar visible: M_ENABLE or M_DISABLE.
M_WINDOW_MENU_BAR_CHANGE	Allow toggling the menu bar presence: M_ENABLE or M_DISABLE.
M_WINDOW_MINBUTTON	Make the window's minimize button visible: M_ENABLE or M_DISABLE.
M_WINDOW_MOVE	Allow window movement: M_ENABLE or M_DISABLE
M_WINDOW_OVERLAP	Allow window to be overlapped by another: M_ENABLE or M_DISABLE (keep window on top).
M_WINDOW_OVR_DESTRUCTIVE	The overlay shown on top of the buffer is allowed to overwrite the buffer's content (to increase display speed or save memory): M_ENABLE or M_DISABLE (default).
M_WINDOW_OVR_FLICKER	The overlay shown on top of the buffer is allowed some flicker (to increase display speed or save memory): M_ENABLE or M_DISABLE (default).

ControlType	Description and ControlValue	
M_WINDOW_PALETTE_NOCOLLAPSE	M_ENABLE	The Windows palette manager attempts the best color usage of the logical palette when realizing the output LUTs. It tries to map colors from the logical palette into the currently-realized output LUTs to reduce the number of requested new entries.
	M_DISABLE (default)	The Windows palette manager loads each component of the logical palette directly “as is” in the corresponding output LUT. This can result in a color occurring more than once in the output LUTs.
M_WINDOW_RANGE	Inform the display that the displayed buffer values will be restricted to between 10 and 245. This allows the optimization of display updates. M_ENABLE or M_DISABLE (default).	
M_WINDOW_RESIZE	Allow window resizing: M_ENABLE (or M_NORMAL_SIZE), M_DISABLE, or M_FULL_SIZE (to force a full-size display).	
M_WINDOW_SCROLLBAR	Make the window's scroll bars visible: M_ENABLE or M_DISABLE.	
M_WINDOW_SNAP_X	Restrict the left-most X-coordinate of window client area to a given multiple of the screen's absolute coordinate. Permissible values are positive or negative integers. Positive snap values adjust the X-coordinate to the closest right pixel; negative ones adjust it to the closest left pixel.	
M_WINDOW_SNAP_Y	Restrict the topmost Y-coordinate of the window client area to a given multiple of the screen's absolute coordinate. Permissible values are positive or negative integers. Positive snap values adjust the Y-coordinate to the closest upper pixel; negative ones adjust it to the closest lower pixel.	
M_WINDOW_SYSBUTTON	Make the window's system button visible: M_ENABLE or M_DISABLE.	
M_WINDOW_TITLE_BAR	Make the window's title bar visible: M_ENABLE or M_DISABLE.	
M_WINDOW_TITLE_BAR_CHANGE	Allow toggling the title bar presence: M_ENABLE or M_DISABLE.	
M_WINDOW_TITLE_NAME	Set the display window title to a specified string (the string must be cast to long).	
M_UPDATE	Allow display updates by MIL when a buffer is modified: M_ENABLE or M_DISABLE. This control type can be used to temporarily disable the display updates in a display when its buffer is being modified by more than one operation; the display can then be updated at end of all operations only.	

ControlType	Description and ControlValue	
M_WINDOW_UPDATE_ON_PAINT	M_ENABLE	Update the display on reception of a WM_PAINT message in Windows.
	M_DISABLE	Update the display on reception of a WM_ERASEBKGND message in Windows.
	M_DEFAULT	Allow MIL to decide which message to receive before updating the display.
M_WINDOW_ZOOM	Allow window zooming: M_ENABLE or M_DISABLE	
M_ZOOM_MAX_X	Set the upper limit in X of the window zooming ($1 \leq x \leq 16$).	
M_ZOOM_MAX_Y	Set the upper limit in Y of the window zooming ($1 \leq x \leq 16$).	
M_ZOOM_MIN_X	Set the lower limit in X of the window zooming ($1 \leq x \leq 16$).	
M_ZOOM_MIN_Y	Set the lower limit in Y of the window zooming ($1 \leq x \leq 16$).	
M_WINDOW_OVR_LUT	Associate a LUT with the overlay buffer (for both windowed and auxiliary displays). Set ControlValue to the LUT buffer's identifier.	
M_WINDOW_OVR_SHOW	Show the overlay buffer: M_ENABLE (default) or M_DISABLE.	
M_WINDOW_OVR_WRITE	Allow annotating the displayed image non-destructively using MIL's overlay-display mechanism. When enabled, the display is associated with a temporary overlay buffer, which can be used to annotate the underlying image with an effect called keying, which makes portions of the overlay show through. The overlay buffer is the size of the image buffer selected to the display. If another image buffer is selected to the display, and this image buffer has different dimensions than the one it is replacing, a new overlay buffer is created and the content of the old overlay buffer is copied into the new one; otherwise, the overlay buffer remains the same (annotations are not cleared). The MIL identifier of the overlay buffer can be inquired, using MdispInquire() . If your display is CPU-assisted, the overlay effect is produced by software:	
	M_ENABLE	Enable MIL's overlay-display mechanism.
	M_DISABLE	Disable MIL's overlay-display mechanism (default).

Example mdispovr.c

See also **MdispInquire()**

MdispDeselect

Synopsis Stop displaying an image buffer.

Format void MdispDeselect(DisplayId, ImageBufId)

MIL_ID DisplayId;	Display identifier
MIL_ID ImageBufId;	Image buffer identifier

Description This function stops displaying the specified image buffer on the specified display. For windowed displays, the display is closed. For auxiliary displays, the display is blanked.

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.

It is not necessary to use **MdispDeselect()** before selecting another buffer for display; just use **MdispSelect()**.

The **DisplayId** parameter specifies the identifier of the display from which to remove the image buffer.

The **ImageBufId** parameter specifies the identifier of the buffer to remove from the display. This buffer must be a currently displayed image buffer, with an M_DISP attribute.

See also MdispSelect()

MdispFree

Synopsis Free a display.

Format **void MdispFree(DisplayId)**

MIL_ID DisplayId;	Display identifier
-------------------	--------------------

Description This function deallocates a display previously allocated with **MdispAlloc()**.

The **DisplayId** parameter specifies the identifier of the display.

See also **MdispAlloc()**, **MappFreeDefault()**

MdispHookFunction

Synopsis Hook a function to a display event.

Format **MDISPHOOKFCTPTR MdispHookFunction(DisplayId, HookType, HookHandlerPtr, UserDataPtr)**

MIL_ID DisplayId;	Display identifier
long HookType;	Type of event to hook
MDISPHOOKFCTPTR HookHandlerPtr;	Pointer to hook function
void MPTYPE *UserDataPtr;	User data pointer

Description This function allows you to attach or detach a user-defined function to a specified display event. Once a hook-handler function is defined and hooked to an event, it is automatically called when the event occurs.

Note that functions hooked to an event execute on a distinct thread. This permits the functions to run asynchronously from the operation that fired the event and from functions hooked to other events. Although there is a small queue to permit a certain amount of overlap, hooked functions should not take longer to execute than the period in which two of their associated events can occur. You cannot determine the instance of the event that fired the function, and even if this were possible, this information would generally not be very useful. Typically, a hooked function performs the minimum number of operations required and, if necessary, performs longer processes by launching other threads.

You can hook more than one function to an event by making separate calls to **MdispHookFunction()** for each function that you want to hook. MIL automatically chains and keeps an internal list of all these hooked functions. When a function is hooked, this new function is added to the end of the list. When the event happens, all user-defined functions in the list will be executed in the same order that they were hooked to the event. You can also remove any function from the list; in this case, MIL preserves the order of the remaining functions in the list. The **DisplayId** parameter specifies the identifier of the target display for the hook.

The **HookType** parameter specifies the display event type. This parameter can be set to the following. Note that a hooked function must be unhooked by combining the **HookType** parameter with M_UNHOOK.

M_FRAME_START	Call the hook-handler function each time a new frame is displayed. You can only hook to this event if DirectDraw is enabled and you are using a windowed display.
---------------	---

The **HookHandlerPtr** parameter specifies the address of the function that should be called when an event occurs.

The hook-handler function, pointed to by **HookHandlerPtr**, must be declared as follows:

long MFTYPE HookHandler(HookType, EventId, UserDataPtr);	
long HookType;	Type of event hooked
MIL_ID EventId;	Reserved for future use
void MPTYPE *UserDataPtr;	Pointer that was passed by MdispHookFunction()

Upon successful completion, the hook-handler function should return M_NULL. Note, MDISPHOOKFCTPTR, MFTYPE and MPTYPE are reserved MIL predefined types for functions and data pointers.

The **UserDataPtr** parameter specifies the address of the user data that you want to make available to the hook-handler function. This address is passed to the hook-handler function, through its *UserDataPtr* parameter, when the specified event occurs. Set this parameter to M_NULL if not used.

Return value The original prototype structure of this function has been kept for backwards compatibility. However, because of the current chaining method, the function always returns null.

See also **MdispControl(), MdispInquire()**

MdispInquire

Synopsis Inquire about a display parameter setting.

Format **long MdispInquire(DisplayId, InquireType, UserVarPtr)**

MIL_ID DisplayId;	Display identifier
long InquireType;	Display parameter to inquire
void *UserVarPtr;	Storage location for inquired information

Description This function inquires about a specified display parameter setting.

The **DisplayId** parameter specifies the identifier of the display.

The **InquireType** parameter specifies the display parameter about which to inquire. This parameter can be set to one of the following values:

InquireType	Description
M_FORMAT	Display data format (MdispAlloc()).
M_FORMAT_SIZE	Number of characters in the data format string (MdispAlloc()).
M_FRAME_START_HANDLER_PTR	Handler pointer hooked using MdispHookFunction() to the start of a displayed frame (MdispSelect()).
M_FRAME_START_HANDLER_USER_PTR	User pointer hooked using MdispHookFunction() to the start of a displayed frame (MdispSelect()).
M_INIT_FLAG	Display initialization flag (MdispAlloc()).
M_KEY_COLOR	Keying color (MdispOverlayKey()).
M_KEY_CONDITION	Keying condition (MdispOverlayKey()).
M_KEY_MASK	Keying mask (MdispOverlayKey()).
M_KEY_MODE	State of keying mode (MdispOverlayKey()).
M_KEY_SUPPORTED	Whether overlay keying is supported by hardware (M_YES or M_NO).
M_LUT_ID	The identifier of the LUT associated with the display.
M_LUT_SUPPORTED	Whether a LUT is supported on the specified display.
M_NUMBER	Display rank in the system (MdispAlloc()).
M_DISPLAY_MODE	Display mode. M_WINDOWED if the display is bounded by a movable frame, or M_NON_WINDOWED if there is no frame.
M_OWNER_SYSTEM	The identifier of the system on which the display has been allocated (MdispAlloc()).
M_PAN_X	Pan X pixel offset (MdispPan()).

InquireType	Description
M_PAN_Y	Pan Y pixel offset (MdispPan()).
M_SELECTED	The identifier of the image buffer currently displayed. M_NULL is returned if no buffer is currently being displayed (MdispSelect()).
M_SIGN	Display data range (M_UNSIGNED).
M_SIZE_BAND	The number of color bands the display is capable of displaying. For windowed displays, 3 will be returned; for auxiliary displays, 1 or 3 will be returned.
M_SIZE_BAND_LUT	Number of color bands of the output LUT (if any) associated with the display.
M_SIZE_BIT	Number of bits (depth) of the display.
M_SIZE_X	Display width.
M_SIZE_Y	Display height.
M_TYPE	Display data type (number of bits + M_UNSIGNED).
M_VGA_PIXEL_FORMAT	Pixel format of the current VGA display resolution. Allocating a display buffer with the same format will ensure maximum performance with regard to display updates.
M_ZOOM_X	Zoom factor in X (MdispZoom()).
M_ZOOM_Y	Zoom factor in Y (MdispZoom()).
The following inquire types are only available with M_WINDOWED displays:	
M_THREAD_PRIORITY	Thread priority.
M_VIEW_BIT_SHIFT	The number of bits by which the buffer data gets shifted when M_VIEW_MODE is set to M_BIT_SHIFT.
M_VIEW_MODE	How a buffer gets remapped to the display: M_BIT_SHIFT, M_AUTO_SCALE, or M_MULTI_BYTES.
M_WINDOW_BUF_ID	Identifier of the copy of the buffer stored in the frame buffer (display memory) or M_NULL. On graphics controllers that do not have non-destructive overlay capabilities, this inquire type returns 0. In this case, M_WINDOW_OVR_BUF_ID should be used instead.
M_WINDOW_BUF_WRITE	Whether direct access to the copy of the buffer stored in the frame buffer is enabled (M_ENABLE or M_DISABLE).
M_WINDOW_CLIP_LIST	Window clip list pointer (LPRGNDATA).
M_WINDOW_CLIP_LIST_SIZE	Window clip list size to allocate.
M_WINDOW_COLOR	Force a constant background color (M_ENABLE or M_DISABLE).
M_WINDOW_COLOR_CHANGE	Current constant color.
M_WINDOW_DDRAW_SURFACE	Pointer to the DirectDraw primary surface (LPDIRECTDRAWSURFACE) used by a display window (if any) or M_NULL.

InquireType	Description
M_WINDOW_DIB_HEADER	Pointer to the header (LPBITMAPINFO) of the DIB buffer associated with the display window (if any) or M_NULL.
M_WINDOW_HANDLE	Windows handle (HWND) of the display window.
M_WINDOW_MAXBUTTON	Maximize button presence (M_ENABLE or M_DISABLE).
M_WINDOW_MENU_BAR	Menu bar presence (M_ENABLE or M_DISABLE).
M_WINDOW_MENU_BAR_CHANGE	State of menu bar changing (M_ENABLE or M_DISABLE).
M_WINDOW_MINBUTTON	Minimize button presence (M_ENABLE or M_DISABLE).
M_WINDOW_MOVE	State of display window moving (M_ENABLE or M_DISABLE).
M_WINDOW_OFFSET_X	Display window client area offset X, relative to the top left of the screen.
M_WINDOW_OFFSET_Y	Display window client area offset Y, relative to the top left of the screen.
M_WINDOW_OVERLAP	State of display window overlapping (M_ENABLE or M_DISABLE).
M_WINDOW_PALETTE_NOCOLLAPSE	Whether the Windows palette is forced to be non-collapsed: M_ENABLE or M_DISABLE.
M_WINDOW_PAN_X	Display window horizontal scroll bar position.
M_WINDOW_PAN_Y	Display window vertical scroll bar position.
M_WINDOW_RANGE	Inform the display that the displayed buffer values will be restricted to between 10 and 245. This allows the optimization of display update. M_ENABLE or M_DISABLE (default).
M_WINDOW_RESIZE	State of display window resizing (M_ENABLE, M_DISABLE, M_FULL_SIZE or M_NORMAL_SIZE).
M_WINDOW_SCROLLBAR	Scroll bar presence (M_ENABLE or M_DISABLE).
M_WINDOW_SIZE_X	Display window client area width.
M_WINDOW_SIZE_Y	Display window client area height.
M_WINDOW_SYSBUTTON	System button presence (M_ENABLE or M_DISABLE).
M_WINDOW_TITLE_BAR	Title bar presence (M_ENABLE or M_DISABLE).
M_WINDOW_TITLE_BAR_CHANGE	State of title bar changing (M_ENABLE or M_DISABLE).
M_WINDOW_TITLE_NAME	Window title string pointer.
M_WINDOW_TITLE_NAME_SIZE	Number of characters in the window's title string.
M_UPDATE	State of window update (M_ENABLE or M_DISABLE).
M_WINDOW_ZOOM	State of display window zooming (M_ENABLE or M_DISABLE).

InquireType	Description
M_WINDOW_ZOOM_X	Window zoom X factor (controlled by zoom buttons).
M_WINDOW_ZOOM_Y	Window zoom Y factor (controlled by zoom buttons).
M_WINDOW_OVR_BUF_ID	Identifier of the overlay buffer associated with the display or M_NULL.
M_WINDOW_OVR_DISP_ID	Identifier of the overlay display associated with the underlay display or M_NULL.
M_WINDOW_OVR_LUT	LUT associated with the overlay buffer of the display (only for windowed displays).
M_WINDOW_OVR_SHOW	Visible state of the overlay (M_ENABLE or M_DISABLE).
M_WINDOW_OVR_WRITE	Whether or not the overlay-display mechanism has been enabled. (M_ENABLE or M_DISABLE).

The **UserVarPtr** parameter specifies the address of the variable in which the requested information is to be written. If **MdispInquire()** also returns the requested information, you can set this parameter to M_NULL instead of passing the address of the variable.

This parameter should be a pointer to a long except when **InquireType** is set to one of the following:

- M_OWNER_SYSTEM, M_SELECTED, and M_LUT_ID, in which case it should be a pointer to a MIL_ID.
- M_FORMAT, in which case it should be a pointer to an array of MIL_TEXT_CHAR.

Return value Except for the M_FORMAT inquire type, the returned value is the setting of the requested display attribute, cast to long. For the M_FORMAT inquire type, the returned value is M_NULL.

See also **MdispAlloc()**, **MdispControl()**, **MdispSelect()**, **MdispPan()**, **MdispOverlayKey()**, **MdispZoom()**

MdispLut

Synopsis Associate a LUT buffer to a display.

Format `void MdispLut(DisplayId, LutBufId)`

MIL_ID DisplayId;	Display identifier
MIL_ID LutBufId;	LUT buffer identifier

Description This function associates a LUT buffer to the specified display. If and when the display is selected, the change required to produce the display (LUT) effect occurs. Note that physical output LUTs are not typically supported for auxiliary displays. For windowed displays, MIL indirectly programs the physical output LUTs through the use of a Windows palette. MIL checks the target display to determine whether or not a LUT is supported. If not, an error is generated. See *Chapter 18:Lookup tables* and *Chapter 19:Displaying an image* in the *MIL User Guide* for more details on using LUTs.

The **DisplayId** parameter specifies the identifier of the display to which the LUT buffer is copied.

The **LutBufId** parameter specifies the identifier of a previously allocated LUT buffer (with an M_LUT attribute). The LUT buffer can be the default LUT (M_DEFAULT), the pseudo LUT (M_PSEUDO), or a custom LUT buffer:

- The default LUT (M_DEFAULT)

If you set **LutBufId** to M_DEFAULT for windowed displays, MIL provides a good default logical palette for the realization of the physical output LUTs. MIL takes into consideration the number of bands of the image, and produces the best performance versus visual quality compromise possible.

- A pseudo-color LUT (M_PSEUDO)

If you set **LutBufId** to M_PSEUDO for windowed displays, the data is loaded in each component of the logical palette.

- A custom LUT buffer identifier

You can associate a custom LUT (allocated with **MbufAlloc1d0** or **MbufAllocColor0**) with the display by setting **LutBufId** to the LUT's buffer identifier (a buffer having the M_LUT attribute).

If you associate a one-band LUT buffer with a windowed display, and then select the display (**MdispSelect()**), the same data is loaded in each component of the logical palette.

If you associate a three-band color LUT buffer (RGB) for a windowed display, and then select the display, each band of the LUT buffer is loaded into its corresponding component of the logical palette.

Refer to both *Chapter 18: Look-up tables (LUTS)*, as well as *Chapter 19: Displaying an image* in the *MIL User Guide* for a detailed description of managing LUT buffers and achieving the appropriate display effect.

LUT buffers used for display have the following restrictions:

- If the LUT buffer values are changed while the image is selected on the display, the changes will not take effect until the next call is made to **MdispInquire()**. That is, the LUT is not automatically updated when the LUT buffer is modified.
- In general, the LUT buffer will not be used when displaying a 3-band 8-bit image under a non-8-bit display resolution.
- In general, LUT buffers are not supported with auxiliary displays.
- The LUT buffer must have one or three bands. Note that the number of LUT buffer entries must be the same as the maximum number of intensities that can be represented in the displayed buffer. In other words, if you want to invert an 8-bit grayscale image (that is, an image that can have 256 intensities), your LUT must also have 256 entries.

Note To obtain good results, the specified color values must be carefully selected to provide the best color match for displaying your image. If the specified values closely match the RGB values that occur frequently in the image to be displayed, very good results can be obtained.

Status Hardware limitations:

Most hardware systems do not support display LUTs for auxiliary displays.

See also **MbufAlloc1d()**, **MbufAllocColor()**, **MgenLutRamp()**, **MgenLutFunction()**, **MbufPut()**, **MbufPut1d()**

MdispOverlayKey

Synopsis Enable or disable overlay keying for the specified display.

Format `void MdispOverlayKey(DisplayId, KeyMode, KeyCond, KeyMask, KeyColor)`

MIL_ID DisplayId;	Display identifier
long KeyMode;	Mode for keying
long KeyCond;	Keying condition
long KeyMask;	Keying mask to apply before comparison
long KeyColor;	Keying color with which to compare

Description This function enables or disables overlay keying, an operation that makes portions of the overlay buffer transparent so that underlying areas of the displayable image show through. This function only has an effect when the MIL overlay-display mechanism is enabled with **MdispControl()**.

The **DisplayId** parameter specifies the identifier of the display.

The **KeyMode** parameter specifies the keying mode. It can be set to one of the following:

M_KEY_OFF	Display the display's overlay buffer only (no keying).
M_KEY_ON_COLOR	Display the image buffer selected on the display only where the pixels of the display's overlay buffer are equal to KeyColor .
M_KEY_ALWAYS	Display the image buffer selected on the display only.

The **KeyCond** parameter specifies the keying condition when keying is enabled. If keying is enabled (M_KEY_ON_COLOR), set this parameter to one of the following:

M_EQUAL	Display the image buffer where the overlay buffer's pixels equal the value of the KeyColor .
M_NOT_EQUAL	Display the image buffer where the overlay buffer's pixels do not equal the value of the KeyColor .

Otherwise, set the **KeyCond** to M_NULL.

The **KeyMask** parameter specifies the mask to apply to the overlay pixels, before performing the comparison and when keying is enabled (M_KEY_ON_COLOR). Only overlay pixel bits corresponding to enabled mask bits are compared with those of the keying color during the keying operation.

To compare overlay pixels to the specified **KeyColor** value, enable all bits in the mask (that is, set **KeyMask** to 0xffff).

To compare overlay pixels to a range of keying colors, enable the mask bits in the required range and specify an appropriate **KeyColor** value. When in an 8-bit display mode, pass an 8-bit value as the mask. When in any other display mode, pass an RGB24 packed value (that is, the least-significant byte corresponding to the red component, the next significant byte corresponding to the green component, and the most-significant byte corresponding to the blue component).

When keying is not enabled (M_KEY_OFF), set **KeyMask** to M_NULL.

Example The following portion of MIL code will display the main frame buffer when the overlay frame buffer color is equal to 10:

```
MdispOverlayKey(DisplayId, M_KEY_ON_COLOR, M_EQUAL, 0xffL, 10L)
```

MdispPan

Synopsis Pan and scroll a display.

Format `void MdispPan(DisplayId, XOffset, YOffset)`

MIL_ID DisplayId;	Display identifier
long XOffset;	X pixel offset relative to top-left corner of buffer
long YOffset;	Y pixel offset relative to top-left corner of buffer

Description This function associates pan and scroll values with the specified display. When an image buffer is selected for display, it will be panned and scrolled on the display according to these values.

The **DisplayId** parameter specifies the identifier of the display.

The **XOffset** and **YOffset** parameters specify the number of pixels by which to pan and scroll, respectively, an image buffer when it is displayed. Specify the pan and scroll in relation to the top-left corner of the image buffer. Specify a positive **XOffset** value to pan the image to the left, a positive **YOffset** value to scroll the image upwards.

Note, the offsets are in image pixels (not pixels), so they are not affected by the current zoom factor. For example, if the display has an associated zoom factor 4, panning by an offset of one image pixel results in panning by 4 on the display.

See also `MdispZoom()`, `MdispControl()`

MdispSelect

Synopsis Select an image buffer to display.

Format **void MdispSelect(DisplayId, ImageBufId)**

MIL_ID DisplayId;	Display identifier
MIL_ID ImageBufId;	Image buffer identifier

Description This function outputs the specified image buffer contents to the specified MIL display. You can only display one buffer at a time on a specific display.

The **DisplayId** parameter specifies the identifier of the display.

The **ImageBufId** parameter specifies the image buffer to display. To be displayable, this buffer must be an image buffer that has an M_IMAGE + M_DISP attribute.

If the specified image buffer is smaller in size than the display size, the border outside the image is blanked out (if the hardware supports this). If the specified buffer is larger in size than the system display, the right and bottom portion of the buffer, the part that exceeds the display size, is not displayed.

Note By default, under Windows, a call to **MdispSelect()** creates a window surrounding the image.

See also **MdispDeselect()**

MdispSelectWindow

Synopsis Select an image buffer to display in a user-defined window.

Format **void MdispSelectWindow(DisplayId, ImageBufId, ClientWindowHandle)**

MIL_ID DisplayId;	Display identifier
MIL_ID ImageBufId;	Image buffer identifier
HWND ClientWindowHandle;	User-defined window handle

Description This function displays the specified image buffer contents in the specified user window, using the specified MIL display.

This function is valid only in a Windows environment.

The **DisplayId** parameter specifies the identifier of the display.

The **ImageBufId** parameter specifies the image buffer to display. To be displayable, this buffer must be an image buffer that has an M_IMAGE + M_DISP attribute.

If the specified image buffer is smaller in size than the target window size, the border outside the image is not modified. If the specified buffer is larger in size than the target window, the right and bottom portion of the buffer, the part that exceeds the window, is not displayed.

The **ClientWindowHandle** parameter specifies the handle of the user-defined window or child window. This window must have been created with the Windows API functions. If this parameter is set to zero, this function behaves like **MdispSelect()**.

Example mwindisp.c

See also **MdispSelect()**, **MdispDeselect()**

MdispZoom

Synopsis Zoom a display.

Format **void MdispZoom(DisplayId, XFactor, YFactor)**

MIL_ID DisplayId;	Display identifier
long XFactor;	X zoom factor
long YFactor;	Y zoom factor

Description This function associates a zoom factor with the specified display. When an image buffer is selected for display, it will be zoomed according to this factor. The image buffer will be displayed starting from its top-left corner, unless it has been panned and/or scrolled, using **MdispPan()**.

The **DisplayId** parameter specifies the identifier of the display.

The **XFactor** and **YFactor** parameters specify the X and Y zoom factor, respectively. You can only zoom an image by integer factors; zoom factors between -16 and 16, inclusive (except 0), are supported.

Example `mmultdis.c`

See also **MdispPan()**, **MdispControl()**

MgenLutFunction

Synopsis Generate data into a LUT buffer using a specified standard mathematical function.

Format void MgenLutFunction(LutBufId, Func, a, b, c, StartIndex, StartXValue, EndIndex)

MIL_ID LutBufId;	LUT buffer identifier
long Func;	Function to use for calculations
double a;	Function constant a
double b;	Function constant b
double c;	Function constant c
long StartIndex;	First LUT index
double StartXValue;	Initial X value
long EndIndex;	Last LUT index

Description This function generates a value for each LUT index within the specified index range (**StartIndex** to **EndIndex** inclusive), according to the specified mathematical function. Each function takes a value X. The **StartXValue** parameter specifies the initial X value. The remaining entries of the index range are generated by incrementing the value of X by 1 for each index.

The **LutBufId** parameter specifies the identifier of the LUT in which to generate values. This parameter must be given a valid LUT buffer identifier. Allocate a LUT buffer, using **MbufAlloc1d()** or **MbufAllocColor()**. If the LUT is a multi-band LUT (allocated with **MbufAllocColor()**), the same data is written to all bands.

The **Func** parameter specifies the mathematical function to use for calculations. This parameter can be set to one of the following:

M_LOG	$a \log_b(x) + c$
M_EXP	$a b^x + c$
M_SIN	$a \sin(bx) + c$
M_COS	$a \cos(bx) + c$
M_TAN	$a \tan(bx) + c$
M_QUAD	$ax^2 + bx + c$

The **a**, **b**, **c** parameters specify the function constants. For M_SIN, M_COS, and M_TAN, X is considered to be in degrees. All results are converted to integer by truncation, except when using a floating-point LUT buffer. Note, if the given parameters cause an overflow or underflow, indeterminate results will be written in the destination LUT.

The **StartIndex** and **EndIndex** specify the first and last LUT index entries for which to generate values. The **StartIndex** value must be less than or equal to the **EndIndex** value.

The **StartXValue** parameter specifies the initial value of X in the function.

See also **MgenLutRamp()**, **MbufPut1d()**, **MbufPutColor()**, **MbufAlloc1d()**, **MbufAllocColor()**.

MgenLutRamp

Synopsis Generate ramp data into a LUT buffer.

Format `void MgenLutRamp(LutId, StartIndex, StartValue, EndIndex, EndValue)`

MIL_ID LutId;	LUT identifier
long StartIndex;	First LUT index
double StartValue;	Start value of input range
long EndIndex;	Last LUT index
double EndValue;	End value of input range

Description This function generates a ramp, inverse ramp, or a constant in the specified LUT buffer region (**StartIndex** to **EndIndex**). The increment between LUT entries is the difference between **StartValue** and **EndValue**, divided by the number of entries.

If you need to generate a more complex LUT, use **MgenLutFunction()** or generate the values with your Host system and load them into a MIL LUT buffer, using **MbufPut1d()** or **MbufPutColor()**.

The **LutId** parameter specifies the identifier of the LUT in which to generate values. This parameter must be given a valid LUT buffer identifier. Allocate a LUT buffer, using **MbufAlloc1d()** or **MbufAllocColor()**.

The **StartIndex** and **EndIndex** parameters specify the first and last LUT index entry for which to generate values. **StartIndex** must be less than or equal to **EndIndex**.

The **StartValue** and **EndValue** parameters specify the extreme values from which the increment is calculated. **StartValue** is the first LUT entry. If both values are the same, the entire LUT range is filled with this value. If **EndValue** is smaller than **StartValue**, an inverse ramp is generated. These parameters accept only integer values, except when using a floating-point LUT buffer.

Examples `mdispovr.c`, `mnatfct.c`

See also **MgenLutFunction()**, **MbufPut1d()**, **MbufPutColor()**, **MbufAlloc1d()**, **MbufAllocColor()**

MgenWarpParameter

Synopsis Generate coefficients or LUTs for use with **MimWarp()**.

Format **void MgenWarpParameter(InWarpParameter, OutXLutOrCoef, OutYLut, OperationMode, Transform, Val1, Val2)**

MIL_ID InWarpParameter;	Input buffer or M_NULL
MIL_ID OutXLutOrCoef;	Output buffer for coefficients or x-LUT entries
MIL_ID OutYLut;	Output buffer for y-LUT entries or M_NULL
long OperationMode;	Operation mode
long Transform;	Type of first-order polynomial warping or M_DEFAULT
double Val1;	Constant or M_NULL
double Val2;	Constant or M_NULL

Description This function can generate:

- Coefficients (or LUTs) for a first-order polynomial warping (M_WARP_POLYNOMIAL **OperationMode**).
- Coefficients (or LUTs) for perspective warpings that map an arbitrary quadrilateral onto a rectangle (M_WARP_4_CORNER **OperationMode**) or that map a rectangle onto an arbitrary quadrilateral (M_WARP_4_CORNER_REVERSE **OperationMode**).
- Look-up tables (LUTs) for 3x3 matrix-defined warpings (M_WARP_LUT **OperationMode**).

For a first-order polynomial warping, coefficients can be generated for a rotation, a scaling, a shearing, or a translation. To combine coefficients (for example, to generate coefficients for a rotation and translation), you need to use separate calls to this function, using the previous output buffer as your current input buffer. After all coefficients are generated, pass the coefficient buffer to **MimWarp()**.

Note that you can perform a first-order polynomial warping using LUTs. To generate the required LUTs, use LUT buffers for your output rather than a coefficient buffer while generating the last set of coefficients. In this case, the coefficients will not be saved. If you need to save the coefficients, use **MgenWarpParameter()** in M_WARP_LUT mode after you have generated all coefficients.

After generating coefficients for perspective warpings, you need to call **MgenWarpParameter()** in M_WARP_LUT mode, in order to generate the LUTs required by **MimWarp()**. Alternatively, you can have the LUTs generated on the first call, by using LUT buffers for your output rather than a coefficient buffer. In this case, the coefficients will not be saved.

A 3x3 matrix-defined warping is performed by associating each pixel position of the destination buffer, (x_d, y_d) , with a specific point in the source

buffer, (x_s, y_s) , according to the following equation:

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

where

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

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

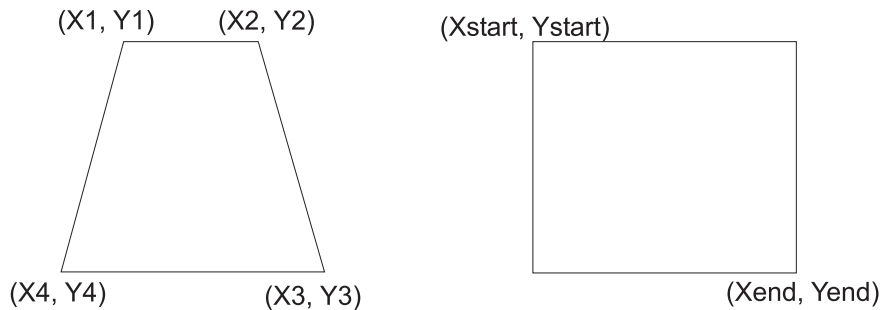
To perform a 3x3 matrix-defined warping, you must supply the 3x3 coefficients $(a_0...a_2, b_0...b_2, c_0...c_2)$ to **MgenWarpParameter()**, which will generate the LUTs required by **MimWarp()** to perform the warping.

The **InWarpParameter** parameter specifies the input buffer from which to generate coefficients or LUT entries.

When generating coefficients for a first-order polynomial warping, set **InWarpParameter** to M_NULL if this is your first call to **MgenWarpParameter()**. If you are combining coefficients and this is a second (or later) call, set **InWarpParameter** to the identifier of the previous output buffer.

When generating coefficients for perspective warpings that map an arbitrary quadrilateral onto a rectangle or that map a rectangle onto an arbitrary quadrilateral, **InWarpParameter** must be

set to the identifier of a one-dimensional floating-point buffer with an M_ARRAY attribute. The buffer must contain 12 entries; each entry corresponds to one of the following points.



entry [0] = X1	entry [8] = Xstart
entry [1] = Y1	entry [9] = Ystart
entry [2] = X2	entry [10] = Xend
entry [3] = Y2	entry [11] = Yend
entry [4] = X3	
entry [5] = Y3	
entry [6] = X4	
entry [7] = Y4	

When generating LUTs for a 3x3 matrix-defined warping, **InWarpParameter** must be set to the identifier of a 3x3 floating-point buffer that has an M_ARRAY attribute. The first row specifies the a_n coefficients, the second row specifies the b_n coefficients, and the third row specifies the c_n coefficients.

The **OutXLutOrCoef** parameter specifies the buffer in which to place generated coefficients or x-LUT entries.

For coefficients (either those for a polynomial warping or a perspective warping), the buffer must be a 3x3 floating-point buffer with an M_ARRAY attribute.

For x-LUT entries, the buffer must be signed 16- or 32-bit integer, have the same x and y size as the destination buffer that you will eventually pass to **MimWarp0**, and have a M_LUT attribute.

The **OutYLut** parameter specifies the buffer in which to place y-LUT entries. This buffer must be signed 16- or 32-bit integer, have the same x and y size as the destination buffer that you will eventually pass to **MimWarp0**, and have a M_LUT attribute.

If you are not generating LUTs, set the **OutYLut** parameter to M_NULL.

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

M_WARP_POLYNOMIAL	Generate coefficients for a first-order polynomial warping.
M_WARP_4_CORNER	Generate coefficients for a perspective warping that maps an arbitrary quadrilateral onto a rectangle.
M_WARP_4_CORNER_REVERSE	Generate coefficients for a perspective warping that maps a rectangle onto an arbitrary quadrilateral.
M_WARP_LUT	Generate LUTs for a 3x3 matrix-defined warping.

When using LUT buffers for your output, you can add M_FIXED_POINT + n to the **OperationMode** parameter, to specify the number of fractional bits for the source address (x_s , y_s). The default value is 0.

The **Transform** parameter specifies the type of first-order polynomial warping for which to generate coefficients. It can be set to:

M_ROTATE	Generate coefficients for a counter-clockwise rotation around (0,0) by Val1 °.
M_SCALE	Generate coefficients for an image scaling, by a factor of Val1 in the x direction and by a factor of Val2 in the y direction.
M_SHEAR_X	Generate coefficients for a shearing in the x direction, by a factor of Val1 .
M_SHEAR_Y	Generate coefficients for a shearing in the y direction, by a factor of Val1 .
M_TRANSLATE	Generate coefficients for a translation by Val1 pixels in the x direction and by Val2 pixels in the y direction.

If you are not generating coefficients for a first-order polynomial warping, set the **Transform** parameter to M_DEFAULT.

The **Val1** and **Val2** parameters specify transform constants. If these parameters are not being used, set them to M_NULL.

Example mwarp.c

MgraAlloc

Synopsis Allocate a graphics context.

Format **MIL_ID MgraAlloc(SystemId, GraphContIdPtr)**

MIL_ID SystemId;	System identifier
MIL_ID *GraphContIdPtr;	Storage location for graphics context identifier

Description This function allocates a graphics context, which specifies drawing and text parameters for use in subsequent MIL graphic functions.

Upon allocation of a graphics context, the drawing and text parameters are set to the following default values:

Foreground color	0xFFFFFFFF
Background color	0x00000000
Font	M_FONT_DEFAULT_SMALL
Font scale	X = 1.0, Y = 1.0

You can modify these values, using **MgraColor()**, **MgraBackColor()**, **MgraFont()**, and **MgraFontScale()**, or inquire about the current values, using **MgraInquire()**.

You can set the attributes of the graphic context (for example, background transparency), using **MgraControl()**.

When a graphics context is no longer required, release it, using **MgraFree()**.

The **SystemId** parameter specifies the system on which the graphics context will be allocated. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. Specify M_DEFAULT_HOST to allocate on the default Host system of the current MIL application. Specify M_DEFAULT to have MIL select the most appropriate system on which to allocate the graphics context (it can be the default Host system or any already allocated system).

The **GraphContIdPtr** parameter specifies the address of the variable in which the graphics context identifier is to be written. Since the **MgraAlloc()** function also returns the buffer identifier, you can set this parameter to M_NULL. If allocation fails, M_NULL is written as the identifier.

Note, upon allocation of an application, a default graphics context is automatically allocated. Rather than using **MgraAlloc()** to allocate a graphics context, you can use this default graphics context, by specifying `M_DEFAULT` wherever a graphics context identifier is required.

Return value The returned value is the graphics context identifier. If allocation fails, `M_NULL` is returned.

See also **MgraFree()**, **MgraColor()**, **MgraBackColor()**, **MgraFont()**, **MgraFontScale()**, **MgraInquire()**

MgraArc

Synopsis Draw an arc.

Format `void MgraArc(GraphContId, DestImageBufId, XCenter, YCenter, XRad, YRad, StartAngle, EndAngle)`

MIL_ID GraphContId;	Graphics context identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long XCenter;	X-coordinate of arc center
long YCenter;	Y-coordinate of arc center
long XRad;	Horizontal radius of elliptic arc
long YRad;	Vertical radius of elliptic arc
double StartAngle;	Starting angle relative to the positive X-axis
double EndAngle;	Ending angle relative to the positive X-axis

Description This function draws an elliptic arc based on an ellipse centered at (**XCenter**, **YCenter**) with radii **XRad** and **YRad**. The arc is defined by the start angle **StartAngle** and the end angle **EndAngle**. The arc is drawn with the foreground color specified in the graphics context.

The **GraphContId** parameter specifies the identifier of the graphics context. This parameter can be set to M_DEFAULT, in which case the default graphics context of the current MIL application is used.

The **DestImageBufId** parameter specifies the identifier of the image buffer in which to draw.

The **XCenter** and **YCenter** parameters specify the X and Y coordinates of the arc center, relative to the top-left corner of the specified target buffer.

The **XRad** and **YRad** parameter specify the elliptic arc radii. The radii should be given in pixels and must be greater than 0.

The **StartAngle** and **EndAngle** specify the angles at which to start and end drawing the arc, respectively, moving in a counter-clockwise direction. Express angles in degrees in relation to the positive X-axis.

If part of the arc falls outside of the specified target buffer, that part is clipped off.

Examples mfft.c, mmeas.c

See also `MgraArcFill()`

MgraArcFill

Synopsis Draw a filled elliptic arc.

Format **void MgraArcFill(GraphContId, DestImageBufId, XCenter, YCenter, XRad, YRad, StartAngle, EndAngle)**

MIL_ID GraphContId;	Graphics context identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long XCenter;	X-coordinate of arc center
long YCenter;	Y-coordinate of arc center
long XRad;	Horizontal radius of elliptic arc
long YRad;	Vertical radius of elliptic arc
double StartAngle;	Starting angle relative to the positive X-axis
double EndAngle;	Ending angle relative to the positive X-axis

Description This function draws a filled elliptic arc based on an ellipse centered at (**XCenter**, **YCenter**) with radii **XRad** and **YRad**. The arc is defined by the start angle **StartAngle** and end angle **EndAngle**. The arc is drawn and filled with the foreground color specified in the graphics context.

The **GraphContId** parameter specifies the identifier of the graphics context. This parameter can be set to M_DEFAULT, in which case the default graphics context of the current MIL application will be used.

The **DestImageBufId** parameter specifies the identifier of the image buffer in which to draw.

The **XCenter** and **YCenter** parameters specify the X and Y coordinates of the arc center relative to the top-left corner of the specified target buffer.

The **XRad** and **YRad** parameters specify the elliptic arc radii. The radii should be given in pixels and must be greater than 0.

The **StartAngle** and **EndAngle** specify the angles at which to start and end drawing the arc, respectively, moving in a counter-clockwise direction. Express angles in degrees in relation to the positive X-axis.

If part of the arc falls outside of the specified target buffer, that part is clipped off.

Example mdisplay.c

See also **MgraArc()**, **MgraFill()**

MgraBackColor

Synopsis Sets the background color of a graphics context.

Format `void MgraBackColor(GraphContId, BackgroundColor)`

MIL_ID GraphContId;	Graphics context identifier
double BackgroundColor;	Background drawing and text color

Description This function sets the background color of a specified graphics context.

The **GraphContId** parameter specifies the identifier of the graphics context with which to associate the background color. This parameter can be set to `M_DEFAULT`, in which case the default graphics context of the current MIL application is used.

The **BackgroundColor** parameter specifies the background color. Set this parameter as follows:

- When using the graphics context to draw in a 1-band buffer, set this parameter to any value. This value will be cast to the type of the destination buffer.
- When using the graphics context to draw in a multi-band buffer with a grayscale background value, set this parameter to any value. This value will be cast to the type of the destination buffer's bands and replicated in each band.
- When using the graphics context to draw in an 8-bit 3-band buffer with an RGB background value, set this parameter using the following macro:

`M_RGB888(red component, green component, blue component)`

- When using the graphics context to draw in a 16-bit or 32-bit multi-band buffer with a color background value, use **MgraControl()**.

Example `mcode.c`

See also `MgraColor()`, `MgraAlloc()`, `MgraInquire()`, `MgraControl()`

MgraClear

Synopsis Clear an image buffer to a specified foreground color.

Format **void MgraClear(GraphContId, DestImageBufId)**

MIL_ID GraphContId;	Graphics context identifier
MIL_ID DestImageBufId;	Destination image buffer identifier

Description This function clears the entire specified buffer to the foreground color specified in the graphics context.

The **GraphContId** parameter specifies the identifier of the graphics context. This parameter can be set to M_DEFAULT, in which case the default graphics context of the current MIL application is used.

The **DestImageBufId** parameter specifies the identifier of the buffer to clear. This parameter must be given a valid image buffer identifier.

See also **MgraColor()**

MgraColor

Synopsis Sets the foreground color of a graphics context.

Format `void MgraColor(GraphContId, ForegroundColor)`

MIL_ID GraphContId;	Graphics context identifier
double ForegroundColor;	Foreground drawing and text color

Description This function sets the foreground color of a specified graphics context.

The **GraphContId** parameter specifies the identifier of the graphics context with which to associate the foreground color. This parameter can be set to `M_DEFAULT`, in which case the default graphics context of the current MIL application is used.

The **ForegroundColor** parameter specifies the foreground color. Set this parameter as follows:

- When using the graphics context to draw in a 1-band buffer, set this parameter to any value. This value will be cast to the type of the destination buffer.
- When using the graphics context to draw in a multi-band buffer with a grayscale foreground value, set this parameter to any value. This value will be cast to the type of the destination buffer's bands and replicated in each band.
- When using the graphics context to draw in an 8-bit 3-band buffer with an RGB foreground value, set this parameter using the following macro:

`M_RGB888(red component, green component, blue component)`

- When using the graphics context to draw in a 16-bit or 32-bit multi-band buffer with a color foreground value, use **MgraControl()**.

Examples `mblob.c`, `mcalib.c`, `mcode.c`, `mdisplay.c`, `mmeas.c`, `mmeasmul.c`, `mocrread.c`, `mwarpc.c`

See also **MgraBackColor()**, **MgraAlloc()**, **MgraInquire()**, **MgraControl()**

MgraControl

Synopsis Control the specified graphic context.

Format `void MgraControl(GraphContId, ControlType, ControlValue)`

MIL_ID GraphContId;	Graphic context identifier
long ControlType;	Control type
double ControlValue;	Control value

Description This function allows you to set the attributes of a graphic context.

The **GraphContId** parameter specifies the identifier of the graphic context (**MgraAlloc()**). To control the default graphic context of the current MIL application, set this parameter to M_DEFAULT.

The **ControlType** and **ControlValue** parameters specify the graphic features to control and the values needed for the control. These two parameters can be set to one of the following combinations:

ControlType	Description & ControlValue	
M_BACKGROUND_MODE	Controls the setting of the background color on the drawing surface.	
	M_OPAQUE	Fill background with the current background color before drawing text. This is the default value (M_DEFAULT).
	M_TRANSPARENT	Do not change background before drawing text. This creates a transparent background for printed characters.
M_COLOR	Sets the foreground color of a specified graphics context.	
M_BACKCOLOR	Sets the background color of a specified graphics context.	

For M_COLOR and M_BACKCOLOR, specify a **ControlValue** as follows:

- When using the graphics context to draw in a 1-band buffer, set **ControlValue** to any value. This value will be cast to the type of the destination buffer.

- To specify a grayscale value when using the graphics context to draw in a multi-band buffer, set **ControlValue** to any value. This value will be cast to the type of the destination buffer's bands and replicated in each band.
- To specify an RGB value when using the graphics context to draw in an 8-bit 3-band buffer, set **ControlValue** using the following macro:

```
M_RGB888(red component, green component, blue component)
```

- To specify a color value when using the graphics context to draw in a 16-bit or 32-bit multi-band buffer, you must call `MgraControl()` for each color component (R,G, and B). Add `M_RED`, `M_GREEN`, or `M_BLUE` to `M_COLOR` or `M_BACKCOLOR` to specify the component. Set **ControlValue** to any value; this value will be cast to the type of the destination buffer's bands. For example, you would make the following call to set the red color component:

```
MgraControl(M_DEFAULT, M_COLOR+M_RED, red color component)
```

Note that you can use the `M_RED`, `M_GREEN`, and `M_BLUE` constants even when using the graphics context to draw in an 8-bit multi-band buffer.

Examples `mcalib.c`, `mdispovr.c`

See also `MgraAlloc()`, `MgraBackColor()`, `MgraColor()`

MgraDot

Synopsis Draw a dot.

Format **void MgraDot(GraphContId, DestImageBufId, XPos, YPos)**

MIL_ID GraphContId;	Graphics context identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long XPos;	X position of dot
long YPos;	Y position of dot

Description This function draws a dot at the specified drawing position, using the foreground color specified in the graphics context.

The **GraphContId** parameter specifies the identifier of the graphics context. This parameter can be set to M_DEFAULT, in which case the default graphics context of the current MIL application is used.

The **DestImageBufId** parameter specifies the identifier of the buffer in which to draw. This parameter must be given a valid image buffer identifier.

The **XPos** and **YPos** parameters specify the X and Y coordinates of the drawing position. The given coordinate is relative to the top-left corner of the specified target buffer. It should be valid in the specified image buffer; otherwise, nothing will be drawn.

See also **MbufPut2d()**, **MbufPutColor()**

MgraFill

Synopsis Perform a boundary-type seed fill.

Format `void MgraFill(GraphContId, DestImageBufId, XStart, YStart)`

MIL_ID GraphContId;	Graphics context identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long XStart;	X-coordinate of seed position
long YStart;	Y-coordinate of seed position

Description This function performs a boundary-type seed fill. It fills in an area of the target buffer, with the foreground color specified in the graphics context, starting from the specified seed position. Filling occurs on adjacent pixels (vertically and horizontally to original seed pixel) that have the same value as the original seed pixel.

If the source buffer is a multi-band buffer, this function will process each band separately. This means that each band of the adjacent pixels will be compared with the corresponding band of the seed pixel. This can produce strange results if, for example, you try to fill the inside of a red circle with blue. The blue will spread to the whole image since the red circle does not exist in the blue band.

The **GraphContId** parameter specifies the identifier of the graphics context. This parameter can be set to `M_DEFAULT`, in which case the default graphics context of the current MIL application is used.

The **DestImageBufId** parameter specifies the identifier of the buffer in which to draw. This parameter must be given a valid image buffer identifier.

The **XStart** and **YStart** parameters specify the X and Y coordinates of the seed position. If the specified point is not within an enclosed area, filling occurs until the boundaries of the buffer are encountered. The given coordinate is relative to the top-left corner of the specified target buffer. It should be valid in the specified image buffer; otherwise, the operation is not performed.

See also `MgraArcFill()`, `MgraRectFill()`

MgraFont

Synopsis Associate a text font with a graphics context.

Format **void MgraFont(GraphContId, FontName)**

MIL_ID GraphContId;	Graphics context identifier
long FontName;	Character font

Description This function associates a character font with the specified graphics context for use with subsequent **MgraText()** function calls.

The **GraphContId** parameter specifies the identifier of the graphics context with which to associate the character font. This parameter can be set to M_DEFAULT, in which case, the default graphics context of the current MIL application is used.

The **FontName** parameter specifies the font with which to write text. This parameter can be set to one of the following:

FontName	Description
M_FONT_DEFAULT_LARGE	Default font with 16x32 pixel wide characters.
M_FONT_DEFAULT_MEDIUM	Default font with 12x24 pixel wide characters.
M_FONT_DEFAULT_SMALL	Default font with 8x16 pixel wide characters.
M_FONT_DEFAULT	In general corresponds to M_FONT_DEFAULT_SMALL.

Examples mocrfont.c, mocrread.c

See also **MgraFontScale()**, **MgraAlloc()**, **MgraText()**, **MgraInquire()**

MgraFontScale

Synopsis Set the font scale of a graphics context.

Format **void MgraFontScale(GraphContId, XFontScale, YFontScale)**

MIL_ID GraphContId;	Graphics context identifier
double XFontScale;	Font scaling factor in X
double YFontScale;	Font scaling factor in Y

Description This function sets the font scale of the specified graphics context for use with subsequent **MgraText()** function calls.

The **GraphContId** parameter specifies the identifier of the graphics context for which to set the font scale. This parameter can be set to **M_DEFAULT**, in which case the default graphics context of the current MIL application is used.

The **XFontScale** and **YFontScale** parameters are used to multiply the width and height of the font characters, respectively. Each of these parameters can be independently set to any positive floating point value. The default X and Y scale factors are 1.0.

Note, using a font with a scale of 1.0 accelerates text drawing.

Example mocrfont.c

See also **MgraFont()**, **MgraAlloc()**, **MgraText()**, **MgraInquire()**

MgraFree

Synopsis Free a graphics context.

Format **void MgraFree(GraphContId)**

MIL_ID GraphContId;	Graphics context identifier
---------------------	-----------------------------

Description This function deallocates a graphics context previously allocated with **MgraAlloc()**.

The **GraphContId** parameter specifies the identifier of the graphics context to deallocate. If M_DEFAULT is specified, an error will occur.

See also **MgraAlloc()**

MgraInquire

Synopsis Inquire about the graphics parameters.

Format `void MgraInquire(GraphContId, InquireType, UserVarPtr)`

<code>MIL_ID GraphContId;</code>	Graphics context identifier
<code>long InquireType;</code>	Graphic parameter to inquire
<code>void *UserVarPtr;</code>	Storage location for inquiry result

Description This function inquires about a graphic parameter in the specified graphics context.

The **GraphContId** parameter specifies the identifier of the graphics context on which to perform the inquiry. This parameter can be set to `MIL_DEFAULT`, in which case the default graphics context of the current MIL application is used.

The **InquireType** parameter specifies the graphic parameter about which to inquire. This parameter can be set to one of the following values:

InquireType	Description
<code>M_COLOR</code>	Foreground color.
<code>M_BACKCOLOR</code>	Background color.
<code>M_BACKGROUND_MODE</code>	Background mode.
<code>M_FONT</code>	Character font.
<code>M_FONT_X_SCALE</code>	Font scaling factor in X.
<code>M_FONT_Y_SCALE</code>	Font scaling factor in Y.
<code>M_OWNER_SYSTEM</code>	MIL identifier (<code>MIL_ID</code>) of the system on which the graphics context has been allocated (MgraAlloc()).

The **UserVarPtr** parameter specifies the address of the variable in which the requested information is to be written. This variable should be defined as follows:

InquireType	Pointer to a:
<code>M_COLOR</code>	double
<code>M_BACKCOLOR</code>	double
<code>M_BACKGROUND_MODE</code>	long
<code>M_FONT</code>	long

InquireType	Pointer to a:
M_FONT_X_SCALE	double
M_FONT_Y_SCALE	double
M_OWNER_SYSTEM	MIL_ID

To inquire the color value used in the graphics context for a 16-bit or 32-bit multi-band buffer, you must inquire each color component (R,G, and B) separately. Add M_RED, M_GREEN, or M_BLUE to M_COLOR or M_BACKCOLOR to specify the component. For example, you would make the following call to inquire the red color component:

```
MgralInquire(M_DEFAULT, M_COLOR+M_RED, &red color component)
```

See also **MgraColor()**, **MgraBackColor()**, **MgraFont()**, **MgraFontScale()**

MgraLine

Synopsis Draw a line.

Format `void MgraLine(GraphContId, DestImageBufId, XStart, YStart, XEnd, YEnd)`

MIL_ID GraphContId;	Graphics context identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long XStart;	X-coordinate of start of line position
long YStart;	Y-coordinate of start of line position
long XEnd;	X-coordinate of end of line position
long YEnd;	Y-coordinate of end of line position

Description This function draws a line starting and ending at the specified coordinates, using the foreground color specified in the graphics context.

The **GraphContId** parameter specifies the identifier of the graphics context. This parameter can be set to M_DEFAULT, in which case, the default graphics context of the current MIL application is used.

The **DestImageBufId** parameter specifies the identifier of the buffer in which to draw. This parameter must be given a valid image buffer identifier.

The **XStart** and **YStart** parameters specify the coordinates of one line extremity, while **XEnd** and **YEnd** specify the coordinates of the other. The given coordinates are relative to the top-left corner of the specified target buffer. They should be valid in the specified buffer; otherwise, the line is clipped outside the buffer boundaries.

Examples mblob.c, mcalib.c, mmeas.c, mmeasmul.c, mpatrot.c, mwarp.c

MgraRect

Synopsis Draw a rectangle.

Format **void MgraRect(GraphContId, DestImageBufId, XStart, YStart, XEnd, YEnd)**

MIL_ID GraphContId;	Graphics context identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long XStart;	X-coordinate of top-left rectangle corner
long YStart;	Y-coordinate of top-left rectangle corner
long XEnd;	X-coordinate of bottom-right rectangle corner
long YEnd;	Y-coordinate of bottom-right rectangle corner

Description This function draws a rectangle starting from the specified top-left coordinate to the specified bottom-right corner. The rectangle is drawn in the foreground color specified in the graphics context.

The **GraphContId** parameter specifies the identifier of the graphics context. This parameter can be set to M_DEFAULT, in which case the default graphics context of the current MIL application is used.

The **DestImageBufId** parameter specifies the identifier of the buffer in which to draw. This parameter must be given a valid image buffer identifier.

The **XStart** and **YStart** parameters specify the coordinates of the top-left corner of the rectangle, **XEnd** and **YEnd** specify the coordinates of the bottom-right corner. The given coordinates are relative to the top-left corner of the specified target buffer. They should be valid in the specified buffer; otherwise, the rectangle is clipped outside the buffer boundaries.

Examples mmeas.c, mmeasmul.c, mrestmod.c, msearch.c, mshift.c

See also **MgraRectFill()**

MgraRectFill

Synopsis Draw a filled rectangle.

Format `void MgraRectFill(GraphContId, DestImageBufId, XStart, YStart, XEnd, YEnd)`

MIL_ID GraphContId;	Graphics context identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long XStart;	X-coordinate of top-left rectangle corner
long YStart;	Y-coordinate of top-left rectangle corner
long XEnd;	X-coordinate of bottom-right rectangle corner
long YEnd;	Y-coordinate of bottom-right rectangle corner

Description This function draws a filled rectangle starting from the specified top-left coordinate to the specified bottom-right corner. The rectangle is drawn and filled in the foreground color specified in the graphics context.

The **GraphContId** parameter specifies the identifier of the graphics context. This parameter can be set to `M_DEFAULT`, in which case the default graphics context of the current MIL application is used.

The **DestImageBufId** parameter specifies the identifier of the buffer in which to draw. This parameter must be given a valid image buffer identifier.

The **XStart** and **YStart** parameters specify the coordinates of the top-left corner of the rectangle, **XEnd** and **YEnd** specify the coordinates of the bottom-right corner. The given coordinates are relative to the top-left corner of the specified target buffer. They should be valid in the specified buffer; otherwise, the rectangle is clipped outside the buffer boundaries.

See also `MgraRect()`, `MgraFill()`

MgraText

Synopsis Write text.

Format **void MgraText(GraphContId, DestImageBufId, XStart, YStart, String)**

MIL_ID GraphContId;	Graphics context identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long XStart;	X-coordinate of writing position
long YStart;	Y-coordinate of writing position
MIL_TEXT_PTR String;	Null terminated string

Description This function writes the specified string to the specified buffer starting at the specified writing position, using the parameters (colors, font, and size) defined in the graphics context. Use **MgraFont()** and **MgraFontScale()** to modify the font and size. Use **MgraControl()** to obtain a transparent background for printed characters.

The **GraphContId** parameter specifies the identifier of the graphics context. This parameter can be set to M_DEFAULT, in which case, the default graphics context of the current MIL application is used.

The **DestImageBufId** parameter specifies the identifier of the buffer in which to write. This parameter must be given a valid image buffer identifier.

The **XStart** and **YStart** parameters specify the coordinates of the position at which to start writing the top-left corner of the first character. The given coordinates are relative to the top-left corner of the buffer. They should be valid in the specified buffer; otherwise, the text is clipped.

The **String** parameter specifies the address of the string that must be written in the destination buffer. There is no restriction on the length of the string, except that the string must be null (\0) terminated.

Examples mcalib.c, mcode.c, mdispovr.c, mocrfont.c, mocrread.c, mstart.c, mthread.c, mwindisp.c

See also **MgraFont()**, **MgraFontScale()**, **MgraControl()**

MimAllocResult

Synopsis Allocate an image processing result buffer.

Format **MIL_ID MimAllocResult(SystemId, NbEntries, ResultType, ImResultIdPtr)**

MIL_ID SystemId;	System identifier
long NbEntries;	Number of result buffer entries
long ResultType;	Type of result buffer
MIL_ID *ImResultIdPtr;	Storage location for image processing result buffer identifier

Description This function allocates a result buffer with the specified number of entries, for use with the image processing module statistical functions.

When the result buffer is no longer required, you should release its memory, using **MimFree()**.

The **SystemId** parameter specifies the system on which the result buffer will be allocated. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. Specify M_DEFAULT_HOST to allocate on the default Host system of the current MIL application. Specify M_DEFAULT to have MIL select the most appropriate system on which to allocate the result buffer (it can be the default Host system or any already allocated system).

The **NbEntries** parameter specifies the number of buffer entries of the specified result buffer type, **ResultType**.

The **ResultType** parameter specifies the type of data that will be stored in this result buffer. This parameter can be set to one of the following:

M_HIST_LIST	For MimHistogram() results.
M_PROJ_LIST	For MimProject() results.
M_EXTREME_LIST	For MimFindExtreme() results.
M_EVENT_LIST	For MimLocateEvent() results.
M_COUNT_LIST	For MimCountDifference() results.

By default, the result buffer is of type long. A floating-point result buffer can be allocated for the M_PROJ_LIST, M_EXTREME_LIST and M_EVENT_LIST types, by appending M_FLOAT to the **ResultType** (for example, M_PROJ_LIST + M_FLOAT).

The **ImResultIdPtr** parameter specifies the address of the variable in which the image processing result buffer identifier is to be written. Since **MimAllocResult()** also returns the image processing result buffer identifier, you can set this parameter to `M_NULL`. If allocation fails, `M_NULL` is written as the identifier.

Note This function is optimized for packed binary buffers for the `M_COUNT_LIST` results only.

Return value The returned value is the image processing result buffer identifier. If allocation fails, `M_NULL` is returned.

Examples `mcount.c`, `mhists.c`

See also **MsysAlloc()**, **MimGetResult()**, **MimGetResult1d()**, **MimFree()**

MimArith

Synopsis Perform a point-to-point arithmetic operation.

Format `void MimArith(Src1ImageBufId, Src2ImageBufId,
DestImageBufId, Operation)`

double Src1ImageBufId;	Source 1 image buffer identifier or a constant
double Src2ImageBufId;	Source 2 image buffer identifier or a constant
MIL_ID DestImageBufId;	Destination image buffer identifier
long Operation;	Operation to perform

Description This function performs the specified point-to-point operation on two images, an image and a constant, an image, or a constant, storing results in the specified destination image buffer.

The **Src1ImageBufId** parameter specifies the data source of the first operand. This parameter can be given an image buffer identifier or a constant. When using a constant, it will be considered to have the same type as the destination buffer.

The **Src2ImageBufId** parameter specifies the data source of the second operand. This parameter can be given an image buffer identifier or a constant. If the selected operation uses only one operand, set this parameter to M_NULL. When using a constant, it will be considered to have the same type as the destination buffer.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier.

The **Operation** parameter specifies the operation to perform. This parameter should be set in accordance to the operands. Operations using two image buffer operands are the following:

M_ADD	M_SUB	M_AND
M_NAND	M_OR	M_XOR
M_NOR	M_XNOR	M_MIN
M_MAX	M_DIV	M_MULT
M_SUB_ABS		M_DIV+M_FIXED_POINT

The following are operations that use an image buffer operand and a constant. If the operation is not commutative, you must give the constant as either the first or second operand, as can be determined from the predefined operation name.

M_ADD_CONST	M_SUB_CONST	M_CONST_SUB
M_AND_CONST	M_NAND_CONST	M_OR_CONST
M_XOR_CONST	M_NOR_CONST	M_XNOR_CONST
M_MIN_CONST	M_MAX_CONST	M_MULT_CONST
M_DIV_CONST	M_CONST_DIV	M_DIV_CONST+M_FIXED_POINT
		M_CONST_DIV+M_FIXED_POINT

Operations using only one image buffer operand are the following:

M_NOT	M_NEG	M_ABS	M_PASS
-------	-------	-------	--------

The operation using only a constant is the following:

M_CONST_PASS (fills the destination with a constant)
--

Operations with the suffix ABS take the absolute value of the operation result. Operations that use the M_FIXED_POINT attribute provide the result in an 0.8 fixed-point format (where the eight most-significant bits are the integer portion and the eight least-significant bits are the fractional portion) for an 8-bit destination. That is, the 8 bits comprise only the fractional portion of the value, and no integer portion. When the destination is 16-bit, operations that use the M_FIXED_POINT attribute provide the result in a 8.8 fixed-point format. For a 32-bit destination, the result is in a 16.16 fixed-point format.

Logical operations (listed below) cannot be performed on a floating-point buffer.

M_OR	M_AND	M_XOR
M_NOR	M_NAND	M_XNOR
M_OR_CONST	M_AND_CONST	M_XOR_CONST
M_NOR_CONST	M_NAND_CONST	M_XNOR_CONST
M_NOT	M_CONST_DIV+M_FIXED_POINT	
M_DIV+M_FIXED_POINT	M_DIV_CONST+M_FIXED_POINT	

To force certain operations to saturate when the resulting value overflows or underflows add `M_SATURATION` to the operation name (for example, `M_ADD+M_SATURATION`). The following operations can be used this way:

<code>M_ADD</code>	<code>M_ADD_CONST</code>	<code>M_SUB</code>
<code>M_SUB_ABS</code>	<code>M_SUB_CONST</code>	<code>M_CONST_SUB</code>
<code>M_MULT</code>	<code>M_MULT_CONST</code>	

Note This function is optimized for packed binary buffers.

When performing a division operation using `M_DIV`, `M_DIV + M_FIXED_POINT`, or any other `M_DIV_xx` attribute, dividing a value by 0 will generate a destination pixel with a value that is impossible to predict. However, an error is not generated.

Status In-place processing is supported (source equals destination), but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

Examples `mcolor.c`, `mcount.c`, `mperim.c`, `msegment.c`, `mthread.c`

MimArithMultiple

Synopsis Perform a point-to-point arithmetic operation using multiple source images.

Format **void MimArithMultiple(Src1ImageBufId, Src2ImageBufId, Src3ImageBufId, Src4ImageBufId, Src5ImageBufId, DestImageBufId, Operation, OperationFlag)**

double Src1ImageBufId;	Source 1 image buffer identifier or a constant
double Src2ImageBufId;	Source 2 image buffer identifier or a constant
double Src3ImageBufId;	Source 3 image buffer identifier or a constant
double Src4ImageBufId;	Source 4 image buffer identifier or a constant
double Src5ImageBufId;	Source 5 image buffer identifier or a constant
MIL_ID DestImageBufId;	Destination image buffer identifier
long Operation;	Operation to perform
long OperationFlag;	Flag associated with the Operation argument

Description This function performs the specified point-to-point operation requiring multiple images, images and constants, or constants, storing results in the specified destination image buffer. Note also, that this function does not take 1-bit buffers in any of its parameters.

The **Src...ImageBufId** parameters specify the data sources of the operands. These parameters can be given an image buffer identifier or a constant. If the selected operation does not require all of the possible operands, set the excess source buffer parameters to M_NULL.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier.

The **Operation** parameter specifies the operation to perform. The following table lists the valid multiple operation names, their equations, and their descriptions, as well as their requirements for the sources of the operands:

Operation, Equation, and Description	Source 1	Source 2	Source 3	Source 4	Source 5
M_OFFSET_GAIN $((\text{Src1} - \text{Src2}) * \text{Src3}) / \text{Src4}$ Apply per-pixel gain and offset correction to an image.	image*	image*	image*	constant and power of 2	M_NULL
* These image buffers must be of the same type.					
M_WEIGHTED_AVERAGE $((\text{Src1} - \text{Src3}) / \text{Src2}) + \text{Src3}$ Apply a weighted averaging to an image and place the results in an accumulator.	image*	constant and power of 2	image*	M_NULL	M_NULL
* These image buffers must be of the same type.					
M_MULTIPLY_ACCUMULATE_1 $((\text{Src1} * \text{Src2}) + \text{Src3}) / \text{Src4}$ Apply the stated equation to the specified image.	image	constant*	constant*	constant* and power of 2	M_NULL
* These constants are cast in the same type as the source 1 image buffer.					
M_MULTIPLY_ACCUMULATE_2 $((\text{Src1} * \text{Src2}) + (\text{Src3} * \text{Src4})) / \text{Src5}$ Apply the stated equation to the specified images.	image*	constant*	image*	constant*	constant and power of 2
* The image buffers and the constants must be of the same type (signed or unsigned).					

To force the operations to saturate when the resulting value overflows or underflows add M_SATURATION to the operation name (for example, M_OFFSET_GAIN+M_SATURATION). In the case of M_OFFSET_GAIN (with unsigned buffers), an additional saturation to 0 is performed after the subtraction in order to avoid negative underflows.

The **OperationFlag** parameter must be set to M_DEFAULT.

Status In-place processing is supported (source equals destination), but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

MimBinarize

Synopsis Perform a point-to-point binary thresholding operation.

Format **long MimBinarize(SrcImageBufId, DestImageBufId, Condition, CondLow, CondHigh)**

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long Condition;	Conditional operator for selection
double CondLow;	Low compare value for the condition
double CondHigh;	High compare value for the condition

Description This function performs binary thresholding on the specified image. Each pixel that meets the specified condition is set to the highest unsigned destination buffer value, while other pixels are set to 0. For example, the highest buffer value for an 8-bit buffer is 0xff (regardless if the source buffer is signed).

MIL can automatically determine the threshold values (**CondLow** and/or **CondHigh**) from the source image's histogram. In this case, if the histogram contains only two peaks, the threshold value is set to the minimum value between these peaks. This option can only be used for 8-bit and 16-bit source images. 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). If required, this function can return the threshold value, without binarizing the source image.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. In general, this parameter must be given an image buffer identifier.

If you want **MimBinarize()** to automatically determine the threshold values, without binarizing the source image, set the **DestImageBufId** parameter to M_NULL and set the **CondLow** and **CondHigh** values to M_DEFAULT.

The **Condition** parameter specifies the thresholding condition. This parameter can be set to one of two types of conditions:

- Conditions that use two limits (**CondLow** and **CondHigh**):
M_OUT_RANGE, M_IN_RANGE
- Conditions that use one limit (**CondLow**):
M_EQUAL, M_NOT_EQUAL, M_GREATER, M_LESS,
M_GREATER_OR_EQUAL, M_LESS_OR_EQUAL.

When the M_OUT_RANGE condition is selected, pixels with values less than **CondLow**, or greater than **CondHigh**, are set to the highest buffer value, while other pixels are set to zero (0).

When the M_IN_RANGE condition is selected, pixels with values from **CondLow** to **CondHigh**, inclusive, are set to the highest buffer value, while other pixels are set to zero (0).

When you select any of the other conditions, if the condition applied to the pixel is true, that pixel is set to the highest buffer value, while other pixels are set to zero (0).

If a floating point destination buffer is specified, pixels that meet the condition are set to one (1).

The **CondLow** and **CondHigh** parameters specify the upper and lower limits of the selected condition. To have one of these parameters automatically determined from the image's histogram, set it to M_DEFAULT. If the condition uses only one limit, set the **CondLow** parameter to the required limit (or to M_DEFAULT) and set **CondHigh** to M_NULL. If the source buffer is binary, **CondLow** and **CondHigh** must be equal to 0 or 1.

Note that the values for **CondLow** and **CondHigh** are cast to the source buffer's data type and depth.

Note This function is optimized for packed binary buffers.

Return value The returned value is the value used for the **CondLow** parameter (either the automatically selected value or the manually chosen value).

Status In-place processing is supported (source equals destination), but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

Examples mblob.c, mmultdis.c, mopen.c, mperim.c

See also MimClip()

MimClip

Synopsis Perform a point-to-point clipping operation.

Format `void MimClip(SrcImageBufId, DestImageBufId, Condition, CondLow, CondHigh, WriteLow, WriteHigh)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long Condition;	Clipping condition
double CondLow;	Low clipping point
double CondHigh;	High clipping point
double WriteLow;	Value written if low clipping condition satisfied
double WriteHigh;	Value written if high clipping condition satisfied

Description This function clips each image pixel that meets the specified condition. If the condition has one clipping point, each pixel that satisfies this condition is replaced with the specified **WriteLow** value. If it has two clipping points, they are either replaced with the **WriteLow** or **WriteHigh** value depending on the condition. Pixels that do not satisfy the condition are not affected.

The **SrcImageBufId** parameter specifies the identifier of the image data source.

The **DestImageBufId** parameter specifies the identifier of the destination image buffer.

The **Condition** parameter specifies the clipping condition. This parameter can be set to one of two types of conditions.

- Conditions with two clipping points:

M_OUT_RANGE	Replace pixel values less than CondLow with WriteLow and those greater than CondHigh with WriteHigh .
M_IN_RANGE	Replace pixel values in the range of CondLow and CondHigh , inclusive, with WriteLow .

- Conditions with one clipping point:

M_EQUAL	Replace pixel values equal to CondLow with WriteLow .
M_NOT_EQUAL	Replace pixel values not equal to CondLow with WriteLow .

M_GREATER	Replace pixel values greater than CondLow with WriteLow .
M_LESS	Replace pixel values less than CondLow with WriteLow .
M_GREATER_OR_EQUAL	Replace pixel values greater than or equal to CondLow with WriteLow .
M_LESS_OR_EQUAL	Replace pixel values less than or equal to CondLow with WriteLow .

The **CondLow** and **CondHigh** parameters specify the upper and lower clipping points of the selected condition. If the condition uses only one limit, set the **CondLow** parameter to the required limit and set **CondHigh** to M_NULL. These parameters are cast to the source buffer's data type and interpreted accordingly. Note, if the source buffer is binary, **CondLow** and **CondHigh** must be equal to 0 or 1.

The **WriteLow** and **WriteHigh** parameters specify the values to write to the destination buffer when a pixel satisfies the specified clipping condition. The condition determines whether **WriteLow** or **WriteHigh** is written. When **WriteHigh** is not used, you should set it to M_NULL. These parameters are cast to the destination buffer's data type. Note, if the destination buffer is binary, **WriteLow** and **WriteHigh** must be equal to 0 or 1.

Note This function is optimized for packed binary buffers.

Status In-place processing is supported (source equals destination), but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

Examples mcolor.c, msegment.c

See also MimBinarize()

MimClose

Synopsis Perform a binary or grayscale closing-type morphological operation.

Format **void MimClose(SrcImageBufId, DestImageBufId,
 NbIteration, ProcMode)**

MIL_ID SrcImageBufId;	Source image buffer
MIL_ID DestImageBufId;	Destination image buffer identifier
long NbIteration;	Number of operation iterations
long ProcMode;	Processing mode

Description This function performs a binary or grayscale closing operation on the given source image for the specified number of iterations. A closing is a dilation followed by an erosion.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier.

The **NbIteration** parameter specifies the number of times to iterate the operation.

The **ProcMode** parameter specifies the processing mode to use. This parameter can be set to the following:

M_BINARY	Non-zero pixels will be treated as ones (1) during processing and the resulting non-zero pixels will have the maximum value of the unsigned buffer (for example, 0xff for an 8-bit buffer).
M_GRAYSCALE	The source image's gray values are used for processing.

In binary mode, this function uses a 3 x 3 full rectangular structuring element; in grayscale mode, a 3 x 3 empty one.

Note This function is optimized for packed binary buffers.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

Example mblob.c

See also **MimOpen(), MimDilate(), MimErode()**

MimConnectMap

Synopsis Perform a 3 by 3 binary connectivity mapping.

Format `void MimConnectMap(SrcImageBufId, DestImageBufId, LutBufId)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
MIL_ID LutBufId;	LUT buffer identifier

Description This function performs a 3 x 3 binary connectivity mapping. It calculates a connectivity code for each pixel in the source image, then maps the codes through the specified LUT buffer.

The **SrcImageBufId** parameter specifies the identifier of the source of the operation. This parameter must be given an image buffer identifier. The source pixels are treated as binary (that is, all non-zero pixels are treated as 1). Pixel connectivity codes are determined in the following order:

$$\begin{bmatrix} n_3 & n_2 & n_1 \\ n_4 & n_8 & n_0 \\ n_5 & n_6 & n_7 \end{bmatrix}$$

where n_i is either 0 or 1

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

Result = LUTMAP (connectivity code)

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier.

The **LutBufId** parameter specifies the identifier of the LUT buffer. As each connectivity code has 9 bits, you should supply a LUT buffer with at least 512 (2^9) entries; otherwise unpredictable results can occur.

Note This function is optimized for packed binary buffers.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

MimConvert

Synopsis Perform a color conversion.

Format **MIL_ID MimConvert(SrcImageId, DestImageId, ConversionType)**

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long ConversionType;	Type of conversion to perform

Description This function performs a color conversion on the source image and places the result in the destination buffer.

The **SrcImageBufId** parameter specifies the identifier of the source image buffer. Source buffer values must be positive. If specifying a floating-point buffer, the values of the buffer must be normalized between 0 and 1 before calling this function.

The **DestImageBufId** parameter specifies the identifier of the destination image buffer. If specifying a floating-point buffer, the values generated will be normalized between 0 and 1.

The **ConversionType** parameter specifies the type of conversion to perform. This parameter can be set to one of the following values:

ConversionType	Description
M_RGB_TO_HLS	An RGB to HLS conversion. The hue, luminance (intensity) and saturation are calculated from the source buffer. Both the source and destination image buffers must be 3-band image buffers allocated with MbufAllocColor() .
M_HLS_TO_RGB	A HLS to RGB conversion. The Red, Green and Blue components are calculated from the source buffer. Both the source and destination image buffers must be 3-band image buffers allocated with MbufAllocColor() .
M_RGB_TO_L	An RGB to L (luminance) conversion. The luminance (intensity) is calculated from the source buffer. The source image must be a 3-band image buffer allocated with MbufAllocColor() and the destination buffer must be a 1-band image buffer allocated with MbufAlloc2d() .

ConversionType	Description
M_L_TO_RGB	An L (luminance) to RGB conversion. The Red, Green and Blue components are calculated from the source buffer. The luminance (intensity) is repeated in each color band of the destination buffer, creating a monochromatic (gray) RGB buffer. The source buffer must be a 1-band image buffer allocated with MbufAlloc2d() and the destination buffer must be a 3-band image buffer allocated with MbufAllocColor() .
M_RGB_TO_Y	An RGB to Y conversion, where Y represents the luminance of the YUV color space. The source buffer must have 3 bands. The destination buffer can have 1 or 3 bands. If it has 3 bands, only the first band is overwritten.

The values generated will be normalized to the range of values in the destination buffer. For example, the hue values from 0 to 360 degrees will generate the values from 0 to 255 in an 8-bit destination buffer. Negative values in signed buffers will return invalid results. Note that for a floating point destination buffer, the values generated will be normalized between 0 and 1.

See also **MbufAllocColor()**, **MbufAlloc2d()**

MimConvolve

Synopsis Perform a general convolution operation.

Format `void MimConvolve(SrcImageBufId, DestImageBufId, KernelBufId)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
MIL_ID KernelBufId;	Kernel buffer identifier

Description This function performs a general convolution operation on the source buffer using the specified kernel, storing results in the specified destination buffer.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier.

The **KernelBufId** parameter specifies the identifier of the kernel buffer. This parameter can be given a custom or predefined kernel identifier. If you use a custom kernel, you must have previously allocated it with **MbufAlloc1d()** or **MbufAlloc2d()** and loaded it with values, using **MbufPut()**.

Operation flags associated with custom kernels can be modified in order to control the behavior of the convolution operation. By using **MbufControlNeighborhood()** you can control how the operation handles the borders of an image (overscan), whether or not the absolute value of the result is taken, which division factor to apply to the result, whether or not to saturate the result, and the position of the kernel's center.

The following is a list of predefined kernel buffer identifiers:

Default identifiers	Corresponding kernels and their associated operation parameters
M_SMOOTH	$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} / 16$
M_SHARPEN	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
M_SHARPEN2	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
M_HORIZ_EDGE	$\left\ \begin{bmatrix} 2 & 2 & 2 \\ 0 & 0 & 0 \\ -2 & -2 & -2 \end{bmatrix} \right\ $
M_VERT_EDGE	$\left\ \begin{bmatrix} -2 & 0 & 2 \\ -2 & 0 & 2 \\ -2 & 0 & 2 \end{bmatrix} \right\ $
M_EDGE_DETECT	$\left(\left\ \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \right\ + \left\ \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \right\ \right) / 2$
M_EDGE_DETECT2	$\left(\left\ \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \right\ + \left\ \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \right\ \right) / 2$
M_LAPLACIAN_EDGE	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
M_LAPLACIAN_EDGE2	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$

Operation flags associated to predefined kernels have the following default states: transparent overscan, results are saturated (saturation enabled), the kernel's center pixel is the top left pixel of the central element in a neighborhood.

You can temporarily disable overscanning by adding `M_OVERSCAN_DISABLE` to a predefined kernel. For example, `M_SMOOTH+M_OVERSCAN_DISABLE`. This accelerates the convolution on certain platforms if the overscan data is not important in the resulting buffer.

If the source and destination are multi-band buffers then the same kernel is applied to every band of the source buffer.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

Examples `mconvol.c`, `mcount.c`, `mopen.c`, `mthread.c`

See also `MbufControlNeighborhood()`, `MbufAlloc1d()`, `MbufAlloc2d()`, `MbufPut()`

MimCountDifference

Synopsis Count the number of pixels that differ in each image.

Format **void MimCountDifference(Src1ImageBufId, Src2ImageBufId, ImResultId)**

MIL_ID Src1ImageBufId;	Source 1 image buffer identifier
MIL_ID Src2ImageBufId;	Source 2 image buffer identifier
MIL_ID ImResultId;	Image processing result buffer identifier

Description This function finds the number of differences between the two specified source buffers and stores the resulting number in the specified result buffer.

You can read the number of differences from the result buffer, using **MimGetResult1d()** or **MimGetResult()**, specifying M_VALUE as the result type.

The **Src1ImageBufId** parameter specifies the identifier of the first image data source. This parameter can be given an image buffer identifier. The image buffer must be one-band.

The **Src2ImageBufId** parameter specifies the identifier of the second image data source. This parameter can only be given an image buffer identifier. The image buffer must be one-band.

The **ImResultId** parameter specifies the identifier of the buffer in which to store the differences. This parameter must be given the identifier of an image processing result buffer that was allocated with **MimAllocResult()** and has an M_COUNT_LIST type. The buffer needs only one entry.

Note This function is optimized for packed binary buffers.

See also **MimAllocResult(), MimGetResult(), MimGetResult1d()**

MimDilate

Synopsis Perform a binary or grayscale dilation-type morphological operation.

Format `void MimDilate(SrcImageBufId, DestImageBufId, NbIteration, ProcMode)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long NbIteration;	Number of operation iterations
long ProcMode;	Processing mode

Description This function performs a binary or grayscale dilation on the given source image for the specified number of iterations.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier.

The **NbIteration** parameter specifies the number of times to iterate the operation.

The **ProcMode** parameter specifies the processing mode to use. This parameter can be set to the following:

M_BINARY	Non-zero pixels will be treated as ones (1) during processing and the resulting non-zero pixels will have the maximum value of the unsigned buffer (for example, 0xff for an 8-bit buffer).
M_GRAYSCALE	The source image's gray values are used for processing and the resulting buffer will also contain gray values.

In binary mode, this function uses a 3 x 3 full rectangular structuring element; in grayscale mode, a 3 x 3 empty one.

Note This function is optimized for packed binary buffers.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

Example mperim.c

See also `MimErode()`, `MimOpen()`, `MimClose()`

MimDistance

Synopsis Perform a distance transformation.

Format **void MimDistance(SrcImageBufId, DestImageBufId, DistanceTransform)**

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long DistanceTransform;	Distance transformation to perform

Description This function determines the shortest distance between each blob pixel and the blob's background, and assigns this distance to the pixel. It produces a type of contour mapping of a blob.

The **SrcImageBufId** parameter specifies the identifier of the source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the resulting image. This parameter must be given an image buffer identifier.

The **DistanceTransform** parameter specifies the way in which the minimum distance from blob pixel to background pixel is calculated. This parameter approximates the true distance from blob pixel to background pixel using a 3 x 3 distance matrix and can be set to one of the following:

Transform	3x3 Distance Matrix	Description
M_CHAMFER_3_4	$\begin{bmatrix} 4 & 3 & 4 \\ 3 & 0 & 3 \\ 4 & 3 & 4 \end{bmatrix}$	Determines the minimum distance using horizontal, vertical or diagonal pixel steps. Horizontal and vertical steps are counted as 3; diagonal steps are counted as 4. This transform provides the best approximation to Euclidean distance. However, when using this transform, the destination buffer must meet certain requirements. The requirements are detailed below this table.
M_CHESSBOARD	$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	Determines the minimum distance using horizontal, vertical, or diagonal pixel steps. All steps count as 1.

Transform	3x3 Distance Matrix	Description
M_CITY_BLOCK	$\begin{bmatrix} \infty & 1 & \infty \\ 1 & 0 & 1 \\ \infty & 1 & \infty \end{bmatrix}$	Determines the minimum distance using only horizontal or vertical pixel steps. Horizontal and vertical steps count as 1.

The M_CHAMFER_3_4 transform requires that the destination buffer be deep enough to hold a number at least three times the maximum distance from a blob pixel to its edge. 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.

Note This function is optimized for packed binary buffers.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

Example msegment.c

MimEdgeDetect

Synopsis Perform a specific edge detection operation and produce a gradient intensity and/or gradient angle image.

Format **void MimEdgeDetect(SrcImageBufId, DestIntensityImageBufId, DestAngleImageBufId, KernelId, ControlFlag, Threshold)**

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestIntensityImageBufId;	Destination gradient intensity image buffer identifier
MIL_ID DestAngleImageBufId;	Destination gradient angle image buffer identifier
MIL_ID KernelId;	Kernel identifier
long ControlFlag;	Flag to control operation
long Threshold;	Threshold value for gradient intensity

Description This function performs an edge detection operation on the specified source image, using the specified kernel. It produces a gradient intensity image and/or a gradient angle image in the specified image buffer(s). If one of the destination images is not required, specify M_NULL as its image buffer identifier.

The **SrcImageBufId** parameter specifies the identifier of the source of the operation. This parameter must be given an image buffer identifier.

The **DestIntensityImageBufId** parameter specifies the identifier of the destination of the resulting gradient intensity image. This parameter must be given an image buffer identifier. Saturation is performed on this buffer.

The **DestAngleImageBufId** parameter specifies the identifier of the destination of the resulting gradient angle image. This parameter must be given an image buffer identifier. The angle is returned from 0° to 360° (counter-clockwise) and mapped in the entire range of the destination buffer. For signed data types, mapping is done in both the positive and negative range of the buffer and represents angle values from -180° to 180°.

The maximum positive value of the angle buffer (255 for an unsigned 8-bit buffer) maps to 360° and is reserved for undefined results produced when a gradient threshold is used.

The **KernelId** parameter specifies the identifier of the edge detection kernel. This parameter must be set to the predefined 3x3 kernel: M_SOBEL. To disable overscan processing, specify M_SOBEL+M_OVERSCAN_DISABLE). This kernel is as follows:

X gradient:	$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$
Y gradient:	$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$

The **ControlFlag** parameter specifies the flag used to control the operation calculations. These flags include various combinations of fast and full gradient and angle computations.

Full gradient computation	Gradient = sqrt(GradientX*GradientX + GradientY*GradientY)
Fast gradient computation	Gradient = (abs(GradientX) + abs(GradientY))/2
Full angle computation	Angle = arctan(GradientY/GradientX)

Note that fast angle approximation introduces a maximum error of +/- 0.4°.

The **ControlFlag** parameter can be set to one of the following:

ControlFlag	Description
M_FAST_EDGE_DETECT	Fast computation of the gradient and angle. The gradient is computed as an approximation, using the average of the absolute value of the two directional components (X and Y). Fast angle approximation is used. This is the default flag (M_DEFAULT).

ControlFlag	Description
M_REGULAR_EDGE_DETECT	Full computation of the gradient and angle. The gradient is computed as the square root of the sum of the square of each directional component (X and Y). This method of calculation is slower but more precise.
M_FAST_ANGLE+M_REGULAR_GRADIENT	Full computation of the gradient. Fast angle approximation.
M_REGULAR_ANGLE+M_FAST_GRADIENT	Fast computation of the gradient. Full computation of the angle.

The **Threshold** parameter specifies the threshold value for calculating gradient intensity. For a gradient value lower than the threshold value, the angle is not computed (not considered a significative edge) and the resulting angle pixel is set to the reserved value (the maximum positive value of the buffer). To perform the full operation, set this parameter to zero or M_NULL.

Note This function can be performed on a floating point buffer. However, the results are not mapped and are returned in the range of 0° to 360° The degree of precision is equal to the precision of the floating point for regular computation and to 0.4° for fast computation. The value for undefined results is the maximum positive value of the floating point buffer.

Example msegment.c

MimErode

Synopsis Perform an erosion-type morphological operation.

Format `void MimErode(SrcImageBufId, DestImageBufId, NbIteration, Procmode)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long NbIteration;	Number of operation iterations
long ProcMode;	Processing mode

Description This function performs a binary or grayscale erosion on the given source image for the specified number of iterations.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier.

The **NbIteration** parameter specifies the number of times to iterate the operation.

The **ProcMode** parameter specifies the processing mode to use. This parameter can be set to the following:

M_BINARY	Non-zero pixels will be treated as ones (1) during processing and the resulting non-zero pixels will have the maximum value of the unsigned buffer (for example, 0xff for an 8-bit buffer).
M_GRAYSCALE	The source image's gray values are used for processing and the resulting buffer will also contain gray values.

In binary mode, this function uses a 3 x 3 full rectangular structuring element; in grayscale mode, a 3 x 3 empty one.

Note This function is optimized for packed binary buffers.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

See also `MimDilate()`, `MimOpen()`, `MimClose()`

MimFindExtreme

Synopsis Find an image buffer’s extremes (minimum and/or maximum pixel values).

Format **void MimFindExtreme(SrcImageBufId,
 ExtremeImResultId, ExtremeType)**

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID ExtremeImResultId;	Extreme value image processing result buffer identifier
long ExtremeType;	Type of result to calculate

Description This function finds the maximum and/or minimum value of the specified source image and stores results in the specified extreme result buffer.

You can read the minimum/or maximum from the result buffer, using **MimGetResult1d()** or **MimGetResult()**, specifying M_VALUE as the result type.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This parameter must be given an image buffer identifier. The buffer must be one-band.

The **ExtremeImResultId** parameter specifies the identifier of the buffer in which to store the extreme values. This parameter must be given the identifier of an image processing result buffer that was allocated with **MimAllocResult()** and that has an M_EXTREME_LIST type. If just the maximum or minimum is calculated, only one entry is needed. If both the minimum and maximum are calculated, the result buffer must have two entries. The minimum value is stored in the first entry and the maximum value is stored in the second.

The **ExtremeType** parameter specifies whether to find the minimum value, maximum value, or both. This parameter can be set to the following:

ExtremeType	Description
M_MIN_ABS_VALUE	Find the minimum value, as an absolute value.
M_MAX_ABS_VALUE	Find the maximum value, as an absolute value.
M_MIN_ABS_VALUE+ M_MAX_ABS_VALUE	Find the minimum and maximum values, as absolute values.
M_MIN_VALUE	Find the minimum value.
M_MAX_VALUE	Find the maximum value.

ExtremeType	Description
M_MIN_VALUE + M_MAX_VALUE	Find the minimum and maximum values.

Example mcount.c

See also **MimAllocResult()**, **MimGetResult1d()**, **MimGetResult()**

MimFlip

Synopsis Perform a horizontal or vertical image-flipping operation.

Format **void MimFlip(SrcImageBufId, DestImageId, Operation, OpFlag)**

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long Operation;	Operation to perform
long OpFlag;	Flag for the operation

Description This function flips the source image to the destination image according to the specified operation.

The **SrcImageBufId** parameter specifies the identifier of the source image buffer.

The **DestImageBufId** parameter specifies the identifier of the destination image buffer.

The **Operation** parameter specifies the operation to perform. This parameter can be set to one of the following:

M_FLIP_HORIZONTAL	Flip the image in a horizontal direction (left to right, along a vertical axis).
M_FLIP_VERTICAL	Flip the image in a vertical direction (top to bottom, along a horizontal axis).

The **OpFlag** parameter must be set to M_DEFAULT.

See also **MimRotate()**

MimFree

Synopsis Free an image processing result buffer.

Format **void MimFree(ImResultId)**

MIL_ID ImResultId;	Image processing result buffer identifier
--------------------	---

Description This function deallocates a result buffer previously allocated with **MimAllocResult()**.

The **ImResultId** parameter specifies the identifier of the image processing result buffer to be released.

See also **MimAllocResult()**

MimGetResult

Synopsis Get values from an image processing result buffer.

Format **void MimGetResult(ImResultId, ResultType, UserArrayPtr)**

MIL_ID ImResultId;	Image processing result buffer identifier
long ResultType;	Type of result to read
void *UserArrayPtr;	Array in which to return results

Description This function copies all the results from the specified result buffer to the specified one-dimensional destination user array.

The **ImResultId** parameter specifies the identifier of the image processing result buffer from which to get results.

The **ResultType** parameter specifies the type of data to read from the MIL result buffer. This parameter can be set to one of the following:

ResultType	Description
M_VALUE	Read values from M_HIST_LIST, M_PROJ_LIST, M_EXTREME_LIST, M_EVENT_LIST, or M_COUNT_LIST type buffer.
M_POSITION_X	Read image X-coordinate of values in M_EVENT_LIST type buffer. When the blob identifier image is calibrated, this coordinate is in calibration units; otherwise it in pixel units.
M_POSITION_Y	Read image Y-coordinate of values in M_EVENT_LIST type buffer. When the blob identifier image is calibrated, this coordinate is in calibration units; otherwise it in pixel units.
M_NB_EVENT	Read the number of occurrences in M_EVENT_LIST type buffer. Only one value is returned.

If an M_EXTREME_LIST type result buffer contains both the minimum and maximum value of an image, the minimum value followed by the maximum value are written. If it contains only one of these values, this value is stored in the first location.

The **UserArrayPtr** parameter specifies the address of the one-dimensional array in which to write results read from the MIL result buffer. This array must have at least the same number of elements as the total number of

entries to get (**NbEntries**), specified at **MimAllocResult()** time. In addition, the array must be of the same type as the result buffer (long or float).

Examples mcount.c, mhist.c

See also **MimAllocResult()**, **MimGetResult1d()**

MimGetResult1d

Synopsis Get values from a 1D region of an image processing result buffer.

Format **void MimGetResult1d(ImResultId, OffEntry, NbEntries, ResultType, UserArrayPtr)**

MIL_ID ImResultId;	Image processing result buffer identifier
long OffEntry;	Result entry offset
long NbEntries;	Number of result entries to get
long ResultType;	Type of result to read
void *UserArrayPtr;	Array in which to return results

Description This function copies the specified number of result entries from the specified result buffer to the specified one-dimensional destination user array.

The **ImResultId** parameter specifies the identifier of the image processing result buffer from which to get results.

The **OffEntry** parameter specifies the offset of the first result to read. The given value must be within the number of allocated result entries.

The **NbEntries** parameter specifies the number of result entries to get starting from the entry specified by **OffEntry**.

The **ResultType** parameter specifies the type of data that the MIL result buffer contains. This parameter can be set to one of the following:

ResultType	Description
M_VALUE	Read values from M_HIST_LIST, M_PROJ_LIST, M_EXTREME_LIST, M_EVENT_LIST, or M_COUNT_LIST type buffer.
M_POSITION_X	Read image X-coordinate of values in M_EVENT_LIST type buffer.
M_POSITION_Y	Read image Y-coordinate of values in M_EVENT_LIST type buffer.
M_NB_EVENT	Read the number of occurrences in M_EVENT_LIST type buffer. Only one value is returned.

If an M_EXTREME_LIST type result buffer contains both the minimum and maximum value of an image, the minimum is written first followed by the second.

The **UserArrayPtr** parameter specifies the address of the one-dimensional array in which to write results from the MIL result buffer. This array must have at least the same number of elements as the total number of entries to get (**NbEntries**) specified at **MimAllocResult()** time. In addition, the array must be of the same type as the result buffer (long or float).

See also **MimAllocResult()**, **MimGetResult()**

MimHistogram

Synopsis Generate the intensity histogram of an image buffer.

Format **void MimHistogram(SrcImageBufId, HistImResultId)**

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID HistImResultId;	Histogram image processing result buffer identifier

Description This function calculates the histogram (or pixel intensity distribution) of the specified source buffer and stores results in the specified histogram result buffer.

You can read the histogram values from the result buffer, using **MimGetResult1d()** or **MimGetResult()**, specifying M_VALUE as the result type.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This buffer must be one-band. The buffer data will be treated as **unsigned**.

The **HistImResultId** parameter specifies the identifier of the destination for the histogram results. This parameter must be given the identifier of an image processing result buffer that was allocated with **MimAllocResult()** and has an M_HIST_LIST type. If the result buffer has fewer entries than the full range of source values, the pixels that are out of range will not be written into the histogram.

Note Floating point values will be cast as unsigned long values before performing the histogram. Therefore, unexpected results can occur if a floating point value is larger than the unsigned long range.

Example mhist.c

See also **MimHistogramEqualize(), MimAllocResult()**

MimHistogramEqualize

Synopsis Perform a histogram equalization of an image.

Format `void MimHistogramEqualize(SrcImageBufId, DestImageBufId, Method, Alpha, Min, Max)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image or LUT buffer identifier
long Method;	Type of equalization
double Alpha;	Adjustment parameter
double Min;	Lowest pixel value to equalize
double Max;	Highest pixel value to equalize

This function performs a histogram equalization of the specified source image. Results are written to a destination buffer, which can be either an image buffer or a LUT buffer.

This function first performs a histogram of the source image buffer. The histogram is then used to calculate a transformation LUT that can be used to enhance the source image (with **MimLutMap()**). If the destination buffer is a LUT, the transformation LUT is copied into the destination LUT. If the destination buffer is an image, the source buffer is remapped through the transformation LUT to produce the destination image.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This parameter will be treated as an **unsigned** image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer or LUT buffer identifier.

The **Method** parameter specifies the type of equalization to perform. The following methods and their associated density functions are available:

Required distribution	Output probability density model	Transfer functions
M_UNIFORM Uniform	$P_g(g) = \frac{1}{g_{max} - g_{min}}$ $g_{min} \leq g \leq g_{max}$	$g = [g_{max} - g_{min}]P_f(f) + g_{min}$
M_EXPONENTIAL Exponential	$P_g(g) = \alpha \left(e^{[(-\alpha)(g - g_{min})]} \right)$ $g \geq g_{min}$	$g = g_{min} - \frac{1}{\alpha} \ln[1 - P_f(f)]$
M_RAYLEIGH Rayleigh	$P_g(g) = \frac{g - g_{min}}{\alpha^2} e^{\left[\frac{(g - g_{min})^2}{2\alpha^2} \right]}$ $g \geq g_{min}$	$g = g_{min} + \frac{1}{2} \left[2\alpha^2 \ln \left\{ \frac{1}{1 - P_f(f)} \right\} \right]$
M_HYPER_CUBE_ROOT Hyperbolic cube root	$P_g(g) = \frac{1}{3} \left(\frac{g^{-2/3}}{\sqrt[3]{g_{max}} - \sqrt[3]{g_{min}}} \right)$	$g = \left(\left[\sqrt[3]{g_{max}} - \sqrt[3]{g_{min}} \right] [P_f(f)] + \sqrt[3]{g_{min}} \right)^3$
M_HYPER_LOG Hyperbolic logarithmic	$P_g(g) = \frac{1}{g[\ln(g_{max}) - \ln(g_{min})]}$	$g = g_{min} \left[\frac{g_{max}}{g_{min}} \right] P_f(f)$
The cumulative probability distribution, $P_f(f)$, of the input image is approximated by its cumulative histogram: $P_f(f) \approx \sum_{m=0}^i H_F(m) -$ Refer to Digital Image Processing, William K. Pratt, United States, John Wiley & Sons, 1978, p.318.		

The **Alpha** parameter is used with M_EXPONENTIAL and M_RAYLEIGH methods. In the case of the M_EXPONENTIAL method, a greater Alpha yields a lower occurrence of the most frequent pixels of the histogram in the resulting image buffer.

In the case of the M_RAYLEIGH method, the greater the alpha is, the greater the occurrence of the most frequent pixels of the histogram in the resulting image buffer.

The **Min** and **Max** parameters specify the range of pixels that will be remapped.

Note Floating point values will be cast as unsigned long values before performing the histogram. Therefore, unexpected results can occur if a floating point value is larger than the unsigned long range.

See also **MimLutMap()**, **MimHistogram()**

MimInquire

Synopsis Inquire about an image processing result buffer parameter setting.

Format **long MimInquire(BufId, InquireType, UserVarPtr)**

long BufId;	Image processing result buffer identifier
long InquireType;	Type of information about which to inquire
void *UserVarPtr;	Storage location for requested information

Description This function inquires about a specified MIL image processing result buffer setting. The result buffer must have been allocated with **MimAllocResult()**.

The **BufId** parameter specifies the identifier of the image processing result buffer.

The **InquireType** parameter specifies the type of information about which to inquire. This parameter can be set to one of the following values:

Inquire Type	Description
M_RESULT_SIZE	Number of entries in the buffer.
M_RESULT_TYPE	Attribute or nature of the buffer.
M_OWNER_SYSTEM	The identifier of the system on which the buffer is allocated.
M_MODIFICATION_COUNT	Number of modifications made to the buffer since it was allocated.
M_HOST_ADDRESS	RAM address of the Host buffer (huge).

The **UserVarPtr** parameter specifies the address of the variable in which the requested information is to be written. The variable must be of type long (for M_OWNER_SYSTEM it must be a MIL_ID pointer). Since the **MimInquire()** function also returns the requested information, you can set this parameter to M_NULL.

Return value The returned value is the requested MIL buffer information, cast to long.

MimLabel

Synopsis Label objects in an image buffer.

Format `void MimLabel(SrcImageBufId, DestImageBufId, ProcMode)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long ProcMode;	Processing mode

Description This function labels all objects (or blobs) in the specified source image, starting from the top-left corner, with unique consecutive values beginning with the value 1. Each pixel in the same blob is given the same numerical value.

If you want to distinguish between touching blobs, you should separate the blobs. For example, you can use an erosion operation before performing the labeling operation.

The **SrcImageBufId** parameter specifies identifier of the data source of the operation. This parameter must be given an image buffer identifier. If the source buffer is not binary, all non-zero pixels are considered as part of an object or blob.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier. The destination buffer should be large enough to hold the maximum number of objects (blobs). For example, an 8-bit buffer can be used for a maximum of 254 blobs and a 16-bit buffer can be used for a maximum of 65534 blobs). If the destination buffer depth is too small, several blobs might be given the same value.

The **ProcMode** parameter specifies the type of connectivity (lattice) to use for processing. This parameter can be set to:

M_8_CONNECTED	Each pixel has 8 neighbors. If two blobs touch on the vertical, horizontal, or diagonal, they are considered one blob.
M_4_CONNECTED	Each pixel has 4 neighbors. If two blobs touch on the vertical and horizontal axis, they are considered one blob.

If the source image is a binarized image (containing 0 or the maximum value of the buffer, achieved for example with **MimBinarize()**), you can add **M_BINARY** to the connectivity mode to accelerate processing, particularly if the labeling is done by the Host. For example, **M_4_CONNECTED+M_BINARY**. The default processing mode is **M_8_CONNECTED+M_GRAYSCALE**.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

Example `mcount.c`

See also **MimBinarize()**

MimLocateEvent

Synopsis Find pixel coordinates or values that satisfies a specified condition.

Format **void MimLocateEvent(SrcImageBufId, EventImResultId, Condition, CondLow, CondHigh)**

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID EventImResultId;	Event image processing result buffer identifier
long Condition;	Event condition
double CondLow;	Low compare value
double CondHigh;	High compare value

Description This function finds the coordinates and value of the pixels that satisfy the specified condition in the specified source image. Results are stored in the specified event result buffer.

Using **MimGetResult()** or **MimGetResult1d()**, specify the result type as **M_VALUE** to read the values of those pixels that satisfy the specified condition; as **M_POSITION_X** or **M_POSITION_Y** to obtain the position of those pixels; as **M_NB_EVENT** to obtain the total number of pixels that satisfy the specified condition.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This parameter must be given an image buffer identifier. The buffer must be one-band.

The **EventImResultId** parameter specifies the identifier of the buffer in which to store the event values. This parameter must be given the identifier of an image processing result buffer allocated with **MimAllocResult()** and an **M_EVENT_LIST** type. The buffer must have enough entries to hold the expected number of events.

If the number of pixels found is less than the number of event result buffer entries, the remaining result buffer entries are set to **M_INVALID**. If the number of events is bigger than the number of allocated result buffer entries, only the first **NbEntries** events are kept.

The **Condition** parameter specifies under what condition pixel values are considered an event. This parameter can be set to one of two types of conditions:

- Conditions that use two limits (**CondLow** and **CondHigh**):
M_OUT_RANGE, M_IN_RANGE.
- Conditions that use one limit (**CondLow**):
M_EQUAL, M_NOT_EQUAL, M_GREATER, M_LESS,
M_GREATER_OR_EQUAL, M_LESS_OR_EQUAL.

When the M_OUT_RANGE condition is selected, pixels with values less than **CondLow**, or greater than **CondHigh**, are considered an event. When the M_IN_RANGE condition is selected, pixels with values from **CondLow** to **CondHigh**, inclusive, are considered an event.

When you select any of the other conditions, if the condition applied using the **CondLow** is true, the pixel is considered an event.

The **CondLow** and **CondHigh** parameters specify the lower and upper limits of the specified condition, **Condition**. When the upper limit is not required, set **CondHigh** to M_NULL.

See also **MimAllocResult()**, **MimGetResult()**, **MimGetResult1d()**

MimLutMap

Synopsis Perform a point-to-point LUT mapping operation.

Format `void MimLutMap(SrcImageBufId, DestImageBufId, LutBufId)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
MIL_ID LutBufId;	LUT identifier

Description This function maps each pixel in the specified source image to values determined by the specified look-up table (LUT).

If the source image is signed, negative numbers are treated as unsigned values and address the LUT accordingly. For example, -1 is represented as 0xff for 8-bit image data and 0xffff for 16-bit image data and will therefore address LUT entry 0xff or 0xffff, respectively.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier.

The **LutBufId** parameter specifies the LUT through which to map source input values. This parameter must be given a valid LUT buffer identifier.

If the LUT has fewer entries than the full range of source values, unexpected results will occur. If the LUT buffer depth is greater than that of the destination buffer, results will be truncated to fit the destination buffer. LUT entries must have been previously initialized (for example, using **MgenLutRamp()** or **MbufPut1d()**).

Note Floating point values will be cast as unsigned long values before performing the LUT map operation. Therefore, unexpected results can occur if a floating point value is larger than the unsigned long range.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

Example `mnatfct.c`

See also `MgenLutRamp()`, `MbufPut1d()`

MimMorphic

Synopsis Perform a morphological transformation using a user-defined kernel.

Format **void MimMorphic(SrcImageBufId, DestImageBufId, StructElemBufId, Operation, NbIteration, ProcMode)**

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
MIL_ID StructElemBufId;	Structuring element buffer identifier
long Operation;	Morphological operation to perform
long NbIteration;	Number of operation iterations
long ProcMode;	Processing mode

Description This function performs one of several morphological transformations on the specified source image, using a user-defined structuring element.

The **SrcImageBufId** parameter specifies the identifier of the source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the resulting image. This parameter must be given an image buffer identifier.

The **StructElemBufId** parameter specifies the identifier of the structuring element buffer. The user-defined structuring element must have been allocated, using **MbufAlloc1d()** or **MbufAlloc2d()** with an **M_STRUCT_ELEMENT Attribute**, and loaded with structuring-element values, using **MbufPut()**. These values can be set to **M_DONT_CARE** to specify that the corresponding image pixels should not be taken into account during the operation. You can set the overscan type and the center of the structuring element, using **MbufControlNeighborhood()**.

If the source and destination are multi-band buffers then the same structuring element is applied to every band of the source buffer.

The **Operation** parameter specifies the operation to perform. Supported operations are:

M_ERODE	Erosion
M_DILATE	Dilation
M_THIN	Thinning
M_THICK	Thickening
M_HIT_OR_MISS	Hit or miss transformation
M_MATCH	Matching

The **NbIteration** parameter specifies the number of times to iterate the operation. For M_HIT_OR_MISS or M_MATCH operations, this parameter must be set to 1.

The **ProcMode** parameter specifies the processing mode to use. This parameter can be set to the following:

M_BINARY	Non-zero pixels will be treated as ones (1) during processing.
M_GRAYSCALE (default value)	The source image's gray values are used for processing and the resulting buffer will also contain gray values.

Note This function is optimized for packed binary buffers.

Status In-place processing is supported, but the source and destination image buffers cannot overlap (a situation that can only occur when using child buffers).

Example `mopen.c`

See also `MbufAlloc1d()`, `MbufAlloc2d()`, `MimDilate()`, `MimErode()`, `MimOpen()`, `MimThin()`, `MimClose()`

MimOpen

Synopsis Perform a binary or grayscale opening-type morphological operation.

Format **void MimOpen(SrcImageBufId, DestImageBufId, NbIteration, Procmode)**

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long NbIteration;	Number of operation iterations
long ProcMode;	Processing mode

Description This function performs a binary or grayscale opening operation on the given source image for the specified number of iterations. An opening is an erosion followed by a dilation.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier.

The **NbIteration** parameter specifies the number of times to iterate the operation.

The **ProcMode** parameter specifies the processing mode to use. This parameter can be set to the following:

M_BINARY	Non-zero pixels will be treated as ones (1) during processing and the resulting non-zero pixels will have the maximum value of the unsigned buffer (for example, 0xff for an 8-bit buffer).
M_GRAYSCALE	The source image's gray values are used for processing and the resulting buffer also contains gray values.

In binary mode, this function uses a 3 x 3 full structuring element; in grayscale mode, a 3 x 3 empty one.

Note This function is optimized for packed binary buffers.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

Examples mblob.c, mcount.c, mopen.c, mperim.c

See also **MimErode()**, **MimDilate()**, **MimClose()**

MimPolarTransform

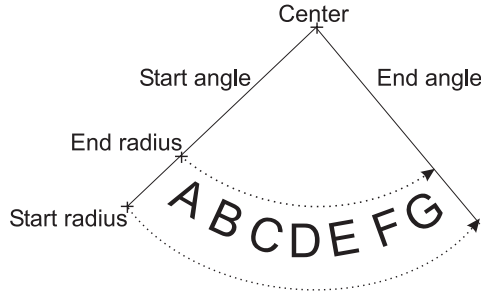
Synopsis Perform a polar-to-rectangular or rectangular-to-polar transform.

Format **void MimPolarTransform(SrcImageBufId, DestImageBufId, CenterPosX, CenterPosY, StartRadius, EndRadius, StartAngle, EndAngle, OperationMode, InterpolationMode, DestSizeXPtr, DestSizeYPtr)**

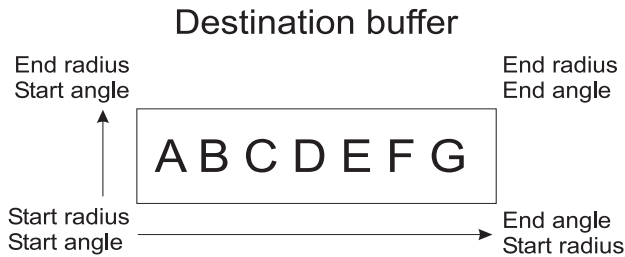
MIL_ID SrcImageBufId;	Source buffer identifier
MIL_ID DestImageBufId;	Destination buffer identifier
double CenterPosX;	X position of the center in the rectangular image
double CenterPosY;	Y position of the center in the rectangular image
double StartRadius;	Start radius of the zone of interest
double EndRadius;	End radius of the zone of interest
double StartAngle;	Start angle of the zone of interest
double EndAngle;	End angle of the zone of interest
long OperationMode;	Polar-to-rectangular or rectangular-to-polar transform
long InterpolationMode;	Interpolation used in the conversion
double *DestSizeXPtr;	Size X of the destination
double *DestSizeYPtr;	Size Y of the destination

Description This function performs a rectangular-to-polar or polar-to-rectangular transformation.

For a rectangular-to-polar transformation set the zone to convert by specifying the center coordinates (**CenterPosX** and **CenterPosY**), the start and end radius (**StartRadius** and **EndRadius**), and the start and end angle (**StartAngle** and **EndAngle**) as follows:



The result will be mapped to the destination buffer as follows:



The increment in angle is determined by the length (in pixels) of the outside arc, calculated as follows:

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

A polar-to-rectangular transform performs the reverse of the transform described above. It takes a polar buffer and maps it to a rectangular buffer. Use **CenterPosX**, **CenterPosY**, **StartAngle**, **EndAngle**, **StartRadius**, **EndRadius** parameters to specify where in the destination buffer the contents of the polar buffer will be mapped.

The **SrcImageBufId** parameter specifies the identifier of the source buffer.

The **DestImageBufId** parameter specifies the identifier of the destination buffer. To determine the required size of the destination buffer, call the function with **DestImageBufId** set to **M_NULL** and obtain the size in **DestSizeXPtr** and **DestSizeYPtr**.

The **CenterPosX** and **CenterPosY** parameters specify the X and Y coordinates of the center in the rectangular image.

The **StartRadius** and **EndRadius** parameters specify the start and end radius of the zone of interest (in pixels).

The **StartAngle** and **EndAngle** parameters specify the start and end of scan in the zone of interest. If the start angle is less than the end angle, the direction of the scan will be counter clockwise. If the start angle is greater than the end angle, the direction of the scan will be clockwise. The valid range for **StartAngle** and **EndAngle** is between -360 to 360 degrees and the maximum span must not exceed 360 degrees.

The **OperationMode** specifies the transform to perform. This can be set to:

M_RECTANGULAR_TO_POLAR	Perform rectangular-to-polar transform (default).
M_POLAR_TO_RECTANGULAR	Perform polar-to-rectangular transform.

The **InterpolationMode** parameter specifies the interpolation used in the conversion. It can be set to:

M_NEAREST_NEIGHBOR	Perform nearest neighbor interpolation.
M_BILINEAR	Perform bilinear interpolation.
M_BICUBIC	Perform bicubic interpolation.

Note that when using bicubic interpolation, saturation is performed according to the type of the destination buffer. For the other interpolation modes, no saturation is performed.

To specify how to determine the value of a destination pixel when its associated point falls outside the source buffer, you can add one of the following defines to the **InterpolationMode** parameter.

M_OVERSCAN_ENABLE	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, leave the destination pixel as is.
M_OVERSCAN_DISABLE	Leave the destination pixel as is.
M_OVERSCAN_CLEAR	Set the destination pixel to 0.

The default for interpolation mode and type is as follows:

M_DEFAULT	M_NEAREST_NEIGHBOR+M_OVERSCAN_ENABLE.
-----------	---------------------------------------

The **DestSizeXPtr** and **DestSizeYPtr** parameters specify the required width and height of the destination buffer. If these parameters are not set to M_NULL, the width and height will be returned.

Example mpolar.c

MimProject

Synopsis Project a 2D image into 1D.

Format **void MimProject(SrcImageBufId, ProjImResultId, ProjAngle)**

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID ProjImResultId;	Projection image processing result buffer identifier
double ProjAngle;	Angle of projection

Description This function projects a two-dimensional buffer into a one-dimensional buffer from the specified angle, and writes results to the specified projection result buffer. The results are generated by adding all pixel values along each diagonal in the image at the specified projection angle. The 90° projection of the image is known as the row profile; the 0° projection as the column profile.

Note that, if the sum of any diagonal is larger than the depth of the result buffer, the result will be undefined.

The **SrcImageBufId** parameter specifies the identifier of the data source of the image. This parameter must be given an image buffer identifier. The buffer must be one-band.

The **ProjImResultId** parameter specifies the identifier of the destination of the projection results. This parameter must be given the identifier of an image processing result buffer allocated with **MimAllocResult()** and an M_PROJ_LIST type. The buffer should have as many locations as there are lines to project in the image at the specified angle.

You can read the projection values from the result buffer, using **MimGetResult1d()** or **MimGetResult()**, specifying M_VALUE as the result type.

The **ProjAngle** parameter specifies the angle of projection, in degrees. Predefined projection angles are the following:

M_0_DEGREE	0° projection (column profile).
M_90_DEGREE	90° projection (row profile).

Status Only 0° and 90° projection angles are supported.

See also **MimAllocResult(), MimGetResult(), MimGetResult1d()**

MimRank

Synopsis Perform a rank filter on the pixels in an image.

Format **void MimRank(SrcImageBufId, DestImageBufId, StructElemBufId, Rank, ProcMode)**

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
MIL_ID StructElemBufId;	Identifier of a structuring element buffer to be used as a mask
long Rank;	Order of rank
long ProcMode;	Processing mode

Description This function performs a rank filter operation on the specified source buffer. It replaces each pixel with that pixel in its neighborhood whose value is the specified **rank***th* value in relation to others. It uses the specified structuring element as a mask. This mask determines the neighborhood size and which pixels in the neighborhood to ignore.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operator. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier.

The **StructElemBufId** parameter specifies the identifier of the structuring element buffer to use as a mask. You can either define a custom structuring element as a mask or use a predefined mask buffer.

To use a custom structuring element, you must have previously allocated it with **MbufAlloc1d()** or **MbufAlloc2d()** (specifying **M_STRUCT_ELEMENT** as the attribute) and loaded it with values, using **MbufPut()**. The structuring element buffer dimensions are used as the neighborhood size. Values in the structuring element set to **M_DONT_CARE** represent neighborhood pixels not to be considered in the ranking operation. All other values in the structuring element must be set to 1, representing neighborhood pixels to be considered in the ranking operation.

The following are predefined mask buffers:

M_3X3_RECT	Applies no mask and sets the neighborhood size to 3 x 3 pixels, in which there are 9 valid neighborhood pixels.
M_3X3_CROSS	Applies a cross (+) mask and sets the neighborhood size to 3 x 3 pixels, in which there are 5 valid neighborhood pixels; the cross mask makes the function ignore the pixels located in the four corners of the neighborhood.

The **Rank** parameter specifies which of the pixel values to select after valid neighborhood values are sorted in increasing order; valid neighborhood pixel values are those that are not masked-out. This parameter can be given a value in the range of 1 to the number of valid neighborhood pixels. If the number of valid pixels is less than the given rank, the number of valid pixels is used as the rank.

If this parameter is set to M_MEDIAN, **MimRank()** performs a median filter (that is, each pixel is replaced with the median neighborhood value).

The **ProcMode** parameter specifies the processing mode to use. This parameter can be set to the following:

M_BINARY	Non-zero pixels will be treated as ones (1) during processing and the resulting non-zero pixels will have the maximum value of the unsigned buffer (for example, 0xff for an 8-bit buffer).
M_GRAYSCALE	The source image's gray values are used for processing and the resulting buffer will also contain gray values (default).

Note This function is optimized for packed binary buffers.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

MimResize

Synopsis Resize an image.

Format `void MimResize(SrcImageBufId, DestImageBufId, ScaleFactorX, ScaleFactorY, InterpolationMode)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
double ScaleFactorX;	Scaling factor in X
double ScaleFactorY;	Scaling factor in Y
long InterpolationMode;	Interpolation mode

Description This function resizes the source image by the specified factors. Results are stored in the destination buffer starting from the top-left corner.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier.

The **ScaleFactorX** and **ScaleFactorY** parameters are used to multiply the width and height of the source image, respectively. These parameters can be independently set to either a non-null positive value or `M_FILL_DESTINATION`. When one of these parameters is set to `M_FILL_DESTINATION`, the source image is resized to fill the entire width and/or height of the destination buffer, depending on the parameter. A factor greater than 1.0 enlarges the source image, while a factor less than 1.0 reduces it.

The **InterpolationMode** parameter specifies the mode of interpolation. This parameter can be set to one of the following:

M_NEAREST_NEIGHBOR	Nearest neighbor interpolation.
M_BILINEAR	Bilinear interpolation.
M_BICUBIC	Bicubic interpolation.
M_AVERAGE	Averaging interpolation.
M_INTERPOLATE	Interpolated resizing: for zooming = bilinear, for dezooming = averaging. Gives the best speed/result compromise for interpolated resizing.

Note that when using bicubic interpolation, saturation is performed according to the type of the destination buffer. For the other interpolation modes, no saturation is performed.

To specify how to determine the value of a destination pixel when its associated point falls outside the source buffer, you can add one of the following defines to the **InterpolationMode** parameter:

M_OVERSCAN_ENABLE	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, leave the destination pixel as is. The default is M_OVERSCAN_ENABLE.
M_OVERSCAN_DISABLE	Leave the destination pixel as is.
M_OVERSCAN_CLEAR	Set the destination pixel to 0.

In addition, one of the following can be added to the interpolation mode to control the speed and precision of the interpolation.

M_FAST	Use the fast interpolation method. This method is less precise. The default setting is M_FAST.
M_REGULAR	Use the slow interpolation method. This method is more precise.

The **InterpolationMode** parameter can also be set to M_DEFAULT, which is equivalent to:
M_NEAREST_NEIGHBOR+M_OVERSCAN_ENABLE+M_FAST.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

MimRotate

Synopsis Rotate an image.

Format `void MimRotate(SrcImageBufId, DestImageBufId, Angle, SrcCenX, SrcCenY, DstCenX, DstCenY, InterpolationMode)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
double Angle;	Angle of rotation
double SrcCenX;	X-coordinate of source rotation center
double SrcCenY;	Y-coordinate of source rotation center
double DstCenX;	X-coordinate of destination rotation center
double DstCenY;	Y-coordinate of destination rotation center
long InterpolationMode;	Interpolation mode

Description This function rotates an image by the specified angle of rotation, using the specified interpolation mode. The center of rotation in the source image is determined by the specified X and Y source rotation-center coordinates. The rotated image will then be clipped to fit the destination buffer. It will be placed in the destination buffer with its center positioned at the specified X and Y destination center coordinates.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier.

The **Angle** parameter specifies the angle of rotation, in degrees. When a positive angle is specified, the function rotates the image in a counter-clockwise direction.

The **SrcCenX** and **SrcCenY** parameters specify the X and Y coordinates to use as the center of rotation in the source image. M_DEFAULT can be used to rotate the image about its true center.

The **DstCenX** and **DstCenY** parameters specify X and Y coordinates in the destination buffer. This is the location to which the specified center of the rotated source image will be mapped. If M_DEFAULT is specified as the coordinates, the true center of the destination buffer will be used.

The **InterpolationMode** parameter specifies the interpolation mode. It can be set to:

M_NEAREST_NEIGHBOR	Nearest neighbor interpolation.
M_BILINEAR	Bilinear interpolation.
M_BICUBIC	Bicubic interpolation.

Note that when using bicubic interpolation, saturation is performed according to the type of the destination buffer. For the other interpolation modes, no saturation is performed.

To specify how to determine the value of a destination pixel when its associated point falls outside the source buffer, you can add one of the following defines to the **InterpolationMode** parameter. The default is M_OVERSCAN_ENABLE.

M_OVERSCAN_ENABLE	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, leave the destination pixel as is.
M_OVERSCAN_DISABLE	Leave the destination pixel as is.
M_OVERSCAN_CLEAR	Set the destination pixel to 0.

Note that the **InterpolationMode** parameter can be set to M_DEFAULT, which is equivalent to M_NEAREST_NEIGHBOR+M_OVERSCAN_ENABLE.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

Examples mpatrot.c, mthread.c

MimShift

Synopsis Perform a point-to-point bit shift.

Format `void MimShift(SrcImageBufId, DestImageBufId, BitsToShift)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long BitsToShift;	Number of bits to shift

Description This function performs left or right bit-shifting on each pixel in the specified image. The shift operation is signed or unsigned depending on the source image buffer's data type.

Note that, if you shift by 0, only a copy operation will be performed.

The **SrcImageBufId** parameter specifies the identifier of the data source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the results. This parameter must be given an image buffer identifier.

The **BitsToShift** parameter specifies the number of bits to shift. If the given value is negative, each pixel in the specified image is right bit-shifted by the specified number of bits; otherwise, it is left bit-shifted.

Note Floating point values will be cast as unsigned long values before performing the shift operation (except when shifting by 0). Therefore, unexpected results can occur if a floating point value is larger than the unsigned long range.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

See also `MimTranslate()`

MimThick

Synopsis Perform a binary or grayscale thickening operation on an image.

Format `void MimThick(SrcImageBufId, DestImageBufId, NbIteration, ProcMode)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long NbIteration;	Number of operation iterations
long ProcMode;	Processing mode

Description This function performs a binary or grayscale thickening on the specified source image for the specified number of iterations.

The **SrcImageBufId** parameter specifies the identifier of the source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the resulting image. This parameter must be given an image buffer identifier.

The **NbIteration** parameter specifies the number of times to iterate the operation. If `M_TO_IDEMPOTENCE` is specified, all objects are enlarged until idempotence is reached.

The **ProcMode** parameter specifies the processing mode to use. This parameter can be set to the following:

M_BINARY	Non-zero pixels will be treated as ones (1) during processing and the resulting non-zero pixels will have the maximum value of the unsigned buffer (for example, 0xff for an 8-bit buffer).
M_GRAYSCALE	The source image's gray values are used for processing and the resulting buffer will also contain gray values.

Note This function is optimized for packed binary buffers.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

See also `MimMorphic()`, `MimThin()`

MimThin

Synopsis Perform a binary or grayscale thinning operation on an image.

Format `void MimThin(SrcImageBufId, DestImageBufId, NbIteration, ProcMode)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long NbIteration;	Number of operation iterations
long ProcMode	Processing mode

Description This function performs a binary or grayscale thinning on the specified source image for the specified number of iterations.

The **SrcImageBufId** parameter specifies the identifier of the source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the resulting image. This parameter must be given an image buffer identifier.

The **NbIteration** parameter specifies the number of times to iterate the operation. If `M_TO_SKELETON` is specified for this parameter, every object will be reduced to its skeleton.

The **ProcMode** parameter specifies the processing mode to use. This parameter can be set to the following:

M_BINARY	Non-zero pixels will be treated as ones (1) during processing and the resulting non-zero pixels will have the maximum value of the unsigned buffer (for example, 0xff for an 8-bit buffer).
M_GRAYSCALE	The source image's gray values are used for processing and the resulting buffer will also contain gray values.

Note This function is optimized for packed binary buffers.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

See also `MimMorphic()`, `MimThick()`

MimTransform

Synopsis Perform a Fast Fourier transform (FFT) or a Discrete Cosine transform (DCT).

Format **void MimTransform(SrcImageRBufId, SrcImageIBufId, DestImageRBufId, DestImageIBufId, TransformType, ControlFlag)**

MIL_ID SrcImageRBufID;	Source image buffer identifier (real part)
MIL_ID SrcImageIBufID;	Source image buffer identifier (imaginary part)
MIL_ID DestImageRBufID;	Destination image buffer identifier (real part)
MIL_ID DestImageIBufID;	Destination image buffer identifier (imaginary part)
long TransformType;	Perform an FFT or a DCT
long ControlFlag;	Perform a forward or reverse transform

Description This function performs forward and reverse FFT or DCT on an image.

The **SrcImageRBufID** and **SrcImageIBufID** parameters specify the identifiers of the source buffers for the real and imaginary components of the image, respectively. **SrcImageIBufID** can be set to M_NULL. For a reverse transform, the source buffers must be 32-bit signed integer or floating point buffers. For an FFT, the width and height of the source buffers must be a power of 2 and their type and pixel depth identical. For a DCT, the width and height of these buffers should be a multiple of 8.

The **DestImageRBufID** and **DestImageIBufID** parameters specify the identifiers of the destination buffers for the real and imaginary components of the image, respectively. **DestImageIBufID** can be set to M_NULL. For a forward transform, the destination buffers must be 32-bit signed integer or floating point buffers. For an FFT, the width and height of the destination buffers must be a power of 2 and their type and pixel depth identical. For a DCT, the width and height of these buffers should be a multiple of 8.

The **TransformType** parameter specifies the type of transform to be performed on the image.

M_FFT	Perform a Fast Fourier transform.
M_DCT8X8	Perform a Discrete Cosine transform on each 8x8 pixel block in the image.
M_DEFAULT	Same as M_FFT.

The **ControlFlag** parameter specifies if the transform is a forward transform or a reverse transform.

M_FORWARD or M_DEFAULT	Perform a forward transform on the image. Calculations for integer source buffers are performed in fixed-point format and returned to the destinations in 23.9 fixed point format for 8-bit sources or in 25.7 fixed point format for 16-bit sources. For 32-bit source buffers, the fixed point format of the destination buffer will be the same as that of the source. In this case, you must left-shift the image prior to a forward transform. If the destination buffer is float, the processing will be performed in floating-point arithmetic. In an FFT, if SrcImageBufID is set to M_NULL, a faster version of the forward transform will be performed.
M_REVERSE	Perform a reverse transform on the image. When performing a reverse transform into 8-bit or 16-bit destination buffers, the format of the source buffers is assumed to be in 23.9 and 25.7 fixed point format, respectively. For 32-bit integer destination buffers, the format of the source buffer will be same as that of the destination. In this case, you must right-shift the image after processing. If the source buffer is float, the processing will be performed in floating-point arithmetic. In an FFT, if DestImageBufID is set to M_NULL, a faster version of the reverse transform will be performed.

These optional controls can be added to the **ControlFlag**:

M_NORMALIZE	Normalize results (divide the final result by 8 for DCT and by (m x n) for FFT where m x n is the size of the image). Usually used with fixed-point arithmetic to avoid overflows.
M_1D_ROWS	Perform a 1-D transform on all rows of the image.
M_1D_COLUMNS	Perform a 1-D transform on all columns of the image.
M_CENTER	Center the real part and the imaginary part of the spectrum. The center of the spectrum is put at the (SizeX/2-1, SizeY/2-1).
M_MAGNITUDE	Compute the magnitude of the forward transform ($\sqrt{R^2 + I^2}$) and return its value in the real destination buffer.
M_PHASE	Compute the phase of the forward transform $\text{atan}(I/R)$ and return its value in the imaginary destination buffer. The phase is returned in the range of -180 to 180 degrees.

M_LOG_SCALE	Scale the magnitude of the forward FFT to be in the range of 0-255. This flag is used to scale the spectrum into a displayable range. It is used in combination with the M_MAGNITUDE control.
-------------	---

Only M_NORMALIZE is supported with both FFT and DCT. All other controls are supported only with the FFT.

If any of M_NORMALIZE, M_1D_ROWS, M_1D_COLUMNS or M_CENTER controls have been specified with the forward transform, the same controls must be specified for the reverse transform as well.

M_MAGNITUDE, M_PHASE and M_LOG_SCALE can only be used in forward transforms processed with floating point buffers.

Note that when the controls M_MAGNITUDE and M_PHASE are used, their results are returned in the real and imaginary buffers respectively, thereby overwriting the real and imaginary components of the image in the destination buffers. A reverse transform of these buffers will not yield a desirable result.

Example mfft.c

MimTranslate

Synopsis Translate an image in X and/or Y displacement.

Format `void MimTranslate(SrcImageBufId, DestImageBufId, XDisplacement, YDisplacement, InterpolationMode)`

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
double XDisplacement;	Displacement in X
double YDisplacement;	Displacement in Y
long InterpolationMode;	Interpolation mode

Description This function translates the source image position by the specified amount, writing results to the destination buffer. This function can be used to align images to subpixel accuracy before, for example, subtracting them.

The **SrcImageBufId** parameter specifies the source image buffer identifier.

The **DestImageBufId** parameter specifies the destination image buffer identifier.

The **XDisplacement** and **YDisplacement** parameters specify the amount by which to displace the source image. These parameters can be set to any positive or negative value.

The **InterpolationMode** parameter specifies the interpolation mode. This parameter can be set to:

M_DEFAULT	Bilinear interpolation (M_BILINEAR + M_OVERSCAN_ENABLE).
M_BILINEAR	Bilinear interpolation (M_BILINEAR only).

To specify how to determine the value of a destination pixel when its associated point falls outside the source buffer, you can add one of the following defines to the **InterpolationMode** parameter.

M_OVERSCAN_ENABLE	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, leave the destination pixel as is. This is the default.
M_OVERSCAN_DISABLE	Leave the destination pixel as is.
M_OVERSCAN_CLEAR	Set the destination pixel to 0.

Example msearch.c

MimWarp

Synopsis Perform a warping.

Format **void MimWarp(SrcImageId, DestImageId, WarpParam1Id, WarpParam2Id, OperationMode, InterpolationMode)**

MIL_ID SrcImageId;	Source buffer ID
MIL_ID DestImageId;	Destination buffer ID
MIL_ID WarpParam1Id;	1 st warp parameter buffer ID
MIL_ID WarpParam2Id;	2 nd warp parameter buffer ID or M_NULL
long OperationMode;	Operation mode
long InterpolationMode;	Interpolation mode

Description This function warps an image through either a first-order polynomial mapping or through look-up tables (LUTs).

A warping associates each pixel position of the destination buffer, (x_d, y_d) , with a specific point in the source buffer, (x_s, y_s) , and then determines the pixel value of (x_d, y_d) from its associated point and from a specified interpolation mode.

When using a first-order polynomial mapping, (x_d, y_d) gets associated with (x_s, y_s) through the following equations:

$$x_s = a_0x_d + a_1y_d + a_2$$

$$y_s = b_0x_d + b_1y_d + b_2$$

In this case, **WarpParam1Id** specifies the required coefficients $(a_0...a_2, b_0...b_2)$ and **WarpParam2Id** must be set to M_NULL. The coefficients can be automatically generated using **MgenWarpParameter()** or can be user-supplied.

When using LUTs, x_s is determined from (x_d, y_d) through one LUT and y_s is determined from (x_d, y_d) through another LUT. In this case, **WarpParam1Id** specifies the LUT for x_s and **WarpParam2Id** specifies the LUT for y_s . The LUTs can be user-supplied or, for 3x3 matrix-defined warplings, can be automatically generated using **MgenWarpParameter()**.

The **SrcImageId** parameter specifies the buffer on which to perform the warping. This buffer can be of any type.

The **DestImageId** parameter specifies the buffer in which to place the results of the warping. This buffer can be of any type.

The **WarpParam1Id** parameter specifies the buffer containing the $(a_0...a_2, b_0...b_2)$ coefficients or the LUT buffer from which x_s is determined.

When **WarpParam1Id** specifies the $(a_0...a_2, b_0...b_2)$ coefficients, the coefficients must be in a single-band 32-bit floating-point buffer that has an **M_ARRAY** attribute and that has dimensions 3x2 or 3x3. The first row specifies the a_n coefficients and the second row specifies the b_n coefficients. If the buffer is 3x3, the third row is ignored (it is assumed to be (0, 0, 1)) since **MimWarp0** does not directly perform polynomial warpings of second-order or higher.

When **WarpParam1Id** specifies the LUT buffer from which x_s is determined, the buffer must be signed 16- or 32-bit integer, have the same x and y size as the destination buffer, and have a **M_LUT** attribute.

The **WarpParam2Id** parameter specifies the LUT buffer from which y_s is determined. This buffer must be signed 16- or 32-bit integer, have the same x and y size as the destination buffer, and have a **M_LUT** attribute.

If you are not using LUTs to perform the warping, set **WarpParam2Id** to **M_NULL**.

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

M_WARP_POLYNOMIAL	Perform the warping through a first-order polynomial mapping.
M_WARP_LUT	Perform the warping through LUTs.

When performing the warping through LUTs, you need to specify the number of fractional bits for the source point (x_s, y_s) . To do so, add the define **M_FIXED_POINT + n** to **M_WARP_LUT**. If nothing is added to **M_WARP_LUT**, it is assumed that there are no fractional bits in the coordinates of the source point (and therefore interpolation is not required).

The **InterpolationMode** parameter specifies the interpolation mode. It can be set to:

M_NEAREST_NEIGHBOR	Nearest neighbor interpolation.
M_BILINEAR	Bilinear interpolation.
M_BICUBIC	Bicubic interpolation.

Note that when using bicubic interpolation, saturation is performed according to the type of the destination buffer. For the other interpolation modes, no saturation is performed.

To specify how to determine the value of a destination pixel when its associated point falls outside the source buffer, you can add one of the following defines to the **InterpolationMode** parameter. The default is M_OVERSCAN_ENABLE.

M_OVERSCAN_ENABLE	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, leave the destination pixel as is.
M_OVERSCAN_DISABLE	Leave the destination pixel as is.
M_OVERSCAN_CLEAR	Set the destination pixel to 0.

Note that the **InterpolationMode** parameter can be set to M_DEFAULT, which is equivalent to M_NEAREST_NEIGHBOR+M_OVERSCAN_ENABLE.

Example mwarp.c

See also **MgenWarpParameter()**

MimWatershed

Synopsis Perform a watershed transformation.

Format `void MimWatershed(SrcImageId, MarkerImageId, DestImageId, MinimumVariation, ControlFlag)`

MIL_ID SrcImageId;	Source buffer ID
MIL_ID MarkerImageId;	Marker buffer ID or M_NULL
MIL_ID DestImageId;	Destination buffer ID
long MinimumVariation;	Variation between extrema or M_NULL
long ControlFlag;	Control flag

Description This function performs a watershed transformation on the specified source buffer. The catchment basins of the source buffer can be determined from extrema (minima or maxima) in the source buffer and/or from a specified marker image. In the former case, a catchment basin is determined from an extrema when the difference in gray levels between it and its closest extrema is equal to or greater than the value specified by the **MinimumVariation** parameter. In the latter case, each group of touching pixels with the value zero in the marker image produces a catchment basin in the corresponding area of the source buffer; pixels in the marker image are considered touching if they are vertically, horizontally, or diagonally adjacent.

- You can specify that the destination buffer contain one of the following:
- Watershed lines. The watershed lines are given the value 0 and all other pixels are given the maximum value in the destination buffer.
 - Labelled catchment basins, without watershed lines. Each catchment basin is given a unique grayscale value, starting at 1.
 - Labelled catchment basins and watershed lines. Each catchment basin is given a unique grayscale value, starting at 1. Watershed lines are given the value 0.

The **SrcImageId** parameter specifies the buffer on which to perform the transform. This buffer can be 8-bit or 16-bit, signed or unsigned.

The **MarkerImageId** parameter specifies the buffer to use as a marker image. This buffer can be 8-bit or 16-bit, signed or unsigned. If you are not using a marker image, set this parameter to M_NULL.

The **DestImageId** parameter specifies the buffer in which to place the results of the transformation. This buffer can be 8-bit or 16-bit, signed or unsigned. The destination buffer can hold $2^x - 5$ catchment basins, where x refers to the number of bits per pixel in the buffer. Therefore, an 8-bit destination buffer can hold 251 catchment basins and a 16-bit destination buffer can hold 65531 catchment basins.

The **MinimumVariation** parameter specifies the minimum variation in gray-levels between extrema. If this parameter is set to M_DEFAULT (or 1), a catchment basin is produced from each extremum in the source buffer.

If you do not want catchment basins determined from extrema in the source buffer, that is, you only want them determined from a marker image, set **MinimumVariation** to M_NULL or M_OFF.

The **ControlFlag** parameter specifies how to perform the transformation. It can be set to any combination of the following five sets of flags. It can also be set to M_DEFAULT, in which case the default flag from each set is used (without M_SKIP_LAST_LEVEL). If no flag from a set is specified, its default flag is used.

M_WATERSHED (default) or M_BASIN or M_WATERSHED+M_BASIN	Have the destination buffer contain one of the following: watershed lines, labelled catchment basins, or labelled catchment basins and watershed lines.
M_MINIMA_FILL (default) or M_MAXIMA_FILL	When determining catchment basins from extrema in the source buffer, use the source buffer's minima or use its maxima.
M_REGULAR (default) or M_STRAIGHT_WATERSHED	Trace the watershed lines exactly or force them to be straight. Note that selecting M_STRAIGHT_WATERSHED automatically includes the M_SKIP_LAST_LEVEL flag.
M_SKIP_LAST_LEVEL	If included, this flag prevents a 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.
M_4_CONNECTED (default) or M_8_CONNECTED	Use 4-connected or 8-connected watershed lines.

Example msegment.c

MimZoneOfInfluence

Synopsis Perform a zone of influence detection.

Format void MimZoneOfInfluence(SrcImageBufId, DestImageBufId, OperationFlag)

MIL_ID SrcImageBufId;	Source image buffer identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long OperationFlag;	Operation flag

Description This function separates an image into zones, according to how much of a blob's surrounding background is within the blob's territorial boundaries, or "zone of influence". The image is considered to be binary, with background pixels equal to 0 (black), and all non-zero pixels treated as blobs. It gives every pixel in a blob's zone of influence the same value. Each zone of influence is numbered consecutively, beginning with 1. A blob's zone of influence consists of all pixels closer to that blob than to any other blob. There are as many zones as blobs.

The **SrcImageBufId** parameter specifies the identifier of the source of the operation. This parameter must be given an image buffer identifier.

The **DestImageBufId** parameter specifies the identifier of the destination of the resulting image. This parameter must be given an image buffer identifier.

The destination buffer should be deep enough to hold the maximum number of zones (blobs). The maximum label value is $2^n - 5$, where n is the depth of the destination buffer in bits. For example, an 8-bit buffer can be used for a maximum of 251 blobs and a 16-bit buffer can be used for a maximum of 65531 blobs. Note that if the destination buffer depth is too small, several zones might be given the same value.

The **OperationFlag** parameter controls the type of 3x3 distance matrix used for the operation and can be set to either M_CHESSBOARD (M_DEFAULT) or M_CHAMFER_3_4. The following table shows each flag's respective 3x3 distance matrix for calculating the distance to a neighboring pixel, which is then used for the zone of influence computation.

OperationFlag	3x3 Distance Matrix	Description
M_CHESSBOARD (or M_DEFAULT)	$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	This operation is faster than M_CHAMFER_3_4.
M_CHAMFER_3_4	$\begin{bmatrix} 4 & 3 & 4 \\ 3 & 0 & 3 \\ 4 & 3 & 4 \end{bmatrix}$	This operation uses a distance algorithm that makes a better approximation to actual Euclidean distance, and is therefore more accurate.

Status In-place processing is supported, but the source and destination image buffers cannot partially overlap (a situation that can only occur when using child buffers).

MmeasAllocContext

Synopsis Allocate a measurement context.

Format MIL_ID MmeasAllocContext(SystemId, ControlFlag, ContextIdPtr)

MIL_ID SystemId;	System identifier
long ControlFlag;	Allocation control flag
MIL_ID *ContextIdPtr;	Storage location for measurement context identifier

Description This function allocates a measurement context on the specified system. Measurement context settings are used to control the behavior of measurement operations (**MmeasFindMarker()** and **MmeasCalculate()**). When the measurement context is no longer required, you should release its memory, using **MmeasFree()**.

Note, upon allocation of an application, a default measurement context is automatically allocated. Rather than using **MmeasAllocContext()** to allocate a measurement context, you can use this default measurement context, by specifying M_DEFAULT wherever a measurement context identifier is required.

The **SystemId** parameter specifies the system on which the measurement context will be allocated. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. Specify M_DEFAULT_HOST to allocate on the default Host system of the current MIL application. Specify M_DEFAULT to have MIL select the most appropriate system on which to allocate the graphics context (it can be the default Host system or any already allocated system).

The **ControlFlag** parameter specifies the allocation control flag and must be set to M_DEFAULT.

The **ContextIdPtr** parameter specifies the address of the variable in which the measurement context identifier is to be written. If allocation fails, M_NULL is written as the identifier. Since the **MmeasAllocContext()** function also returns the marker identifier, you can set this parameter to M_NULL.

Return value The returned value is the measurement context identifier. If allocation fails, M_NULL is returned.

Default values These measurement context control types are set to the following default values upon allocation and can be changed at any time, using **MmeasControl()**:

ControlType	Default value
M_PIXEL_ASPECT_RATIO	1.0
M_PIXEL_ASPECT_RATIO_INPUT	M_CORRECTED
M_PIXEL_ASPECT_RATIO_OUTPUT	M_CORRECTED

See also **MmeasFree(), MmeasControl(), MmeasFindMarker(), MmeasCalculate()**

MmeasAllocMarker

Synopsis Allocate a measurement marker.

Format `MIL_ID MmeasAllocMarker(SystemId, MarkerType, ControlFlag, MarkerIdPtr)`

MIL_ID SystemId;	System identifier
long MarkerType;	Type of marker
long ControlFlag;	Allocation control flag
MIL_ID *MarkerIdPtr;	Storage location for marker identifier

Description This function allocates a measurement marker on the specified system. Once allocated, a marker's characteristics can be specified, using **MmeasSetMarker()**. For an edge or stripe marker, these characteristics are used as the criteria for finding the marker in a target image. An edge or stripe marker can be located in a target image and its measurements taken, using **MmeasFindMarker()**. Once found, an edge or stripe can then be used as a reference position in calculations involving two markers (**MmeasCalculate()**). A point marker cannot be searched for, but is placed in the required location as a reference position. When the marker is no longer required, release its memory, using **MmeasFree()**.

The **SystemId** parameter specifies the system on which the marker will be allocated. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. Specify M_DEFAULT_HOST to allocate on the default Host system of the current MIL application. Specify M_DEFAULT to have MIL select the most appropriate system on which to allocate the marker (it can be the default Host system or any already allocated system).

The **MarkerType** parameter specifies the type of marker to allocate. This parameter can be set to one of the following:

MarkerType	Description
M_POINT	A single point.
M_EDGE	An edge.
M_STRIPE	A pair of edges.

Any of the above types can be a multiple marker by changing the number of occurrences (M_NUMBER) with **MmeasSetMarker()**. A multiple marker is more than one instance of the same point, edge, or stripe characteristics.

The **ControlFlag** parameter specifies the allocation control flag. This parameter must be set to `M_DEFAULT`.

The **MarkerIdPtr** parameter specifies the address of the variable in which the marker identifier is to be written. Since the **MmeasAllocMarker()** function also returns the marker identifier, you can set this parameter to `M_NULL`. If allocation fails, `M_NULL` is written as the identifier.

Return value The returned value is the marker identifier. If allocation fails, `M_NULL` is returned.

Default values Marker characteristics are set to the following default values upon allocation and can be changed at any time, using **MmeasSetMarker()**. Note, a characteristic with the value `M_ANY` will not be considered as a criteria for finding the marker in a target image.

Characteristic	Default value
Number of points, edges, or stripes:	1
Spacing between edges or stripes:	<code>M_ANY</code>
Spacing variation:	<code>M_ANY</code>
Marker orientation:	<code>M_VERTICAL</code> (vertical orientation)
Marker polarity:	<code>M_ANY</code> for edge markers. <code>M_ANY</code> for the first (from top-left) edge of a stripe marker. <code>M_OPPOSITE</code> (opposite polarity from the first edge) for the second edge of a stripe marker.
Marker contrast:	<code>M_ANY</code>
Marker contrast variation:	<code>M_ANY</code>
Marker width:	<code>M_ANY</code>
Marker width variation:	<code>M_ANY</code>
Marker position:	<code>M_ANY</code>
Marker position variation:	<code>M_ANY</code>
Size of measurement box:	<code>M_DEFAULT</code> (image size)
Origin of measurement box:	Top-left corner of the image X-coordinate = 0.0 Y-coordinate = 0.0
Measurement box's center:	<code>M_DEFAULT</code> (image center)
Measurement box's angle:	0.0
Marker reference:	<code>M_DEFAULT</code> (center of the marker)

Characteristic	Default value
Edge strength:	M_ANY
Edge strength variation:	M_ANY
Edge threshold:	2.0 (2%)
Stripe inside edges:	M_ANY
Stripe inside-edge variation:	M_ANY
Stripe inside position:	M_ANY
Box angle mode:	M_DISABLE
Box angle delta negative and Box angle delta positive:	M_DEFAULT (180° search for a symmetrical stripe marker; 360°, complete rotation, for non-symmetrical)
Box angle tolerance:	5.0°
Box angle accuracy:	M_DISABLE
Box angle interpolation:	M_BILINEAR
Box angle reference:	M_BOX_CENTER
Weight factors:	Weight of 50% to the strength of an edge and the remaining 50% to any other characteristics not set to M_ANY.

Examples mmeas.c, mcalib.c

See also MmeasFree(), MmeasSetMarker(), MmeasFindMarker(), MmeasCalculate()

MmeasAllocResult

Synopsis

Allocate a measurement result buffer.

Format

MIL_ID MmeasAllocResult(SystemId, ResultType, MeasResultIdPtr)

MIL_ID SystemId;	System identifier
long ResultType;	Type of result buffer
MIL_ID *MeasResultIdPtr;	Storage location for measurement result buffer identifier

Description

This function allocates a buffer, on the specified system, to be used for storing measurement results obtained from a **MmeasCalculate()** operation. When the result buffer is no longer required, you should release its memory, using **MmeasFree()**. Note, a result buffer is not required for a **MmeasFindMarker()** operation.

The **SystemId** parameter specifies the system on which the result buffer will be allocated. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. Specify M_DEFAULT_HOST to allocate on the default Host system of the current MIL application. Specify M_DEFAULT to have MIL select the most appropriate system on which to allocate the buffer (it can be the default Host system or any already allocated system).

The **ResultType** parameter specifies the type of result buffer to allocate and must be set to M_CALCULATE.

The **ResultIdPtr** specifies the address of the variable in which the measurement result identifier is to be written. If allocation fails, M_NULL is returned as the identifier. Since the **MmeasAllocResult()** function also returns the measurement result identifier, you can set this parameter to M_NULL.

Return value

The returned value is the measurement result identifier. If allocation fails, M_NULL is returned.

See also

MmeasFree(), MmeasCalculate(), MmeasGetResult()

MmeasCalculate

Synopsis Calculate measurements between two markers.

Format **void MmeasCalculate(ContextId, Marker1Id, Marker2Id, MeasResultId, MeasurementList)**

MIL_ID ContextId;	Measurement context identifier
MIL_ID Marker1Id;	Identifier of first marker
MIL_ID Marker2Id;	Identifier of second marker
MIL_ID MeasResultId;	Measurement result identifier
long MeasurementList;	List of measurements to calculate

Description This function calculates the specified measurements between two markers. Edge and stripe markers must have been previously located in the target image with **MmeasFindMarker()**. The position of point markers must have been previously set, using **MmeasSetMarker()**.

The measurement context settings will control the behavior of this function and can be set with **MmeasControl()**.

Results are stored in the specified measurement result buffer and can be obtained using **MmeasGetResult()**.

The **ContextId** parameter specifies the identifier of the measurement context. This parameter must be given a valid measurement context identifier or can be set to M_DEFAULT, in which case, the default measurement context of the current MIL application will be used.

The **Marker1Id** and **Marker2Id** parameters specify the identifiers of the markers to use as the first and second reference positions for calculating measurements, respectively.

If both markers are multiple markers, then calculations are made using the edges or stripes of the first marker and the corresponding points, edges, or stripes in the second marker, for the entire first marker's number of points, edges, or stripes. The number of calculations performed is limited to the smallest number of points, edges, or stripes held in either marker (that is, if a marker contains only one point, edge, or stripe, then only one calculation is performed, regardless of the number of points, edges, or stripes contained in the first marker).

The **MeasResultId** parameter specifies the identifier of the result buffer in which to place results.

The **MeasurementList** parameter specifies which measurement(s) to calculate. The following lists the measurements that can be calculated. To calculate more than one measurement, add the predefined values together (for example, M_DISTANCE+M_ANGLE):

M_DEFAULT	Perform all of the calculations below.
M_ANGLE	Calculate the angle of the line joining two markers, relative to the positive X-axis. The value can be any between 0° and 360°.
M_DISTANCE	Calculate the distance between both markers.
M_LINE_EQUATION	Calculate the equation of the line joining both markers.

Example mmeas.c, mmeasmul.c

See also **MmeasGetResult()**, **MmeasSetMarker()**, **MmeasFindMarker()**

MmeasControl

Synopsis Control a measurement context setting.

Format `void MmeasControl(ContextId, ControlType, ControlValue)`

MIL_ID ContextId;	Measurement context identifier
long ControlType;	Type of control to set
double ControlValue;	Value of control type

Description This function sets the specified measurement context processing control. Measurement context settings control the behavior of measurement operations.

The **ContextId** parameter specifies the identifier of the measurement context.

The **ControlType** and **ControlValue** parameters specify the processing feature to control and the value needed for the control. Note, when referring to "data relative to the corrected image", this means that the data (for example, a specified position) is given as if it were relative to an image resized by a factor equal to the aspect ratio.

These parameters can be set to one of the following combinations:

ControlType	Description	ControlValue
M_PIXEL_ASPECT_RATIO	The ratio of the width of the pixel to its height.	Pixel width / pixel height. (default is 1.0)
M_PIXEL_ASPECT_RATIO_INPUT	How the measurement module interprets specified measurement characteristics relative to the pixel aspect ratio.	M_CORRECTED (default): Specified measurement characteristics are relative to the corrected image. M_NORMAL: Input data is relative to the given image. (ratio of 1.0)
M_PIXEL_ASPECT_RATIO_OUTPUT	How the measurement module returns results relative to the pixel aspect ratio.	M_CORRECTED (default): Results are relative to the corrected image. M_NORMAL: Output data is relative to the given image. (ratio: 1.0)

See also `MmeasAllocContext()`, `MmeasInquire()`

MmeasDraw

Synopsis Draw specific features of either a marker or result occurrence(s) in the destination image buffer.

Format **void MmeasDraw(GraphContId, MarkerOrResultId, DestImageId, Operation, Index, ControlFlag)**

MIL_ID GraphContId;	Graphics context identifier
MIL_ID MarkerOrResultId;	Identifier of the marker or result buffer
MIL_ID DestImageId;	Destination buffer identifier
long Operation;	Drawing operation
long Index;	Index of the result
long ControlFlag;	Control flag

Description This function provides graphics utilities to draw specific marker or result occurrence features in the destination buffer. **MmeasDraw()** can also be used with multiple results.

The **GraphContId** parameter specifies the graphics contexts to use. The graphics context must have been previously allocated using **MgraAlloc()**. Set this parameter to M_DEFAULT to use the default graphics context.

The **MarkerOrResultId** parameter specifies from which marker or result to extract the features to draw.

The **DestImageId** parameter specifies the destination image buffer identifier into which to draw. You can also choose to annotate an image non-destructively, by drawing into its display's overlay buffer.

The **Operation** parameter specifies the type of operation to perform. Operations can be added together to draw multiple features at a time. For example, to draw both the result occurrence position and active edges, you would specify M_DRAW_POSITION+M_DRAW_EDGES as the **Operation** parameter.

The possible **Operation** settings for a marker are:

Operation	Description
M_DRAW_BOX	Draws the measurement box, respecting the position and angle. Not available for a point marker.
M_DRAW_CENTER	Draws the measurement box center, respecting the position and angle. Not available for a point marker.

Operation	Description
M_DRAW_POSITION_VARIATION	Draws an H-type line (-) indicating possible position variation of the marker. The expected position variation must have been set with MmeasSetMarker() . Not available for a point marker.
M_DRAW_WIDTH_VARIATION	Draws an H-type line (-) indicating possible width variation of the marker. The expected width variation must have been set with MmeasSetMarker() . Not available for a point marker.

The possible **Operation** settings for results from a **MmeasFindMarker()** operation are:

Operation	Description
M_DRAW_EDGES_PROFILE	Draws the results' edge profile. M_DRAW_IN_BOX can be added to this setting to draw the edge profile at the position and angle of the measurement box. Not available for a point marker.
M_DRAW_SPACING	Draws an H-type line (-) indicating the found spacing. Not available for a point marker.
M_DRAW_LINE	Draws a line joining 2 markers.

The possible **Operation** settings for results from a **MmeasCalculate()** operation are:

Operation	Description
M_DRAW_LINE	Draws a line joining 2 markers.

The possible **Operation** settings for marker and results are:

Operation	Description
M_DRAW_POSITION	Draws a cross at the expected or found position and angle of the marker. For a stripe marker, M_EDGE_FIRST and M_EDGE_SECOND can be added to this flag but in this case, the expected width and orientation must have been set with MmeasSetMarker() .
M_DRAW_WIDTH	Draws an H-type line (-) at the possible or found width of the stripe marker. When drawing a stripe marker, the expected width must have been set with MmeasSetMarker() . Not available for a point or edge marker.
M_DRAW_EDGES	Draws a line along the edges of the marker at the expected or found position. Not available for a point marker.

The **Index** parameter specifies the index of the result to draw. Set index to M_ALL to draw all results. M_DEFAULT always refers to all results. When drawing features of a marker, the index must be set to M_DEFAULT. Note that when drawing a marker which is a multiple marker, only the first instance of the marker is drawn.

The **ControlFlag** parameter specifies whether to draw the definition of a marker (M_MARKER) or of a result (M_RESULT). M_DEFAULT is equivalent to M_RESULT.

See also **MmeasGetResult()**, **MmeasSetMarker()**

MmeasFindMarker

Synopsis Find a marker in an image and take the specified measurements.

Format `void MmeasFindMarker(ContextId, ImageBufId, MarkerId, MeasurementList)`

MIL_ID ContextId;	Measurement context identifier
MIL_ID ImageBufId;	Image identifier
MIL_ID MarkerId;	Marker identifier
long MeasurementList;	List of measurements to take

Description This function finds an edge or a stripe marker in the image and takes the measurements specified in the measurement list. The marker's characteristics are used to help locate the marker and can be set using **MmeasSetMarker()**.

Measurement context settings will control the behavior of this function and can be set, using **MmeasControl()**.

Results are stored with the marker (not in a result buffer), and can be obtained using **MmeasGetResult()**.

The **ContextId** parameter specifies the identifier of the measurement context. This parameter must be given a valid measurement context identifier or can be set to M_DEFAULT, in which case, the default measurement context of the current MIL application will be used.

The **ImageBufId** parameter specifies the identifier of the image buffer in which to locate the marker.

The **MarkerId** parameter specifies the identifier of the marker to be located in the image.

The **MeasurementList** parameter specifies which measurement(s) to perform. To take more than one measurement, add the predefined values together (for example, M_POSITION+M_ANGLE). Use M_DEFAULT to select all of the measurements.

It is recommended that the search region be set to an angle close to that of the required marker (that is, within the marker's particular rotational tolerance) to ensure that the marker is found and that results are accurate:

the greater the angle of the marker relative to the search region, the greater the distortion of the marker's actual characteristics or the chance that the marker may not be successfully located with **MmeasFindMarker()**.

Note, 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 measurement list can contain the following values:

MeasurementList	Description
M_DEFAULT	Take all of the measurements below.
M_ANGLE	Determine the angle of the marker relative to the positive X-axis of the target image. The value can be any between 0° and 360°.
M_CONTRAST	Determine the average grayscale difference between an edge and its background. For a stripe marker, this produces two values, one for each of the stripe marker's edges.
M_EDGE_INSIDE	Determine the number of edges located between the two exterior edges of a stripe marker.
M_LENGTH	Determine the length of the marker, in pixels or real-world units. The length is limited to the dimensions of the measurement box.
M_LINE_EQUATION	Determine the equation of the mean line following an edge. When performing this calculation for a stripe marker, three line equations are calculated: the line equation of the stripe's first edge, the line equation of the stripe's second edge, and the mean of the line equations of both its edges.
M_NUMBER	Determines the number of edges or stripes to find in the target image. (always selected)
M_POLARITY	Determine whether the edge (or edges of a stripe marker) are rising (positive polarity) or falling (negative polarity).
M_POSITION	Determine the X and Y coordinates of the center of the marker within the measurement box.
M_POSITION_VARIATION	Determine the position variation (or uncertainty) of the marker, in pixels or real-world units. If the marker is an edge, the position variation is equal to half of its width. If the marker is a stripe, the position variation is the sum of the position variations of both of its edges.
M_WIDTH	Determine the width of a marker in pixels or real-world units. The width of an edge marker is a measure of the transition in grayscale values. The width of a stripe marker is the average distance between its edges and is calculated at the angle of the measurement box.

M_WIDTH_VARIATION	Determine the width variation (or uncertainty) of a stripe marker, in pixels or real-world units. It is the sum of the position variations of both of its edges.
Note: The following values cannot be requested specifically. However, by specifying M_POSITION in the measurement list, all of the following will be also be accessible through MmeasGetResult() .	
M_BOX_EDGE_VALUES	Determine the edge value for every possible profile value of the measurement box. Therefore, for a vertical marker, the number of returned values is equal to the width of the measurement box; for a horizontal marker, the number is equal to the height. The value is represented as a normalized percentage of the image buffer's maximum possible value.
M_EDGE_STRENGTH	Determine the minimum/maximum edge value along the width of the edge marker. The value is represented as a normalized percentage of the image buffer's maximum possible value.
M_ORIENTATION	Determine the orientation of the marker.
M_POSITION_MAX	Determine the X and Y coordinates of the maximum position of an edge or stripe marker within the measurement box. The maximum position is always on the side of the measurement box furthest from the origin.
M_POSITION_MIN	Determine the X and Y coordinates of the minimum position of an edge or stripe marker within the measurement box. The minimum position is always on the side of the measurement box adjacent the origin, regardless of how the measurement box is rotated.
M_SPACING	Determine the inter-edge or inter-stripe spacing of a multiple marker.

Examples mcalib.c, mmeas.c, mmeasmul.c

See also MmeasGetResult(), MmeasSetMarker()

MmeasFree

Synopsis Free a measurement context, marker, or result buffer.

Format **void MmeasFree(MeasId)**

MIL_ID MeasId;	Measurement identifier
----------------	------------------------

Description This function deletes the specified marker, result buffer, or context identifier and releases any memory associated with it.

The **MeasId** parameter specifies the measurement identifier to free. The identifier must have been successfully allocated with **MmeasAllocMarker()**, **MmeasAllocResult()**, or **MmeasAllocContext()** prior to calling this function.

See also **MmeasAllocMarker(), MmeasAllocResult(), MmeasAllocContext()**

MmeasGetResult

Synopsis Get the results of measurements taken.

Format `void MmeasGetResult(MarkerOrMeasResultId, ResultType, FirstResultArrayPtr, SecondResultArrayPtr)`

MIL_ID MarkerOrMeasResultId;	Marker identifier or measurement result buffer identifier
long ResultType;	Type of measurement for which to get results
void *FirstResultArrayPtr;	Array in which to return first results
void *SecondResultArrayPtr;	Array in which to return second results (if any)

Description This function obtains the measurement result of the specified type from the specified marker or result buffer. Results should be obtained from a marker if an **MmeasFindMarker()** operation was performed or from a measurement result buffer if an **MmeasCalculate()** operation was performed.

The **MarkerOrMeasResultId** parameter specifies the identifier of the marker buffer (allocated with **MmeasAllocMarker()**) or measurement result buffer (allocated with **MmeasAllocResult()**) in which results are stored.

The **ResultType** parameter specifies the type of result to obtain.

FirstResultArrayPtr and **SecondResultArrayPtr** specify the address of the array in which to write results. With a multiple marker, results for each edge or stripe are held in an array. The results for each instance of the marker will be stored in a separate element of the specified arrays. Note that the arrays which you pass must be large enough to hold the results of all instances of the marker. For most result types, only **FirstResultArrayPtr** will be used and **SecondResultArrayPtr** should be set to M_NULL.

All positional results are relative to the top-left pixel in the target image or to the origin of the coordinate system of a calibrated image.

If results are being obtained from a marker, **ResultType** can be set to one of the values specified in the table that follows. To obtain results for a specific edge (either the first or second edge) of a stripe marker, add M_EDGE_FIRST

or M_EDGE_SECOND to **ResultType**. For example, M_LINE_EQUATION+M_EDGE_FIRST will return the line equation for the first edge of a stripe marker.

Result types which return a position always return the X coordinate to **FirstResultArrayPtr** and the Y coordinate to **SecondResultArrayPtr**.

For certain result types, two values can be returned for a stripe marker: the value for the first edge is returned to **FirstResultArrayPtr** and that of the second edge to **SecondResultArrayPtr**. For an edge marker, all results are returned to **FirstResultArrayPtr**.

ResultType	Description
M_ANGLE	The angle of the marker in degrees relative to the positive X axis. Returns a value between 0 and 360 degrees.
M_BOX_CORNER_BOTTOM_LEFT	The X and Y positions of the bottom-left corner of the measurement box.
M_BOX_CORNER_BOTTOM_RIGHT	The X and Y positions of the bottom-right corner of the measurement box.
M_BOX_CORNER_TOP_LEFT	The X and Y positions of the top-left corner of the measurement box.
M_BOX_CORNER_TOP_RIGHT	The X and Y positions of the top-right corner of the measurement box.
M_BOX_EDGE_VALUES	The edge values (as a normalized percentage of the buffer's maximum possible value for every profile value of the measurement box.) Results are returned to FirstResultArrayPtr as an array.
M_BOX_EDGE_VALUES_NUMBER	The number of values to be returned by M_BOX_EDGE_VALUES.
M_CONTRAST	The average grayscale difference between each marker's edge and its background. Returns one value for an edge marker; two for a stripe marker.
M_EDGE_FIRST+...	Results for the first exterior edge of a stripe for the requested characteristic. (for example, M_EDGE_FIRST+M_ANGLE)
M_EDGE_INSIDE	The number of edges located between the two exterior edges of a stripe marker.
M_EDGE_SECOND+...	Results for the second exterior edge of a stripe for the requested characteristic. (for example, M_EDGE_SECOND+M_ANGLE)

ResultType	Description
M_EDGE_STRENGTH	The maximum edge value variation (as a normalized percentage of the buffer's maximum possible value) of each edge of a marker. Returns one value for an edge marker, two for a stripe marker.
M_LENGTH	The length of the marker in pixels or real-world units.
M_LINE_EQUATION	The equation of the mean line of the marker. The slope is returned to the FirstResultArrayPtr and the Y intercept is returned to the SecondResultArrayPtr .
M_LINE_EQUATION_INTERCEPT	The Y intercept of the line equation of the marker.
M_LINE_EQUATION_SLOPE	The slope of the line equation of the marker.
M_MAX+...	Maximum value for the requested characteristic in all the edges or stripes found. Only one result is returned. (for example M_MAX +M_CONTRAST)
M_MEAN+...	Mean value for the requested characteristic in all the edges or stripes found. Only one result is returned. (for example M_MEAN+M_EDGE_STRENGTH)
M_MIN+...	Minimum value for the requested characteristic in all the edges or stripes found. Only one result is returned (for example M_MIN+M_SCORE).
M_NUMBER	Number of edges or stripes found in the measurement box.
M_ORIENTATION	The orientation of the marker.
M_POLARITY	The polarity of the marker (either M_NEGATIVE or M_POSITIVE). Returns one value for an edge marker; two for a stripe marker.
M_POSITION	The X and Y coordinates of the marker center in the image.
M_POSITION_MAX	The X and Y coordinates of the maximum position of an edge or stripe marker within the measurement box.
M_POSITION_MIN	The X and Y coordinates of the minimum position of an edge or stripe marker within the measurement box.

ResultType	Description
M_POSITION_VARIATION	The position variation of the marker, in pixels or real-world units (+/-).
M_SCORE	The confidence score for the find marker operation (as a percentage).
M_SPACING	Inter-edge or inter-stripe spacing between consecutive edges or stripes.
M_STANDARD_DEVIATION+...	Standard deviation of the values for the requested characteristic in all the edges or stripes found. Only one result is returned. (for example M_STANDARD_DEVIATION+M_WIDTH).
M_TOTAL_SCORE	The average confidence score of all edges or stripes found.
M_VALID_FLAG	The flag that denotes whether or not a marker was found (M_TRUE or M_FALSE).
M_WIDTH	The width of the marker in pixels or real-world units.
M_WIDTH_VARIATION	The width variation of the stripe marker in pixels or real-world units.

If results are being obtained from a measurement result buffer (**MmeasCalculate()**), the **ResultType** can be:

ResultType	Description
M_ANGLE	The angle, in degrees, of the line joining the two markers relative to the positive X axis. A value between 0 and 360 is returned.
M_DISTANCE	The distance, in pixels or real-world units, between two markers.
M_DISTANCE_X	The distance, in pixels or real-world units, on the X axis between two markers.
M_DISTANCE_Y	The distance, in pixels or real-world units, on the Y axis between two markers.
M_LINE_EQUATION	The equation of the line joining two markers. The slope is returned to the FirstResultArrayPtr and the Y intercept is returned to the SecondResultArrayPtr .
M_LINE_EQUATION_INTERCEPT	The Y intercept of the line equation joining two markers.
M_LINE_EQUATION_SLOPE	The slope of the line equation joining two markers.

When the equation of the line has an infinite slope, the value returned as the slope is M_INFINITE_SLOPE (1.0E+300). Results are returned as type "double". To have results returned as type "long", combine the result type with M_TYPE_LONG (for example, M_VALID_FLAG+M_TYPE_LONG). Note, M_INVALID is returned if a measurement was not calculated.

Examples mcalib.c, mmeas.c, mmeasmul.c

See also MmeasFindMarker(), MmeasCalculate(), MmeasAllocMarker(), MmeasAllocResult()

MmeasGetResultSingle

Synopsis Get a single result from a multiple marker or its result buffer.

Format `void MmeasGetResultSingle(MarkerOrMeasResultId, ResultType, FirstResultArrayPtr, SecondResultArrayPtr, ResultIndex)`

<code>MIL_ID MarkerOrMeasResultId;</code>	Marker identifier or measurement result buffer identifier
<code>long ResultType;</code>	Type of measurement for which to get results
<code>void *FirstResultArrayPtr;</code>	Array in which to return first results
<code>void *SecondResultArrayPtr;</code>	Array in which to return second results (if any)
<code>long ResultIndex;</code>	Index of the result to retrieve

Description This function obtains a specific measurement result of a specified type from the specified identifier. A specific instance is identified by a **ResultIndex** number from 0 to N-1, where N is the number of edges or stripes found or the number of results calculated.

The number of edges or stripes should be determined first by using `M_NUMBER` as the result type with **MmeasGetResult()** to ensure that the index of the result to retrieve is valid.

This function is otherwise identical to **MmeasGetResult()**: see **MmeasGetResult()** for more information, a list of result types, and their descriptions.

Note that in general, **FirstResultArrayPtr** and **SecondResultArrayPtr** should be single entry arrays, except when the result type returns multiple values, for example `M_BOX_EDGE_VALUES`.

See also **MmeasGetResult()**, **MmeasFindMarker()**, **MmeasCalculate()**, **MmeasAllocMarker()**, **MmeasAllocResult()**, **MmeasGetResultSingle()**

MmeasInquire

Synopsis Inquire about a measurement context, marker, or result buffer.

Format **long MmeasInquire(MeasId, InquireType, FirstUserVarPtr, SecondUserVarPtr)**

MIL_ID MeasId;	Identifier of a measurement context, marker, or measurement result buffer
long InquireType;	Setting to inquire
void *FirstUserVarPtr;	Storage location for first value
void *SecondUserVarPtr;	Storage location for second value

Description This function inquires about a specific marker characteristic, context-control setting, or measurement result buffer type.

The **MeasId** parameter specifies the identifier of the marker, measurement context, or measurement result buffer from which information will be obtained.

The **InquireType** parameter specifies the particular setting to inquire about. The **FirstUserVarPtr** and **SecondUserVarPtr** parameters specify the addresses of the variables in which the requested information is to be written. If only one value is to be returned, it will be returned to **FirstUserVarPtr** and **SecondUserVarPtr** should be set to **M_NULL**.

When performing an inquiry on a marker (to determine the default or user-defined value of a marker characteristic) **InquireType** can be set to values listed in the table below.

For certain inquire types, two values can be returned for a stripe marker: the value for the first edge is returned to **FirstUserVarPtr** and that of the second edge to **SecondUserVarPtr**. For an edge marker, all results are returned to **FirstUserVarPtr**.

Inquire types which return a position always return the X-coordinate to the **FirstUserVarPtr** and the Y-coordinate to the **SecondUserVarPtr**.

InquireType	Description
M_BOX_ANGLE	The angle of the measurement box (MmeasSetMarker()).
M_BOX_ANGLE_ACCURACY	The required precision for the resulting angle of the marker (MmeasSetMarker()).

InquireType	Description
M_BOX_ANGLE_DELTA_NEG and M_BOX_ANGLE_DELTA_POS	The range of angles to be searched for the marker: from (M_BOX_ANGLE - M_BOX_ANGLE_DELTA_NEG) to (M_BOX_ANGLE + M_BOX_ANGLE_DELTA_POS) inclusively, starting with an angle close to that of M_BOX_ANGLE (MmeasSetMarker()).
M_BOX_ANGLE_INTERPOLATION_MODE	The type of interpolation used when performing a search at an angle (MmeasSetMarker()).
M_BOX_ANGLE_MODE	Whether multiple-angle search is enabled (MmeasSetMarker()).
M_BOX_ANGLE_REFERENCE	The center of rotation used when performing a search at an angle (MmeasSetMarker()).
M_BOX_ANGLE_TOLERANCE	The rotation tolerance of the marker. This 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. This determines the step angle used for a multiple-angle search (MmeasSetMarker()).
M_BOX_CENTER	The X and Y coordinates of the measurement box's center (MmeasSetMarker()).
M_BOX_ORIGIN	The X and Y coordinates of the measurement box's origin (MmeasSetMarker()).
M_BOX_SIZE	The width and height of the measurement box (MmeasSetMarker()). The width is returned to FirstUserVarPtr and the height is returned to SecondUserVarPtr .
M_CONTRAST	The contrast of a marker (one value for an edge; two for a stripe) (MmeasSetMarker()).
M_CONTRAST_VARIATION	The contrast variation of a marker (MmeasSetMarker()).
M_CONTROL_FLAG	The control flag given at marker allocation time (MmeasAllocMarker()).
M_EDGE_INSIDE	The number of edges between the external edges of a stripe marker (MmeasSetMarker()).

InquireType	Description
M_EDGE_INSIDE_VARIATION	The tolerance of the number of edges between the external edges of a stripe marker (MmeasSetMarker()).
M_EDGE_STRENGTH	The maximum edge value variation of the edge marker (as a percentage) (MmeasSetMarker()).
M_EDGE_STRENGTH_VARIATION	The tolerance of the maximum edge value of the edge marker (MmeasSetMarker()).
M_EDGE_THRESHOLD	The minimum edge value in order to consider an edge in the search. The value is represented as a percentage of the image buffer's maximum possible value (MmeasSetMarker()).
M_MARKER_REFERENCE	The X and Y-offsets of the marker reference (MmeasSetMarker()).
M_MARKER_TYPE	The type of marker. (MmeasAllocMarker()).
M_NUMBER	The number of edges or stripes.
M_NUMBER_MIN	The minimum number of edges or stripes.
M_ORIENTATION	The orientation of the marker (MmeasSetMarker()).
M_POLARITY	The polarity of a marker (MmeasSetMarker()).
M_POSITION	The X and Y coordinates of the marker center (MmeasSetMarker()).
M_POSITION_INSIDE_STRIPE	Whether the specified position is inside or outside of the stripe (M_YES, M_NO, or M_ANY) (MmeasSetMarker()).
M_POSITION_VARIATION	The tolerance of position variation (MmeasSetMarker()).
M_POSITION_X	The X coordinate of the marker center (MmeasSetMarker()).
M_POSITION_Y	The Y coordinate of the marker center (MmeasSetMarker()).
M_SPACING	The typical spacing, in pixels, between consecutive edges or stripes (MmeasSetMarker()).
M_SPACING_VARIATION	The typical variation in spacing, in pixels, between edges or stripes (MmeasSetMarker()).

InquireType	Description
M_WEIGHT_FACTOR+ M_EDGE_STRENGTH or M_CONTRAST or M_POSITION or M_WIDTH or M_EDGE_INSIDE or M_SPACING	The specific weight assigned to the specified characteristic (as a percentage) (MmeasSetMarker()).
M_WIDTH	The width of a stripe marker (MmeasSetMarker()).
M_WIDTH_VARIATION	The width variation of a marker (MmeasSetMarker()).

When performing an inquiry on a measurement context buffer, **InquireType** can be set to one of the following values.

InquireType	Description
M_CONTROL_FLAG	The control flag given at context allocation time (MmeasAllocContext()).
M_PIXEL_ASPECT_RATIO	The pixel aspect ratio (MmeasControl()).
M_PIXEL_ASPECT_RATIO_INPUT	The input pixel aspect ratio interpretation mode (MmeasControl()).
M_PIXEL_ASPECT_RATIO_OUTPUT	The output pixel aspect ratio mode (MmeasControl()).

When performing an inquiry on a measurement result buffer, **InquireType** can be set to M_RESULT_TYPE to determine the type of result buffer allocated (**MmeasAllocResult()**).

Results are returned to **FirstUserVarPtr** and **SecondUserVarPtr** as type "double".

To have results returned as type "long", combine the **InquireType** with M_TYPE_LONG (for example, M_MARKER_TYPE+M_TYPE_LONG).

Return value The returned value is the requested first value buffer information, cast to long.

See also **MmeasAllocMarker()**, **MmeasAllocResult()**, **MmeasAllocContext()**, **MmeasSetMarker()**, **MmeasControl()**

MmeasRestoreMarker

Synopsis Restore a marker from disk.

Format **MIL_ID MmeasRestoreMarker(FileName, SystemId, ControlFlag, MarkerIdPtr)**

MIL_TEXT_PTR FileName;	Marker file name
MIL_ID SystemId;	System identifier
long ControlFlag;	Control flag
MIL_ID *MarkerIdPtr;	Storage location for marker identifier

Description This function restores a previously saved marker from disk and returns a handle to it. It also restores all the marker characteristics that were in effect when the marker was saved.

The **FileName** parameter specifies the marker file name. The function internally handles the opening and closing of the file.

The **SystemId** parameter specifies the system on which the measurement marker will be restored. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. Specify M_DEFAULT_HOST to allocate on the default Host system of the current MIL application. Specify M_DEFAULT to have MIL select the most appropriate system on which to allocate the marker (it can be the default Host system or any already allocated system).

The **ControlFlag** parameter specifies the restore control flag and should be set to M_DEFAULT.

The **MarkerIdPtr** parameter specifies the address of the variable in which the marker identifier is to be written. Since the **MmeasRestoreMarker()** function also returns the marker identifier, you can set this parameter to M_NULL. If allocation fails, M_NULL is written as the identifier.

Return value The returned value is the marker identifier. If allocation fails, M_NULL is returned.

See also **MmeasSaveMarker()**

MmeasSaveMarker

Synopsis Save a marker to disk.

Format **void MmeasSaveMarker(FileName, MarkerId, ControlFlag)**

MIL_TEXT_PTR FileName;	Marker file name
MIL_ID MarkerId;	Marker identifier
long ControlFlag;	Control flag

Description This function saves an allocated marker to disk, including all of its current characteristics. The marker and its characteristics can later be restored, using **MmeasRestoreMarker()**.

The **FileName** parameter specifies the name of the file in which to save the marker. If this file already exists, it will be overwritten. The function internally handles the opening and closing of this file.

The **MarkerId** parameter specifies the identifier of the marker to save.

The **ControlFlag** parameter specifies the save control flag and must be set to M_DEFAULT.

See also **MmeasAllocMarker(), MmeasRestoreMarker()**

MmeasSetMarker

Synopsis Set a marker characteristic.

Format `void MmeasSetMarker(MarkerId, CharacteristicToSet, FirstValue, SecondValue)`

MIL_ID MarkerId;	Marker identifier
long CharacteristicToSet;	Characteristic to set
double FirstValue;	First value
double SecondValue;	Second value or M_NULL

Description This function sets a marker characteristic. Marker characteristics are used in determining the marker location in a target image.

MmeasFindMarker() locates the edge(s) or stripe(s) that best correspond to the specified marker's characteristics. A marker's characteristics should describe the marker as accurately as possible to ensure that it will be successfully located in an image. See **MmeasAllocMarker()** for the default values of these characteristics.

The **MarkerId** parameter specifies the identifier of the marker.

The **CharacteristicToSet** parameter specifies the type of characteristic to set. The **FirstValue** and **SecondValue** parameters specify the values for the parameter being set. The **FirstValue** must always be provided. The **SecondValue** should only be provided when required; otherwise, it should be set to M_NULL.

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). It is possible to override this default by adding M_WEIGHT_FACTOR to these characteristics and specifying the percentage of importance. When specifying weight factors it is recommended, in order to better control the search, that you assign a weight factor to all enabled characteristics which support weight factors to a total of 100%. Characteristics that can be weighted in this way are M_EDGE_STRENGTH, M_CONTRAST, M_POSITION, M_WIDTH, M_EDGE_INSIDE, and M_SPACING.

Various marker characteristics and their possible values can be set as shown in the following tables.

The **CharacteristicToSet**, **FirstValue** and **SecondValue** parameters can be set to the following values:

CharacteristicToSet	Description	Values
<i>For point, edge, and stripe markers:</i>		
M_BOX_ANGLE	The angle of the measurement box. Note, if set to M_ANY, the contents of the measurement box are analyzed to determine the angle of the marker. The measurement box's center of rotation is set with M_ANGLE_REFERENCE (default is M_BOX_CENTER). For a multiple point marker, the box angle determines the angle from which subsequent points are placed (at the interval set with M_SPACING), proceeding from the coordinates defined by M_POSITION, M_POSITION_X, or M_POSITION_Y.	Set FirstValue to a value from 0 to 360 degrees (default is 0) or to M_ANY. Set SecondValue to M_NULL.
M_NUMBER	The number of edges or stripes to locate in the measurement box. 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.	Set FirstValue to the number of edges or stripes. Set the SecondValue to M_NULL. Default is M_ALL.
M_POSITION	X and Y coordinates of the center of the marker in the target image (approximate). The defined position must be within the measurement box (taking into account the angle or scope of the angular search, if enabled). Note, the position is ignored for edge or stripe markers when M_NUMBER is greater than 1.	Numerical values valid within the target image or M_ANY (default). Set FirstValue to the X coordinate. Set SecondValue to the Y coordinate.
M_POSITION_X	X-coordinate of the marker in the target image (approximate). Note, the position cannot be set for a multiple marker.	Set FirstValue to a numerical value valid within the target image or M_ANY. Set SecondValue to M_NULL.

CharacteristicToSet	Description	Values
M_POSITION_Y	Y-coordinate of the marker in the target image (approximate). Note, the position is ignored for edge or stripe markers when M_NUMBER is greater than 1.	Set FirstValue to a numerical value valid within the target image or M_ANY. Set SecondValue to M_NULL.
M_SPACING	The typical spacing, or distance, between consecutive edges or stripes.	Set FirstValue to the required spacing in pixels. M_SAME applies the average spacing of all located edges or stripes. Default is M_ANY. Set the SecondValue to M_NULL.
<i>For edge and stripe markers:</i>		
M_BOX_ANGLE_ACCURACY	The accuracy of the angular search. This determines the size of the step angle to use once the approximate location of the marker is found.	Set FirstValue to a value from 0.1 to 180 degrees or to M_DISABLE to force an angle of accuracy equal to the angle of tolerance. Set SecondValue to M_NULL.
M_BOX_ANGLE_DELTA_NEG and M_BOX_ANGLE_DELTA_POS	The positive and negative range of angles within which to search for the marker: from M_BOX_ANGLE_DELTA_NEG to M_BOX_ANGLE_DELTA_POS inclusively, starting with an angle close to that of M_BOX_ANGLE.	Set FirstValue to a value from 0.1 to 360 degrees or to M_DEFAULT (default of 180° for a symmetrical stripe marker, 360° for an edge marker or non-symmetrical stripe marker). Set SecondValue to M_NULL.
M_BOX_ANGLE_INTERPOLATION_MODE		
	The type of interpolation used when M_BOX_ANGLE__MODE is enabled.	Set FirstValue to M_NEAREST_NEIGHBOR, M_BILINEAR (default), or M_BICUBIC.
M_BOX_ANGLE_MODE	Enable the use of multiple-angle search, as specified by the settings of other M_BOX_ANGLE... characteristics. The angle with the highest match score is returned.	Set FirstValue to M_ENABLE or M_DISABLE (default, search only at the angle specified by M_BOX_ANGLE). Set SecondValue to M_NULL.
M_BOX_ANGLE_REFERENCE	The center of rotation used when M_BOX_ANGLE is not zero.	Set FirstValue to M_BOX_CENTER (default) or M_BOX_ORIGIN.
M_BOX_ANGLE_TOLERANCE	The rotation tolerance of the marker. This 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. This determines the step angle used for a multiple-angle search.	Set FirstValue to a value from 0.1 to 360 degrees or to M_DEFAULT (5.0°). Set SecondValue to M_NULL.

CharacteristicToSet	Description	Values
M_BOX_CENTER	The X and Y-coordinates of the measurement box's center. These coordinates are relative to the top-left corner of the target image.	Numerical values valid within the target image or M_DEFAULT. Set FirstValue to the X coordinate; SecondValue to the Y coordinate. When M_DEFAULT is specified, the target image's corresponding center coordinate is used.
M_BOX_ORIGIN	The X and Y-coordinates of the measurement box's top-left corner. These coordinates are relative to the top-left corner of the target image.	Numerical values valid within the target image or M_DEFAULT. Set FirstValue to the X coordinate; SecondValue to the Y coordinate. When M_DEFAULT is specified, the coordinate(s) [0, 0] of the target image's top-left pixel are used.
M_BOX_SIZE	The width and height of the measurement box.	Numerical values valid within the target image or M_DEFAULT. Set FirstValue to the width and SecondValue to the height. When M_DEFAULT is specified, the target image's width and/or height is used.
M_CONTRAST	Typical grayscale difference between an edge and its background. For a stripe marker the values should be the typical grayscale difference for each of its edges.	Set FirstValue to a numerical value between 0 and the maximum value of the buffer (that is, 255 for an 8-bit buffer and 65535 for a 16-bit buffer) or to M_ANY (default). Set SecondValue to M_NULL for an edge marker or, for the second edge of a stripe marker, to the appropriate value, M_SAME (to specify that the two edges of the stripe have approximately the same contrast), or M_ANY.
M_CONTRAST_VARIATION	Used with M_CONTRAST, this characteristic specifies the contrast variation of an edge marker. For a stripe marker, this value should be the average of the contrast variation for each of its edges.	Set FirstValue to a numerical value between 0 and the maximum value of the buffer (that is, 255 for an 8-bit buffer and 65535 for a 16-bit buffer) or to M_ANY (default). Set SecondValue to M_NULL.

CharacteristicToSet	Description	Values
M_EDGE_STRENGTH	The maximum/minimum edge value along the width of the edge. The sign of the edge value represents the polarity of the edge. For example, in an 8-bit image buffer, the maximum pixel value is 255. Therefore, a 50% edge strength in this buffer represents a maximum difference in average adjacent profile values of 128 and a rising edge.	Set FirstValue to a value between 0 and 100. This value is represented as a normalized percentage of the maximum pixel value possible for the specific image buffer. Set to M_ANY (default) to select the strongest edge value encountered in the search. Set SecondValue for second edge of a stripe or to M_NULL.
M_EDGE_STRENGTH_VARIATION	Used with M_EDGE_STRENGTH, this characteristic specifies the tolerance of the maximum (+/-) edge value of the edge marker. For a stripe marker, the edge strength variation applies to both edges.	Set FirstValue to a value between 0 and 100. This represents a normalized percentage of the image buffer's maximum possible value (for example, with an 8-bit image buffer, 10% is an edge strength variation of 25.5). Default setting is M_ANY. Set SecondValue to M_NULL.
M_EDGE_THRESHOLD	The edge value threshold beneath which a grayscale variation is not considered an edge.	Set FirstValue to a value between 0.0 and 100.0 or M_DEFAULT (2.0). The value represents a percentage of the maximum grayscale value permissible in the target image buffer.*
	<p>* For example, with a value of 2.0, a grayscale variation is considered an edge only if it is greater than 2% of the maximum value of the buffer multiplied by the box size.</p> <p>To consider all grayscale variation as edges, irrespective of their values, set FirstValue to 0.0. However, due to the increased number of edges, MmeasFindMarker() will execute relatively slowly.</p>	
M_MARKER_REFERENCE	The X and Y-offsets of the marker's reference position relative to the marker's center.	Numerical values valid within the target image or M_DEFAULT. Set FirstValue to the X offset and SecondValue to the Y offset. M_DEFAULT corresponds to the marker center (that is, X-offset = 0 and Y-offset = 0).

CharacteristicToSet	Description	Values
M_NUMBER_MIN	The minimum number of edges or stripes to locate in the measurement box.	Set FirstValue to the minimum number of edges or stripes. Set the SecondValue to M_NULL. Default is M_NUMBER.
M_ORIENTATION	The orientation of an edge or stripe marker.	Set FirstValue to: M_VERTICAL (default) if the marker has a vertical orientation. M_HORIZONTAL if the marker has a horizontal orientation. M_ANY if the marker orientation is not known. Set SecondValue to M_NULL.
M_POSITION_VARIATION	The tolerance of position variation.	Set FirstValue to a numerical value valid within the target image or M_ANY (default). Set SecondValue to M_NULL.
M_POLARITY	The polarity specifies whether an edge is a rising edge (increase in grayscale value) or a falling edge (decrease in grayscale value).	When setting this characteristic for an edge marker, set FirstValue to the polarity and SecondValue to M_NULL. When setting this characteristic for a stripe marker, set FirstValue to the polarity of the stripe marker's first (from top-left) edge and set SecondValue to the polarity of its second edge. M_POSITIVE: rising edge. M_NEGATIVE: falling edge. M_ANY: not a criteria (default for edges and first stripe edge). M_OPPOSITE: the specified edge of a stripe marker has an opposite polarity to the other edge. (default for second stripe edge) M_SAME: the specified edge of a stripe marker has the same polarity as the other edge.
M_SPACING_VARIATION	Inter-edge or inter-stripe spacing tolerance.	Set FirstValue to the variation in pixels, or M_ANY to ignore variations (default). Set the SecondValue to M_NULL.

CharacteristicToSet	Description	Values
<i>For weighting purposes only:</i>		
M_WEIGHT_FACTOR +M_EDGE_STRENGTH or M_WEIGHT_FACTOR +M_CONTRAST or M_WEIGHT_FACTOR +M_POSITION or M_WEIGHT_FACTOR +M_WIDTH or M_WEIGHT_FACTOR +M_SPACING or M_WEIGHT_FACTOR +M_EDGE_INSIDE	Add a specific weight to the specified characteristic.	The value assigned to each characteristic represents the percentage of weight assigned to that characteristics. The sum of values of the M_WEIGHT_FACTOR+M... characteristics must be 100. Default assigns 50% to M_EDGE_STRENGTH and 50% to all other characteristics.
<i>For stripe markers only:</i>		
M_POSITION_INSIDE_STRIPE	The location of the given position (M_POSITION) relative to the stripe. If the position is defined as being within the stripe, the search algorithm proceed outwards in both directions from that point. If defined as outside, no stripe including the position is considered.	Set FirstValue to: M_YES if the position is inside the stripe. M_NO if the position is outside the stripe. M_ANY if the position can be either inside or outside the stripe. Set SecondValue to M_NULL.
M_EDGE_FIRST	Used to set values for the first exterior edge of a stripe for the specified characteristic. For example, M_EDGE_FIRST+M_ANGLE.	Set to the required values for the chosen characteristic.
M_EDGE_INSIDE	Typical number of edges occurring inside a stripe marker.	Set FirstValue to a numerical value or to M_ANY. Set SecondValue to M_NULL.
M_EDGE_INSIDE_VARIATION	The maximum variation (tolerance) in the number of inside edges of a stripe.	Set FirstValue to a numerical value or to M_ANY. Set SecondValue to M_NULL.
M_EDGE_SECOND	Used to set values for the second exterior edge of a stripe for the specified characteristic. For example, M_EDGE_FIRST+M_ANGLE.	Set to the required values for the chosen characteristic.

CharacteristicToSet	Description	Values
M_WIDTH	Typical width of a stripe marker, in pixels (the distance between the marker's edges).	Set FirstValue to a numerical value valid within the target image or M_ANY (default). M_SAME applies the average width of all located stripes. Set SecondValue to M_NULL.
M_WIDTH_VARIATION	Used with M_WIDTH, this characteristic specifies the maximum variation (tolerance) of a stripe's width.	Set FirstValue to a numerical value valid within the target image or M_ANY (default). Set SecondValue to M_NULL.
<i>For weighting purposes only:</i>		
M_WEIGHT_FACTOR +M_WIDTH or M_WEIGHT_FACTOR +M_EDGE_INSIDE	Add a specific weight to the specified characteristic.	The value assigned to each characteristic represents the percentage of weight assigned to that characteristic. The sum of values of the M_WEIGHT_FACTOR+M... characteristics must be 100. Default assigns 50% to M_EDGE_STRENGTH and 50% to all other characteristics.

Examples mcalib.c, mmeas.c, mmeasmul.c

See also MmeasAllocMarker(), MmeasFindMarker(), MmeasGetResult()

MmodAlloc

Synopsis Allocate a model finder context.

Format `MIL_ID MmodAlloc(SystemId, ModelFinderType, ControlFlag, ContextIdPtr)`

MIL_ID SystemId;	System identifier
long ModelFinderType;	Type of model finder context
long ControlFlag;	Operation control flag
MIL_ID ContextIdPtr;	Storage location for context identifier

Description This function allocates a model finder context on the specified system. A model finder context contains all the information necessary to perform a **MmodFind()** search, including global search settings, and the individual model(s) to locate. When the model finder context is no longer required, you should release its memory, using **MmodFree()**.

Models can be defined and added to a model finder context using **MmodDefine()**.

Model finder context and individual model search settings can be adjusted using **MmodControl()**.

The **SystemId** parameter specifies the system on which to allocate the models. This parameter must be set to a valid system identifier, `M_DEFAULT_HOST`, or `M_DEFAULT`. To use the default Host system of the current MIL application, specify `M_DEFAULT_HOST`. If you specify `M_DEFAULT`, MIL will select the most appropriate system on which to allocate the models (the Host system or any already allocated system).

The **ModelFinderType** parameter specifies the type of model finder context and must be set to the following:

ModelFinderType	Description
<code>M_GEOMETRIC</code>	General geometric search, utilizing geometric features to locate user-specified model(s).

The **ControlFlag** parameter is reserved for future expansion and must be set to `M_DEFAULT`.

The **ContextIdPtr** parameter specifies the address of the variable in which to write the model finder context identifier. Since the **MmodAlloc()** function also returns the model finder context identifier, you can set this parameter to M_NULL.

Return value The returned value is the model finder context's identifier. If allocation fails, M_NULL is returned.

Example MmodFind.c

See also MmodControl()

MmodAllocResult

Synopsis Allocate a model finder result buffer.

Format MIL_ID MmodAllocResult(SystemId, ControlFlag, ModResultIdPtr)

MIL_ID SystemId;	System identifier
long ControlFlag;	Operation control flag
MIL_ID ModResultIdPtr;	Storage location for model finder result buffer identifier

Description This function allocates a result buffer, on the specified system, to store results obtained from a **MmodFind()** operation. When the result buffer is no longer required, release its memory, using **MmodFree()**.

The **SystemId** parameter specifies the system on which to allocate the result buffer. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to allocate the buffer (the Host system or any already allocated system).

The **ControlFlag** parameter is reserved for future expansion and must be set to M_DEFAULT.

The **ModResultIdPtr** specifies the address of the variable in which to write the model finder context result buffer identifier. Since the **MmodAllocResult()** function also returns the model finder context identifier, you can set this parameter to M_NULL.

Return value The returned value is the model finder result buffer identifier. If allocation fails, M_NULL is returned.

Example MmodFind.c

See also MmodGetResult()

MmodControl

Synopsis Control a model finder context or individual model setting.

Format `void MmodControl(ContextId, Index, ControlType, ControlValue)`

MIL_ID ContextId;	Identifier of model finder context
long Index;	Index of the model
long ControlType;	Type of control to set
double ControlValue;	Value of control

Description This function sets the specified control for either the model finder context itself, or one (or all) of the models contained therein. These settings control the execution of **MmodFind()** operations.

The **ContextId** parameter specifies the model finder context whose settings you want to modify. The model finder context must have been previously allocated on the system using **MmodAlloc()**.

The **Index** parameter specifies which model finder context, or model contained therein, to control. To specify general model finder context settings, set **Index** to M_CONTEXT. To specify the settings of an individual model, set **Index** to the model's index. If **Index** is set to M_ALL, the same control setting will be applied to all models. M_DEFAULT passed as an index always refers to M_ALL. User labels cannot be used as indices; to retrieve the index of a model from its user label, use **MmodInquire()** with M_INDEX_FROM_LABEL.

The **ControlType** parameter specifies the setting to change, while the **ControlValue** parameter specifies its new value. All values are set in pixels, regardless of whether the model is calibrated. All these settings can be inquired by **MmodInquire()**. Changing certain **ControlType** settings can require preprocessing of the model finder context again, using **MmodPreprocess()**. To know if the model finder context needs to be preprocessed, call **MmodInquire()**, with the M_PREPROCESSED inquire type.

The following **ControlType** and corresponding **ControlValue** setting can be specified for both M_CONTEXT and models:

ControlType	Control Values	Description																																																																														
M_NUMBER	1,2,3,...M_ALL, or M_DEFAULT	<p>For M_CONTEXT, M_NUMBER specifies the maximum number of all model occurrences (for all models within the context together) to find in the target image. Once the number of occurrences for the context with scores and target scores above their certainty levels has been reached, the search will stop, regardless of whether the actual number set for any individual model has been found.</p> <p>M_ALL finds the maximum number of occurrences specified for each model. The default value (M_DEFAULT) is M_ALL.</p> <p>For a model, M_NUMBER specifies the maximum number of occurrences of the specified model to find in the target image. M_ALL finds any number of occurrences. The default value is 1.</p> <p>Note that setting this control type to M_ALL for the context and/or the model(s) can increase the search time; always set this control type to a specific number whenever possible.</p> <p>To further illustrate the relationship between the M_NUMBER setting for the context and that of individual models, the table below shows the maximum possible results which can be returned for different context M_NUMBER settings, for a sample context with three models.</p> <table><tr><th colspan="2">Settings for M_NUMBER</th><th colspan="7">Maximum possible results</th></tr><tr><td>Context:</td><td>M_ALL</td><td colspan="7"></td></tr><tr><td>Model 0:</td><td>1</td><td colspan="7">1</td></tr><tr><td>Model 1:</td><td>M_ALL</td><td colspan="7">All</td></tr><tr><td>Model 2:</td><td>2</td><td colspan="7">2</td></tr><tr><td>Context:</td><td>6</td><td colspan="7"></td></tr><tr><td>Model 0:</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>Model 1:</td><td>M_ALL</td><td>6</td><td>5</td><td>5</td><td>4</td><td>4</td><td>3</td></tr><tr><td>Model 2:</td><td>2</td><td>0</td><td>0</td><td>1</td><td>1</td><td>2</td><td>2</td></tr></table>	Settings for M_NUMBER		Maximum possible results							Context:	M_ALL								Model 0:	1	1							Model 1:	M_ALL	All							Model 2:	2	2							Context:	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
Settings for M_NUMBER		Maximum possible results																																																																														
Context:	M_ALL																																																																															
Model 0:	1	1																																																																														
Model 1:	M_ALL	All																																																																														
Model 2:	2	2																																																																														
Context:	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																																																																									

The possible **ControlType** and corresponding **ControlValue** settings for M_CONTEXT are:

ControlType	Control Values	Description
M_ACCURACY	M_MEDIUM, M_HIGH, M_DEFAULT.	Sets the accuracy with which to find occurrences. The precision achieved is dependent on the quality of the model and the image. The default setting is M_MEDIUM. Note that higher accuracy can slow down the MmodFind() operation.
M_DETAIL_LEVEL	M_MEDIUM, M_HIGH, M_VERY_HIGH, M_DEFAULT.	Sets the level of details to extract from the model source and target images. The detail level determines what is considered an edge/background. The default setting is M_MEDIUM.
M_SEARCH_ANGLE_RANGE	M_ENABLE, M_DISABLE, M_DEFAULT	<p>Sets whether to force the algorithm to search for model occurrences in the specified angular range (M_ANGLE_DELTA_NEG, M_ANGLE_DELTA_POS) or not. The nominal angle of the search is set for each model using the M_ANGLE control type. Disable this control type to search at the nominal angle (M_ANGLE) only. The default setting is M_ENABLE.</p> <p>When set to M_DISABLE, MIL can still find occurrences within the specified angular range (depending on the characteristics of the actual model). If you want to ensure that MIL finds occurrences within a specific range of angles only, you should specify the required positive and negative deltas (the default for both is 180 degrees).</p>

ControlType	Control Values	Description
M_SEARCH_SCALE_RANGE	M_ENABLE, M_DISABLE, M_DEFAULT	Sets whether the algorithm searches for model occurrences in the model's specified scale range (M_SCALE_MIN_FACTOR, M_SCALE_MAX_FACTOR) or not. The default setting is M_DISABLE. Disable this control type to search at the nominal scale (M_SCALE) only. The nominal scale of the search is set for each model using the M_SCALE control type. When search with scale is enabled (M_ENABLE), occurrences will be sought in the full valid scale range (M_SCALE_MIN_FACTOR and M_SCALE_MAX_FACTOR). Note that a greater search scale range can slow down the MmodFind() operation. However, even when search with scale is set to M_DISABLE, MIL can typically find occurrences in a range of 0.95-1.05 of the nominal scale (M_SCALE).
M_SHARED_EDGES	M_ENABLE, M_DISABLE, M_DEFAULT	Sets whether to enable sharing of edges between occurrences. M_DISABLE (default) specifies that edges that are part of more than one occurrence are considered as part of the occurrence with the greatest score only.
M_SMOOTHNESS	0.0-100.0, or M_DEFAULT	Sets the degree of smoothness applied to the model source images and target images in the edge extraction. The default setting (M_DEFAULT) is 50.
M_SPEED	M_MEDIUM, M_HIGH, M_VERY_HIGH, M_DEFAULT.	Sets the algorithm's search speed. The default setting is M_MEDIUM. Note that increasing the search speed can decrease the robustness and sub-pixel accuracy of the MmodFind() operation.
M_TIMEOUT	Double value or M_DISABLE	Sets the maximum search time for MmodFind() , in msec. If a search times out, the number of occurrences found at that point will be returned. The default setting is 2000 msec.





The possible **ControlType** and corresponding **ControlValue** settings for models are:

ControlType	Control Values	Description
M_ACCEPTANCE	0.0-100.0	Sets the acceptance level for the score. 100% indicates that for every active edge in the model, a corresponding edge must be found in the occurrence with a perfect fit (generally hard to obtain). The default value is 60%.
M_ACCEPTANCE_TARGET	0.0-100.0	Sets the acceptance level for the target score. The default is 0%, meaning any number of extra edges is tolerated (100% allows for no extra edges in the occurrence and the occurrence must have a perfect fit).
M_ANGLE	Double value (0.0-360.0) or M_DEFAULT	Sets the nominal search angle, relative to the model reference axis angle. Default is 0.0 degrees.
M_ANGLE_DELTA_NEG	1.0-180.0 or M_DEFAULT	Sets the possible clockwise angle variation, relative to the nominal search angle (M_ANGLE). Occurrences with angles outside this range cannot be returned as results. Default value is 180.0 degrees. Note, that M_SEARCH_ANGLE_RANGE must be enabled to force searching through the specified angular range. Otherwise, the search is performed at the angle specified with M_ANGLE (nominal angle).

ControlType	Control Values	Description
M_ANGLE_DELTA_POS	1.0-180.0 or M_DEFAULT	Sets the possible counter-clockwise angle variation, relative to the nominal search angle (M_ANGLE). Occurrences with angles outside this range cannot be returned as results. Default value is 180.0 degrees. Note, that M_SEARCH_ANGLE_RANGE must be enabled to force searching through the specified angular range. Otherwise, the search is performed at the angle specified with M_ANGLE (nominal angle).
M_ASSOCIATED_CALIBRATION	Calibration object identifier, or M_NULL	Associates the specified calibration object with the specified model. M_NULL removes association between the calibration object and this model. Note that you must reassociate (if necessary) each model with its calibration object after restoring a model finder context using MmodRestore() , since the calibration object is not saved with the model finder context.

ControlType	Control Values	Description
M_CERTAINTY	0.0-100.0	Sets the certainty level for the score, as a percentage. Default value is 80%. If both the score and target scores (see below) are above their respective certainty levels, the occurrence is considered a match, without searching the rest of the target image for better matches (provided the specified number of occurrences has been found). If you set the certainty level too high (close to 100%), you slow down the search because you force the search algorithm to check the whole search region for the best possible match(es). A good certainty level is slightly lower than the expected score, so that the search can finish as soon as a match is found. However, if you set the certainty level too low, false matches might be found.
M_CERTAINTY_TARGET	0.0-100.0	Sets the certainty level for the target score. If both the score and target scores (see above) are above their respective certainty levels, the occurrence is considered a match, without searching the rest of the target image for better matches (provided the specified number of occurrences has been found). The default is 0%.
M_FIT_ERROR_WEIGHTING_FACTOR	0.0-100.0	Sets the fit error weighting factor. This factor determines the contribution of the fit error in the score and target score calculation. The higher the percentage, the greater the contribution of the fit error in determining the score and target score. The default is 25%.

ControlType	Control Values	Description
M_MIN_SEPARATION_ANGLE	$0.0 < x \leq 180.0$, M_DISABLE, or M_DEFAULT	Sets the minimum angular separation required for two occurrences to be considered distinct matches. This value is specified as an absolute angle value. Default value is 10.0. When disabled (M_DISABLE), occurrences are never distinct for this control type. Note, only one of the separation criteria needs to be met for occurrences to be considered distinct.
M_MIN_SEPARATION_SCALE	$1.0 < x \leq 4.0$, M_DISABLE, or M_DEFAULT	Sets the minimum separation required in scale for two occurrences to be considered distinct matches. This value is specified as a scale factor. Default value is 1.1. When disabled (M_DISABLE), occurrences are never distinct for this control type. Note, only one of the separation criteria needs to be met for occurrences to be considered distinct.
M_MIN_SEPARATION_X	Double value, as a percentage, M_DISABLE, or M_DEFAULT	Sets the minimum separation required along the X-axis for two occurrences to be considered distinct matches. This value is specified as a percentage of the model's width at M_SCALE. This value can be greater than 100%. The default value is 10.0. When disabled (M_DISABLE), occurrences are never distinct for this control type. Note, only one of the separation criteria needs to be met for occurrences to be considered distinct.

ControlType	Control Values	Description
M_MIN_SEPARATION_Y	Double value, as a percentage, M_DISABLE, or M_DEFAULT	Sets the minimum separation required along the Y-axis for two occurrences to be considered distinct matches. This value can be greater than 100%. The default value is 10.0. When disabled (M_DISABLE), occurrences are never distinct for this control type. This value is specified as a percentage of the model's height at M_SCALE. Note, only one of the separation criteria needs to be met for occurrences to be considered distinct.
M_POLARITY	M_SAME, M_REVERSE, M_SAME_OR_REVERSE, M_ANY	<p>Sets the expected polarity of occurrences, compared to that of the model. Default value is M_SAME.</p> <div> <div> <div>Model</div>  </div> <div> <div>Occurrences</div> <div>  <div>M_SAME polarity</div> </div> <div>  <div>M_REVERSE polarity</div> </div> <div>  <div>M_ANY polarity</div> </div> <div> <div></div> <div>M_SAME_OR_REVERSE polarity</div> </div> </div> </div>
M_REFERENCE_ANGLE	Double value (0.0-360.0) or M_DEFAULT	Sets the angle of the reference axis for the model. Angle results are returned relative to this axis, rather than the model source image axis. The default value is 0.0 degrees.
M_REFERENCE_X	Double value, in pixels, or M_DEFAULT	Sets the X-offset of the reference axis of the model, relative to the origin of the model source image. The default value is the center of the model.
M_REFERENCE_Y	Double value, in pixels, or M_DEFAULT	Sets the Y-offset of the reference axis of the model, relative to the origin of the model source image. The default value is the center of the model.
M_SCALE	Double value (0.5-2.0) or M_DEFAULT	Sets the nominal search scale. Default value is 1.0.

ControlType	Control Values	Description
M_SCALE_MAX_FACTOR	Double value (1.0-2.0) or M_DEFAULT	Sets the factor used to determine the maximum permitted scale. M_SCALE is multiplied by this factor. Default value is 2.0. Note, that M_SEARCH_SCALE_RANGE must be enabled to search through the specified scale range. However, occurrences can still be found in the scale range even if M_SEARCH_SCALE_RANGE is disabled.
M_SCALE_MIN_FACTOR	Double value (0.5-1.0) or M_DEFAULT	Sets the factor used to determine the minimum permitted scale. M_SCALE is multiplied by this factor. Default value is 0.5. Note, that M_SEARCH_SCALE_RANGE must be enabled to search through the specified scale range. However, occurrences can still be found in the scale range even if M_SEARCH_SCALE_RANGE is disabled.
M_SEARCH_OFFSET_X	Double value, in pixels, M_ALL*, or M_DEFAULT.	Sets the search region's offset in the X direction from the target image's origin. Note, the search region limits the possible reference axis positions that can be returned as results for a model occurrence; coordinates which fall outside of this region cannot be returned as results. The default value is M_ALL*.
M_SEARCH_OFFSET_Y	Double value, in pixels, M_ALL*, or M_DEFAULT.	Sets the search region's offset in the Y direction from the target image's origin. Note, the search region limits the possible reference axis positions that can be returned as results for a model occurrence; coordinates which fall outside of this region cannot be returned as results. The default value is M_ALL*.

ControlType	Control Values	Description
M_SEARCH_SIZE_X	Double value, in pixels, M_ALL*, or M_DEFAULT.	Sets the search region width. Note, the search region limits the possible reference axis positions which can be returned for a model occurrence; coordinates which fall outside of this region cannot be returned as results. Therefore, the search region can be smaller than the actual model size. The default value is M_ALL*.
M_SEARCH_SIZE_Y	Double value, in pixels, M_ALL, or M_DEFAULT.	Sets the search region height. Note, the search region limits the possible reference axis positions which can be returned for a model occurrence; coordinates which fall outside of this region cannot be returned as results. Therefore, the search region can be smaller than the actual model size. The default value is M_ALL*.
*When M_SEARCH_OFFSET_X, M_SEARCH_OFFSET_Y, M_SEARCH_SIZE_X, and M_SEARCH_SIZE_Y are set to M_ALL (default), the search region is defined as the entire image plane, meaning that all coordinates (even outside the target image) can be returned as results. If search region offsets are specified, setting M_SEARCH_SIZE_X and M_SEARCH_SIZE_Y to M_ALL defines the search region as the positive half of the image plane, relative to the offsets.		
M_USER_LABEL	Integer, M_DEFAULT, or M_NO_LABEL	Sets a user defined label for the specified model. This label can be used as a means of identifying your model, independently from its index in the model finder context. Default value is M_NO_LABEL. Note, only M_DEFAULT and M_NO_LABEL control types can be used with this function when Index is set to M_ALL.

Example MmodFind.c

See also MmodAlloc(), MmodDefine(), MmodFind(), MmodInquire(), MmodPreprocess()

MmodDefine

Synopsis Add a model to, or delete a model from, a model finder context.

Format **void MmodDefine(ContextId, ModelType, Param1, Param2, Param3, Param4, Param5)**

MIL_ID ContextId;	Identifier of model finder search context
long ModelType;	Model type
double Param1;	First parameter
double Param2;	Second parameter
double Param3;	Third parameter
double Param4;	Fourth parameter
double Param5;	Fifth parameter

Description This function allows you to add a model to, or delete a model from, a model finder context. Models can be defined from any 1-band 8-bit unsigned image. If the model source image is calibrated, then that calibration object is automatically associated with the model. Models can use different calibration objects within the same model finder context, provided all calibrations map to the same world. However, you cannot mix calibrated and non-calibrated models within a model finder context; all models must be either calibrated or not.

When a call to **MmodFind()** is made, all models that are part of the specified context are searched for simultaneously in the target image. Note, when a model is added or deleted, the search context must be preprocessed again, using **MmodPreprocess()**.

The search is performed according to the general search settings specified in the model finder search context (M_CONTEXT) as well as the individual model search settings. Both the context and individual model search settings can be specified using **MmodControl()**.

The model index starts at 0, and each subsequent model added to the model finder context is given a sequential index number, in the order that the model was added. If a model is deleted, all entries with higher indices are shifted down one.

Refer to the *Guidelines for choosing models* section in The MIL Geometric Model Finder chapter of the Matrox Imaging Library User Guide for more information on defining models.

The **ContextId** parameter specifies the model finder context to which to add, or from which to delete, the model. The model finder context must have been previously allocated on the system using **MmodAlloc()**.

The **ModelType** parameter specifies the type of model to define when adding models to the context. To delete models from a model finder context, set this parameter to M_DELETE.

The **Param1-Param5** parameters are variable parameters and their definitions are dependent on the model type chosen.

The following table describes each **ModelType** parameter setting and its respective **Param1** to **Param5** definitions:

ModelType	Description & Param1-5	
M_IMAGE	Defines a model from the specified source image (Param1). The default settings use the entire image as the model. The minimum and maximum model sizes are 16.0 x 16.0 and 1024.0 x 1024.0 pixels, respectively. If the model source image is calibrated, then that calibration object is automatically associated with the model. Models can use different calibration objects within the same model finder context, provided all calibrations map to the same world. However, you cannot mix calibrated and non-calibrated models within a model finder context; all models must be either calibrated or not.	
	Param1	Identifier of the image buffer from which to extract the model (model source image). Only 1-band 8-bit unsigned images can be used to define a model.
	Param2	Offset X of the model origin in the source image, in pixels. The default (M_DEFAULT) is the origin of the model source image.
	Param3	Offset Y of the model origin in the source image, in pixels. The default (M_DEFAULT) is the origin of the model source image.
	Param4	Width of the model (in X), in pixels. The default (M_DEFAULT) is the width of the model source image. The minimum model width is 16.0 pixels and the maximum model width is 1024.0 pixels.
	Param5	Height of the model (in Y), in pixels. The default (M_DEFAULT) is the height of the model source image. The minimum model height is 16.0 pixels and the maximum model height is 1024.0 pixels.

ModelType	Description & Param1-5	
M_DELETE	Deletes a model from the context. The index of the model must be specified as Param1.	
	Param1	Model index. User labels cannot be used as indices; to retrieve the index of a model from its user label, use <i>MmodInquire()</i> with M_INDEX_FROM_LABEL.
	Param2-5	Must be set to M_DEFAULT.

Example MmodFind.c

See also MmodAlloc(), MmodControl(), MmodMask(), MmodPreprocess()

MmodDraw

Synopsis Draw specific features of the model or result occurrence in the destination image buffer.

Format **void MmodDraw(GraphContId, ContextOrResultId, DestImageId, Operation, Index, ControlFlag)**

MIL_ID GraphContId;	Graphics context identifier
MIL_ID ContextOrResultId;	Identifier of the model finder context or result buffer
MIL_ID DestImageId;	Destination image buffer identifier
long Operation;	Drawing operation
long Index;	Index of the model or result
long ControlFlag;	Control flag

Description This function draws specific features of models or result occurrences in the destination buffer.

The **GraphContId** parameter specifies the graphics context to use when drawing. The graphics context must have been previously allocated using **MgraAlloc()**. Set this parameter to M_DEFAULT to use the default graphics context.

The **ContextOrResultId** parameter specifies the model finder context or result buffer from which to extract the features to draw. The model finder context or result buffer must have been previously allocated on the system using **MmodAlloc()** or **MmodAllocResult()**.

The **DestImageId** parameter specifies the identifier of the destination image buffer in which to draw. This destination image can be any supported MIL image. You can also choose to annotate an image non-destructively, by drawing into its display's overlay buffer. When using **MmodDraw()** to draw features extracted from model source images or target images that are associated to a calibration object, the destination drawing buffer must also be calibrated to the same world. **MmodDraw()** draws all the requested features according to whether the destination image buffer is physically corrected or not.

The **Operation** parameter specifies the type of operation to perform. Operations can be added together to draw multiple features at a time. For example, to draw both the result occurrence's position and active edges, you

would set the **Operation** parameter to M_DRAW_POSITION+M_DRAW_EDGES. The **Operation** parameter can be set to a combination of the following values:

Operation	Description
M_DRAW_BOX	Draws a bounding box around the occurrence. Note that when drawing an occurrence, the bounding box is drawn maintaining the angle and scale of the occurrence.
M_DRAW_POSITION	Draws a cross-like symbol at the model reference axis position or occurrence position. The cross is drawn maintaining the angle of the model or occurrence.
M_DRAW_IMAGE	Draws the region of the model source image from which the model was extracted (copy). Available for models only.
M_DRAW_EDGES	Draws the active edges of the model or result occurrence.
M_DRAW_DONT_CARES	Draws the model's "don't care" pixels. Available for models only.
M_DRAW_FLAT_REGION	Draws the model's "flat regions". Available for models only.

The **Index** parameter specifies the index of the model in the specified model finder context, or of the result occurrence. Set **Index** to M_ALL to draw all models or all occurrences. M_DEFAULT refers to index 0 for models and M_ALL for results. User labels cannot be used as indices; to retrieve the index of a model from its user label, use **MmodInquire()** with M_INDEX_FROM_LABEL.

The **ControlFlag** parameter specifies, when drawing a model, whether to draw at the top-left corner of the destination image (M_DEFAULT) or at the offsets used to define the model region in the source image (M_ORIGINAL). When drawing features from a result buffer, this parameter must be set to M_DEFAULT.

Example MmodFind.c

See also MgraAlloc()

MmodFind

Synopsis

Searches for the model(s) of the specified model finder context in the target image buffer.

Format

MIL_ID ContextId;

Identifier of model finder context

MIL_ID TargetImageId;

Target image identifier

MIL_ID ModResultId;

Model finder result buffer identifier

Description

This function searches for the model(s) defined in the specified model finder context in the target image and writes the results in the provided result buffer. The search is performed using both the model finder context's and each model's current search settings (see **MmodControl()**). Note, the model finder context must be preprocessed (**MmodPreprocess()**) before calling this function. The resulting values can be read with the **MmodGetResult()** function.

The **ContextId** parameter specifies the model finder context to use for the search. The model finder context must have been previously allocated on the system using **MmodAlloc()**.

The **TargetImageId** parameter specifies the target image in which to search for the models. Target images must be 1-band 8-bit unsigned images. The minimum and maximum target image sizes are 16x16 and 32768x32768 pixels, respectively. If you are using a model finder context whose models have been defined from calibrated model source images, the target image must also be calibrated, using the same world. Conversely, if your model finder context is uncalibrated, your target images must be uncalibrated as well.

The **ModResultId** parameter specifies the result buffer in which to write the results of the search. The result entries are ordered by match score, starting with the highest score (regardless of model). In general, positional results are returned in pixels, relative to the top-left pixel in the target image. If you are using a calibrated target image, positional results are automatically returned in the output coordinate system specified in the associated calibration object of the target image. To change the output coordinate system, use **McalControl()** with **M_OUTPUT_COORDINATE_SYSTEM** set to **M_PIXEL** or **M_WORLD** to return all

results in pixel units or world units, respectively. Alternatively, to transform specific results from calibrated units to pixel units, use **McalTransformResult()**.

Example `MmodFind.c`

See also `MmodAlloc()`, `MmodControl()`, `MmodDefine()`, `MmodGetResult()`

MmodFree

Synopsis Free a model finder context or a result buffer.

Format **void MmodFree(ObjectId)**

MIL_ID ObjectId;	Model finder context identifier or result buffer identifier
------------------	---

Description This function deletes the specified model finder context (and all its models) or result buffer identifier, and releases any memory associated with it. Note that to only delete individual models in the context, use **MmodDefine()**.

All model finder contexts allocated on a particular system must be freed before the system can be freed.

The **ObjectId** specifies the identifier of the model finder context or result buffer to free. These must have been successfully allocated (with **MmodAlloc()** or **MmodAllocResult()**) prior to calling this function.

See also **MmodAlloc()**

MmodGetResult

Synopsis Get the model finder result values.

Format `void MmodGetResult(ResultId, ResultIndex, ResultType, ResultArrayPtr)`

MIL_ID ResultId;	Model finder result buffer identifier
long ResultIndex;	Index of the result to return
long ResultType;	Type of result
void* ResultArrayPtr;	Array for results

Description This function obtains the result of the specified type from the specified result buffer, after an **MmodFind()** call.

Positional results are automatically returned in the output coordinate system specified in the associated calibration object of the target image. To change the output coordinate system, use **McalControl()** with `M_OUTPUT_COORDINATE_SYSTEM` set to `M_PIXEL` or `M_WORLD` to return all results in pixel units or world units, respectively. You should ensure that model source and target images are calibrated using the same world. If the target image is not calibrated, positional results are returned in pixels, relative to the top-left pixel in the target image.

The **ResultId** parameter specifies the identifier of the result buffer that was used to store results obtained by **MmodFind()**.

The **ResultIndex** parameter specifies the index of the result to get and can be set from 0 to `M_NUMBER - 1`, `M_ALL`, or `M_DEFAULT`.

The **ResultType** parameter specifies the type of result to return for each model occurrence. The result entries will be ordered by match score, regardless of the result type requested. Depending on the model finder context and the model used, some results might not be available. To test if a result is available, add `M_SUPPORTED` to the **ResultType**. The return value will be non-zero if the result is available.

The **ResultType** parameter can be set to one of the following values:

ResultType	Description
All the following result types return a single result (global):	
M_NUMBER	Number of occurrences found (all models).
M_CONTEXT_ID	Identifier of the context used for MmodFind() . Note that this identifier may not be valid if the model finder context has been freed using MmodFree() .
All the following result types return an array of results (one result per occurrence). The result array must be large enough to receive the requested data. Use M_NUMBER to determine the required array size.	
M_SCORE	Score of the occurrence (as a percentage). The score is calculated as follows: $\text{Score} = \text{Model coverage} \times (1 - \text{Fit error weighting factor} \times \text{Normalized fit error})$ Note: the normalized fit error is the fit error converted to a number between 0.0-1.0.
M_SCORE_TARGET	Target score of the occurrence. The target score is calculated as follows: $\text{Target Score} = \text{Target coverage} \times (1 - \text{Fit error weighting factor} \times \text{Normalized fit error})$ Note: the normalized fit error is the fit error converted to a number between 0.0-1.0.
M_MODEL_COVERAGE	Model coverage of the occurrence. The model coverage is the percentage of the total length of the model's active edges found in the occurrence. 100% indicates that for every edge in the model, a corresponding edge was found in the occurrence.
M_TARGET_COVERAGE	Target coverage of the model. The target coverage is the total length of the model's active edges found in the occurrence, divided by the total length of edges present in the occurrence bounding box, given as a percentage. Edges found in the target that are not present in the model will reduce this score.
M_FIT_ERROR	The fit error is the average quadratic distance, in pixels or calibrated units, between the edges in the occurrence and the corresponding edges in the model. A perfect fit is represented as 0.0.
M_POSITION_X	X-coordinate of the occurrence. This is the position of the reference axis for the occurrence, specified for the model using MmodControl() with M_REFERENCE_X.
M_POSITION_Y	Y-coordinate of the occurrence. This is the position of the reference axis for the occurrence, specified for the model using MmodControl() with M_REFERENCE_Y.

ResultType	Description
M_ANGLE	Angle of the occurrence, relative the reference axis angle specified for the model, using MmodControl() with M_REFERENCE_ANGLE.
M_SCALE	Scale of the occurrence.
M_POLARITY	Polarity of the occurrence.
*M_A_FORWARD	Forward transformation coefficient A, for the occurrence.
*M_B_FORWARD	Forward transformation coefficient B, for the occurrence.
*M_C_FORWARD	Forward transformation coefficient C, for the occurrence.
*M_D_FORWARD	Forward transformation coefficient D, for the occurrence.
*M_A_REVERSE	Reverse transformation coefficient A, for the occurrence.
*M_B_REVERSE	Reverse transformation coefficient B, for the occurrence.
*M_C_REVERSE	Reverse transformation coefficient C, for the occurrence.
*M_D_REVERSE	Reverse transformation coefficient D, for the occurrence.
<p>* The transformation coefficients allow you to map any point in the model region extracted from the model source image to the corresponding point in the occurrence in the target image (forward) or from the occurrence in the target to the corresponding point in the model region extracted from the model source image (reverse) with the following equations:</p> $x_d = ax_s + by_s + c$ $y_d = -bx_s + ay_s + d$ <p>where:</p> <p>a, b, c, d: Transformation coefficients (forward or reverse).</p> <p>x_s, y_s : Source coordinate (relative to the origin of the model for forward transformation or target image for reverse transformation).</p> <p>x_d, y_d : Destination coordinate (relative to the origin of the target image for forward transformation or model for reverse transformation).</p>	
M_INDEX	Index of the found model in the context.
M_USER_LABEL	User label of the found model in the context.

The **ResultArrayPtr** parameter specifies the pointer to the array into which to write results.

You can add M_TYPE_LONG or M_TYPE_DOUBLE to the **ResultType** to specify the data type of the user array pointed to by the **ResultArrayPtr** parameter. The default is M_TYPE_DOUBLE.

Example MmodFind.c

See also MmodAllocResult(), MmodFind(), MmodInquire()

MmodInquire

Synopsis Inquire information from a specified model finder context.

Format `long MmodInquire(ContextId, Index, InquireType, UserVarPtr)`

MIL_ID ContextId;	Identifier of model finder context
long Index;	Index of the model
long InquireType;	Setting to inquire about
void *UserVarPtr;	Storage location for inquired information

Description This function inquires information about the specified model finder context or the models contained therein.

The **ContextId** parameter specifies the model finder context about which to inquire information. The model finder context must have been previously allocated on the system using **MmodAlloc()**.

The **Index** parameter specifies the model finder context or individual model about which to inquire information. To inquire information from a model finder context, set the **Index** parameter to M_CONTEXT. To inquire information from a model, set the **Index** parameter to the index of the required model. M_DEFAULT refers to an index of 0. To retrieve the index of a model from its user label, use **MmodInquire()** with the user label as index, and with M_INDEX_FROM_LABEL as the inquire type.

The **InquireType** parameter specifies the parameter about which to inquire.

For M_CONTEXT or individual model inquiries, the **InquireType** value can be set to the following:

InquireType	Description
M_OBJECT_TYPE	The type of object currently being inquired (M_GEOMETRIC for a context or M_IMAGE for a model)

The possible settings to inquire for M_CONTEXT are:

InquireType	Description
M_ACCURACY	The search accuracy (M_MEDIUM or M_HIGH).
M_DETAIL_LEVEL	The level of details to extract from the model source and target image (M_MEDIUM, M_HIGH, or M_VERY_HIGH).
M_MODIFICATION_COUNT	The current value of the modification counter. The modification counter is increased by one each time settings for the context are modified. The modification counter is initialized to a number that is unique to the model finder context and is given its own unique range. If the context is freed, this number will not be reassigned to a new context. This feature can be useful when designing your application. For example, if the modification counter has changed, you can prompt the user to save before closing the application.
M_NUMBER	The maximum number of occurrences of all models to find in the target image.
M_NUMBER_MODELS	The number of models in the context.
M_PREPROCESSED	Whether the model finder context is preprocessed (0 = No). This inquire type will be set to false when certain settings of the model finder context are changed with MmodControl() , or if a model is added or removed with MmodDefine() .
M_SEARCH_ANGLE_RANGE	Whether forcing a search in an angular range is enabled (M_ENABLE or M_DISABLE).
M_SEARCH_SCALE_RANGE	Whether forcing a search in a scale range is enabled (M_ENABLE or M_DISABLE).
M_SHARED_EDGES	Whether multiple occurrences can share edges (M_ENABLE or M_DISABLE).
M_SMOOTHNESS	The degree of smoothness applied to model source and target images before edges are extracted (0.0.-100.0).
M_SPEED	The search speed (M_MEDIUM, M_HIGH, or M_VERY_HIGH).
M_TIMEOUT	The maximum search time for MmodFind() , in msec.

The possible settings to inquire for a model are:

InquireType	Description
M_ACCEPTANCE	The acceptance level for the score (0.0-100.0).
M_ACCEPTANCE_TARGET	The acceptance level for the target score (0.0-100.0).
M_ALLOC_OFFSET_X	The X-offset, in pixels, of the model in the model source image.
M_ALLOC_OFFSET_Y	The Y-offset, in pixels, of the model in the model source image.
M_ALLOC_SIZE_X	The width, in pixels, of the region that defined the model in the model source image.
M_ALLOC_SIZE_Y	The height, in pixels, of the region that defined the model in the model source image.
M_ANGLE	The nominal search angle value (0.0-360.0).
M_ANGLE_DELTA_NEG	The clockwise angle variation, relative to the nominal search angle (0.0-180.0).
M_ANGLE_DELTA_POS	The counter-clockwise angle variation, relative to the nominal search angle (0.0-180.0).
M_ASSOCIATED_CALIBRATION	The calibration object associated with the specified model.
M_CERTAINTY	The certainty level for the score (0.0-100.0).
M_CERTAINTY_TARGET	The certainty level for the target score (0.0-100).
M_CHAIN_INDEX	The indices which differentiate each active edge's chain of pixels within the model. The first chain is identified as index 1, with each subsequent chain's index incremented by 1. Use M_NUMBER_OF_CHAINED_PIXELS to determine the size of the array necessary to hold the chained pixel information.
M_CHAIN_X	The X-coordinates, with sub-pixel accuracy, of pixels in the chains composing the model's active edges. Coordinates are returned for all the active edge chains contained within the model and are relative to the model's origin in the model source image. Use M_NUMBER_OF_CHAINED_PIXELS to determine the size of the array necessary to hold the chained pixel coordinates.

InquireType	Description
M_CHAIN_Y	The Y-coordinates, in pixels (with sub-pixel accuracy), of each pixel in the chains composing the model's active edges. Coordinates are returned for all the active edge chains contained within the model and are relative to the model's origin in the model source image. Use M_NUMBER_OF_CHAINED_PIXELS to determine the size of the array necessary to hold the chained pixel coordinates.
M_FIT_ERROR_WEIGHTING_FACTOR	The contribution of the fit error in the score and target score calculation (0.0-100.0).
M_MIN_SEPARATION_ANGLE	The minimum angular separation required for two occurrences to be considered distinct matches. This value is an absolute angle value ($0.0 < x \leq 360.0$).
M_MIN_SEPARATION_SCALE	The minimum separation required in scale for two occurrences to be considered distinct matches. This value is a scale factor ($1.0 < x \leq 4.0$).
M_MIN_SEPARATION_X	The minimum separation required along the X-axis for two occurrences to be considered distinct matches. This value is a percentage of the model's height at M_SCALE ($0.0 < x \leq 100.0$).
M_MIN_SEPARATION_Y	The minimum separation required along the Y-axis for two occurrences to be considered distinct matches. This value is a percentage of the model's height at M_SCALE ($0.0 < x \leq 100.0$).
M_NUMBER	The maximum number of occurrences of the specified model to find in the target image.
M_NUMBER_OF_CHAINED_PIXELS	The total number of chained pixels in the active edges of the model.
M_ORIGINAL_X	The X-coordinate of the model's reference axis position, relative to the model source image's origin.
M_ORIGINAL_Y	The Y-coordinate of the model's reference axis position, relative to the model source image's origin.
M_POLARITY	The expected polarity of occurrences relative to the model (M_SAME, M_REVERSE, M_SAME_OR_REVERSE, or M_ANY).
M_REFERENCE_ANGLE	The reference axis angle of the model (0.0-360.0).
M_REFERENCE_X	The X-coordinate, in pixels, of the reference axis position of the model, relative to the model's origin in the model source image.

InquireType	Description
M_REFERENCE_Y	The Y-coordinate, in pixels, of the reference axis position of the model, relative to the model's origin in the model source image.
M_SCALE	The nominal search scale value (0.5-2.0).
M_SCALE_MAX_FACTOR	The maximum factor applied to the nominal search scale (1.0-2.0).
M_SCALE_MIN_FACTOR	The minimum scale factor applied to the nominal search scale (0.5-1.0).
M_SEARCH_OFFSET_X	The search area offset, in pixels, in the X-direction from the target image's origin.
M_SEARCH_OFFSET_Y	The search area offset, in pixels, in the Y-direction from the target image's origin.
M_SEARCH_SIZE_X	The search area width, in pixels.
M_SEARCH_SIZE_Y	The search area height, in pixels.
M_USER_LABEL	The user defined label.

For user label inquiries (the label is specified as the index), the **InquireType** value can be set to the following:

InquireType	Description
M_INDEX_FROM_LABEL	The model index associated with a model user label.

The **UserVarPtr** parameter specifies the address of the variable, or array, in which to write the requested information. Results are returned in the user variable, or array, as type "double". If you want results to be returned as type "long", combine the specified result type with M_TYPE_LONG (for example, M_ORIGINAL_REF_X+M_TYPE_LONG).

Return Value The return value is the required information, cast as a long. All position or size-related information is returned in pixels.

See also **MmodControl()**, **MmodDefine()**

MmodMask

Synopsis Mask regions of a model.

Format **void MmodMask(ContextId, Index, MaskBufferId, MaskType, ControlFlag)**

MIL_ID ContextId;	Identifier of the model finder search context
long Index;	Index of the model
MIL_ID MaskBufferId;	Mask buffer identifier
long MaskType;	Type of mask
long ControlFlag;	Control flag

Description This function allows you to apply a mask to regions of a model. The mask can either set pixels in the specified model to one of two states: a “don’t care” state or a “flat region” state.

“Don’t care” pixels in a model will not be considered when searching for occurrences of the model in the target image

“Flat region” pixels denote a region where no features are expected, other than consistent pixel values with no edges present; any edge found in a “flat region” reduces the score.

The **ContextId** parameter specifies the model finder context of the model to mask. The model finder context must have been previously allocated on the system using **MmodAlloc()**.

The **Index** parameter specifies the individual model for which to create a mask. M_DEFAULT refers to index 0. User labels cannot be used as indices; to retrieve the index of a model from its user label, use **MmodInquire()** with M_INDEX_FROM_LABEL.

The **MaskBufferId** parameter specifies the identifier of the image buffer used to identify the masked pixels in the model. This buffer must be a 1-band 8-bit unsigned buffer, the same size as the model. Note that a masked pixel corresponds to a non-zero value in the mask buffer. This image buffer can be calibrated, however, this is not necessary (even if the model source image is calibrated); the mask is applied on a pixel basis. Setting the **MaskBufferId** parameter to M_NULL is equivalent to passing a mask buffer cleared to 0.

The **MaskType** parameter determines which type of mask to employ and can be set to one of the following:

MaskType	Description
M_DONT_CARES	Features present in the masked region (non-zero pixels) are ignored. Features present in this region of the occurrence will not affect the scores.
M_FLAT_REGIONS	No features are expected in the masked region (non-zero pixels). If features are present in this region of the occurrence, the target score will decrease.

The **ControlFlag** parameter is reserved for future expansion and must be set to M_DEFAULT.

Example MmodFind.c

MmodPreprocess

Synopsis Preprocess a model finder context.

Format `void MmodPreprocess(ContextId, ControlFlag)`

MIL_ID ContextId;	Identifier of the model finder context to pre-process
long ControlFlag;	Control flag

Description This function preprocesses the specified model finder context. It extracts the active edges of the models contained within the model finder context and sets internal search settings so that future searches will be optimized for speed and robustness. Essentially, **MmodPreprocess()** trains the system to search for the model in the most efficient manner. The procedure is potentially quite lengthy (up to a few seconds) and increases the size of the model finder context significantly.

This function must be called before performing the first **MmodFind()** search. Call this function after all search settings have been set. When you save the model finder context, the model's preprocessing changes are not stored with the model. Upon restoration, the model must be preprocessed again.

Note that if some of the model's search settings are changed after a call to **MmodPreprocess()**, the model must be preprocessed again. To inquire if your model is in a preprocessed state, use **MmodInquire()** with `M_PREPROCESSED` (this value must be true to perform an **MmodFind()** search).

The **ContextId** parameter specifies the model finder context of the model to preprocess. The model finder context must have been previously allocated on the system using **MmodAlloc()**.

The **ControlFlag** parameter can be set to `M_RESET` to undo preprocessing. This can be useful if you want to conserve system memory within an application and preserve model finder context settings. Otherwise this parameter must be set to `M_DEFAULT`.

Example `MmodFind.c`

See also **MmodFind()**, **MmodRestore()**, **MmodSave()**

MmodRestore

Synopsis Restore a model finder context from disk.

Format MIL_ID MmodRestore(FileName, SystemId, ControlFlag, ContextIdPtr)

MIL_TEXT_PTR FileName;	Name of the input file to read
MIL_ID SystemId;	System identifier
long ControlFlag;	Control flag
MIL_ID *ContextIdPtr;	Storage location of the model finder context identifier

Description This function restores a model finder context that was previously saved to a file, using **MmodSave()**. This function restores all of the model finder context's settings that were in effect when the model finder context was saved. A restored model finder context is not preprocessed, therefore you must call **MmodPreprocess()** before performing a search with **MmodFind()**. If you had associated a calibration object with the model(s) in your model finder context, you must reassociate the calibration object, using **MmodControl()** with M_ASSOCIATED_CALIBRATION.

The **FileName** parameter specifies the model file name. The function handles (internally) the opening and closing of the file.

The **SystemId** parameter specifies the system on which to restore the model finder context. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to restore the model finder context (the Host system or any already allocated system).

The **ControlFlag** parameter is reserved for future expansion and must be set to M_DEFAULT.

The **ContextIdPtr** parameter specifies the address of the variable in which to write the model finder context identifier. Since the **MmodRestore()** function also returns the model finder context identifier, you can set this parameter to M_NULL. If allocation fails, M_NULL is written as the identifier.

Return value The returned value is the model finder context identifier. If allocation fails, M_NULL is returned.

See also **MmodControl(), MmodFind(), MmodRestore(), MmodSave()**

MmodSave

Synopsis Save a model finder context to a file.

Format **void MmodSave(FileName, ContextId, ControlFlag)**

MIL_TEXT_PTR FileName;	Name of the output file
MIL_ID ContextId;	Identifier of model finder context
long ControlFlag;	Control flag

Description This function saves all the information about the previously allocated model finder context to disk, including all of the individual model's current settings. This information can be reloaded, using **MmodRestore()**. However, preprocessing and any associated calibration objects are not saved.

The **FileName** parameter specifies the file name of the model finder context; it is recommended that you use the *.mmf* file extension for easier use with other Matrox Imaging software products. If this file already exists, it will be overwritten. The function handles (internally) the opening and closing of this file.

The **ContextId** parameter specifies the identifier of the model finder context to save.

The **ControlFlag** parameter is reserved for future expansion and must be set to M_DEFAULT.

See also **MmodRestore()**

MocrAllocFont

Synopsis Allocate an OCR font buffer.

Format MIL_ID MocrAllocFont(SystemId, FontType, CharNumber, CharBoxSizeX, CharBoxSizeY, CharOffsetX, CharOffsetY, CharSizeX, CharSizeY, CharThickness, StringLength, InitFlag, FontIdPtr)

MIL_ID SystemId;	System identifier
long FontType;	Font type
long CharNumber;	Number of characters in font
long CharBoxSizeX;	Width of a character's box
long CharBoxSizeY;	Height of a character's box
long CharOffsetX;	X offset of a character in its box
long CharOffsetY;	Y offset of a character in its box
long CharSizeX;	Width of a character
long CharSizeY;	Height of a character
long CharThickness;	Thickness of a character
long StringLength;	Length of string to read/verify
long InitFlag;	Initialization flag
MIL_ID *FontIdPtr;	Storage location for font identifier

Description This function allocates a new, empty font that can be used to create a custom font. After allocation the font buffer must have its grayscale character representations initialized using the **MocrImportFont()** or **MocrCopyFont()** functions. If the default processing controls and character constraints associated with the newly created font are not satisfactory, they can be changed using **MocrControl()** and **MocrSetConstraint()**.

The main defaults for a newly created font are:

- All font characters can appear at any position in the string.
- No checksum is activated.
- Target character size (X and Y) is set to the font character size.
- Target character spacing is equal to the target character size X.

Refer to **MocrControl()** for other default values.

Note that for **FontTypes** M_SEMI_M12_92 and M_SEMI_M13_88, the character constraints have been set according to the SEMI font standards and a checksum calculation is activated. The target character size and spacing are the same as above.

When a font is no longer required, it should be freed with **MocrFree()**.

The **SystemId** parameter specifies the system on which to allocate the font buffer. This parameter must be given a valid system identifier.

Alternatively, the parameter can be set to M_DEFAULT to have MIL automatically select the most appropriate system on which to allocate the buffer (either on the default Host system or on another system already allocated). If this parameter is set to M_DEFAULT_HOST, the default Host system of the current MIL application will be chosen for the allocation.

The **FontType** parameter specifies the type of font being defined. Possible values for this parameter are:

M_DEFAULT	A general user-defined font.
M_SEMI_M12_92	A font respecting the SEMI M12-92 standard.
M_SEMI_M13_88	A font respecting the SEMI M13-88 standard.

The **CharNumber** parameter determines how many characters can be stored in the font.

The **CharBoxSizeX** and **CharBoxSizeY** parameters specify the font character's box width and height.

The **CharOffsetX** and **CharOffsetY** parameters specify the X and Y offsets of a character in its character box.

The **CharSizeX** and **CharSizeY** parameters specify the font character width and height.

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

$$2 * \text{CharOffsetX} + \text{CharSizeX} \leq \text{CharBoxSizeX}$$

$$2 * \text{CharOffsetY} + \text{CharSizeY} \leq \text{CharBoxSizeY}$$

The **CharThickness** parameter specifies the maximum thickness (stroke width) of the font character.

The **StringLength** parameter specifies the maximum length of the string that will be read/verified using the newly allocated font.

The **InitFlag** parameter is an initialization flag and should be set to either:

M_FOREGROUND_WHITE	The characters to read/verify are brighter than the background.
M_FOREGROUND_BLACK	The characters to read/verify are darker than the background.

The **FontIdPtr** parameter specifies the address of the variable to which the font buffer identifier is to be written. If allocation fails, M_NULL is written as the identifier. Since the **MocrAllocFont()** function also returns the buffer identifier, you can set this parameter to M_NULL.

Return value This function returns the new font identifier. If allocation fails, M_NULL is returned as the identifier.

See also **MocrCopyFont()**, **MocrImportFont()**, **MocrFree()**, **MocrControl()**, **MocrSetConstraint()**

MocrAllocResult

Synopsis Allocate an OCR result buffer.

Format **MIL_ID MocrAllocResult(SystemId, InitFlag, OcrResultIdPtr)**

MIL_ID SystemId;	System identifier
long InitFlag;	Initialization flag
MIL_ID *OcrResultIdPtr;	Storage location for OCR result buffer identifier

Description This function allocates a result buffer and returns an identifier to it for use with other OCR functions. When the result buffer is no longer required, you should release it using the **MocrFree()** function.

The **SystemId** parameter specifies the system on which to allocate the OCR result buffer. This parameter must be given a valid system identifier. Alternatively, the parameter can be set to M_DEFAULT to have MIL automatically select the most appropriate system on which to allocate the buffer (either on the default Host system or on another system already allocated). If this parameter is set to M_DEFAULT_HOST, the default Host system of the current MIL application will be chosen for the allocation.

The **InitFlag** parameter is an initialization flag and should be set to M_DEFAULT.

The **OcrResultIdPtr** parameter specifies the address of the variable to which the OCR result buffer identifier is to be written. If allocation fails, M_NULL is returned as the identifier. Since the **MocrAllocResult()** function also returns the OCR result buffer identifier, you can set this parameter to M_NULL.

Return value This function returns the OCR result buffer identifier. If allocation fails, M_NULL is returned as the identifier.

See also **MocrFree(), MocrGetResult()**

MocrCalibrateFont

Synopsis Calibrate font character size to match a sample image.

Format `void MocrCalibrateFont(ImageBufId, FontId, String,
TargetCharSizeXMin, TargetCharSizeXMax,
TargetCharSizeXStep, TargetCharSizeYMin,
TargetCharSizeYMax, TargetCharSizeYStep, Operation)`

MIL_ID ImageBufId;	Sample calibration image identifier
MIL_ID FontId;	Font buffer identifier
MIL_TEXT_PTR String;	Character string in sample image
double TargetCharSizeXMin;	Minimum width of the target character
double TargetCharSizeXMax;	Maximum width of the target character
double TargetCharSizeXStep;	Width increment
double TargetCharSizeYMin;	Minimum height of the target character
double TargetCharSizeYMax;	Maximum height of the target character
double TargetCharSizeYStep;	Height increment
long Operation;	Operation flag

Description This function automatically calibrates the X and Y size and spacing of the font characters to match that of the string in the sample calibration image. The sample image must contain the string specified in the **String** parameter, and its size and spacing must be representative of the images to be read or verified with the calibrated font. The cleaner the sample image, the better the calibration results. If a string cannot be located in the image, an error message will be issued.

MocrCalibrateFont() should be called with new fonts or when the character size in the target images changes. This function, which can take several seconds to execute, determines the size and spacing of the characters in the target image.

Note that the target character size and spacing can be set manually by calling **MocrControl()** with the M_TARGET_CHAR_SIZE_X, M_TARGET_CHAR_SIZE_Y, and M_TARGET_CHAR_SPACING parameters.

The **ImageBufId** parameter specifies the buffer identifier of the sample calibration image. The characters of the string in this sample image must be of the same type and size as that in the images to be read/verified using the calibrated font.

The **FontId** parameter specifies the buffer identifier of the font to be calibrated.

The **String** parameter specifies the character string present in the sample calibration image. This string must be null-terminated.

The **TargetCharSizeXMin**, **TargetCharSizeXMax**, **TargetCharSizeYMin** and **TargetCharSizeYMax** together determine the range in which the character size may vary in the sample image. The increments specified by **TargetCharSizeXStep** and **TargetCharSizeYStep** specify the precision of the calibration.

Note that if the range is too wide or the step too low, the function may take up to several minutes to execute. In general, a range of +/- 1 pixel with a step of 0.125 pixel provides good results in a reasonable amount of time.

The **Operation** parameter specifies the operation to be performed. It should be set to M_DEFAULT.

Example `mocrread.c`

See also `MocrControl()`, `MocrReadString()`, `MocrVerifyString()`

MocrControl

Synopsis Set an OCR processing control.

Format `void MocrControl(FontId, ControlType, ControlValue)`

MIL_ID FontId;	Font buffer identifier
long ControlType;	Type of control to set
double ControlValue;	Value of control parameter

Description This function sets various OCR controls for the read/verify operation, such as inter-character spacing, etc.

The **FontId** parameter specifies the identifier of the font with which to associate the control settings.

The **ControlType** and **ControlValue** parameters together specify the type and value of control to set.

ControlType	Description	ControlValue
M_STRING_ACCEPTANCE	Minimum acceptance score for successful read/verification of string (default is 1 %).	0 - 100%
M_CHAR_ACCEPTANCE	Minimum acceptance score for successful read/verification of an individual character (default is 1 %).	0 - 100%
M_CHAR_INVALID	Symbol for unrecognized characters (set to M_NULL when no special character is desired). Default is M_NULL.	0 - 255
M_STRING_LENGTH	Specifies the string length to read. Must be less than the string length passed at the allocation time.	
M_TARGET_CHAR_SPACING	Specifies inter-character spacing (also set by calibration with MocrCalibrateFont()).	> 1
M_TARGET_CHAR_SIZE_X	Specifies target character X size (also set by calibration with MocrCalibrateFont()).	> 1
M_TARGET_CHAR_SIZE_Y	Specifies target character Y size (also set by calibration with MocrCalibrateFont()).	> 1
M_CHAR_ERASE	Specifies the character to erase from the font.	Any character present in the font.

ControlType	Description	ControlValue
M_SKIP_CONTRAST_ENHANCE	Skip contrast enhancement step (default is M_DISABLE).	M_ENABLE or M_DISABLE
M_SKIP_STRING_LOCATION	Skip string location step (default is M_DISABLE).	M_ENABLE or M_DISABLE
M_SPEED	Specifies the robustness/speed factor (default is M_MEDIUM).	M_VERY_HIGH: -very fast but least reliable M_HIGH: -fast for good images M_MEDIUM: -default setting M_LOW: -reliable for noisy images M_VERY_LOW: -most reliable but slow

See also **MocrSetConstraint()**

MocrCopyFont

Synopsis Copy a font character to or from an image buffer.

Format `void MocrCopyFont(ImageBufId, FontId, Operation, CharListString)`

MIL_ID ImageBufId;	Image buffer identifier
MIL_ID FontId;	Font buffer identifier
long Operation;	Operation flag
MIL_TEXT_PTR CharListString;	String containing the list of characters to copy

Description This function copies a grayscale representation of one, or many, font characters to/from the specified image buffer. It can be used to initialize a font, to change a font, or to obtain a visual representation of the font's characters.

The **ImageBufId** parameter specifies the identifier of the image buffer to/from which characters are copied. If the image buffer is being copied to the font, the font must be large enough to hold the representations of all the specified characters. You can use **MocrInquire()** to determine the number of characters in the font and the size of each character.

The **FontId** parameter specifies the identifier of the font to/from which the characters are copied.

The **Operation** parameter specifies the direction of the copy operation. It can take one of the following values:

M_COPY_TO_FONT	Copy character(s) from an image buffer to a font buffer.
M_COPY_FROM_FONT	Copy character(s) from a font buffer to an image buffer.

If M_ALL_CHAR is added to M_COPY_TO_FONT or to M_COPY_FROM_FONT, the **CharListString** parameter is ignored (it should be set to M_NULL) and all font characters are copied to/from the image buffer, stacked from left to right and from top to bottom.

The **CharListString** parameter specifies a string containing the list of characters to be copied. This string must be null-terminated.

If `M_COPY_TO_FONT` is specified and a character exists in both the list and the font, the font character grayscale representation will be overwritten with the data of the character from the list. New characters will be added to the font provided that there are sufficient free entries. Note that it is crucial that the character representation respects the foreground value as set in **MocrAllocFont()**. If the foreground value of the font does not match, the font will be unusable.

If `M_COPY_FROM_FONT` is specified, **CharListString** must point to the list of characters to be copied from the font. The grayscale representation of font characters specified in this list will be copied to the specified image buffer.

The **CharListString** parameter must be set to `M_NULL` if `M_ALL_CHAR` is added to the **Operation** parameter.

See also **MocrAllocFont()**, **MocrSaveFont()**, **MocrInquire()**, **MocrImportFont()**, **MocrModifyFont()**

MocrFree

Synopsis Free an OCR font or result buffer.

Format **void MocrFree(FontOrResultId)**

MIL_ID FontOrResultId;	OCR font or result buffer identifier
------------------------	--------------------------------------

Description This function deletes the specified OCR font or result buffer identifier, and releases any memory associated with it.

The **FontOrResultId** parameter specifies the identifier of the OCR font or result buffer to free. The buffer must have been successfully allocated with **MocrAllocFont()** or **MocrAllocResult()** prior to calling this function.

See also **MocrAllocFont(), MocrAllocResult()**

MocrGetResult

Synopsis Read results from an OCR result buffer.

Format **void MocrGetResult(OcrResultId, ResultToGet, ResultPtr)**

MIL_ID OcrResultId;	OCR result buffer identifier
long ResultToGet;	Result type
void *ResultPtr;	Storage location for OCR result buffer

Description This function reads the specified results of a read or verify operation from an OCR result buffer.

The **OcrResultId** parameter specifies the identifier of the OCR result buffer to be read. This buffer typically contains results obtained by **MocrReadString()** or **MocrVerifyString()**.

The **ResultToGet** parameter specifies the type of result to read from the result buffer. It can be set to one of the following values:

M_STRING_VALID_FLAG	Description Read the flag that denotes the validity of the entire string. This flag is set depending on the user-specified acceptance score. To set the acceptance score value above which a string will be considered valid, call MocrControl() with the M_STRING_ACCEPTANCE parameter. By default, all strings with a score greater or equal to 1 are valid.
	Returned values M_TRUE or M_FALSE.
	ResultPtr Type Pointer to a double.

M_STRING	<p>Description Read the null-terminated string found during the read/verify operation. By default, any unrecognized character in the string will be replaced by its most-likely value, even if the confidence score for that value is lower than the user-specified character acceptance threshold (M_CHAR_ACCEPTANCE). If desired, the user can replace all unrecognized characters in the string with an invalid character marker by setting M_CHAR_INVALID to the appropriate value using MocrControl().</p> <p>Returned Values Null-terminated string of characters.</p> <p>ResultPtr Type Pointer to an array of MIL_TEXT_CHAR.</p>
M_STRING_SCORE	<p>Description Read the confidence score for the entire string calculated during the read/verify operation.</p> <p>Returned Values Returned values range numerically from 0 to 100%. The string score will probably never exactly reach 100%, and you should experiment to determine the range of scores corresponding to the desired level of confidence.</p> <p>ResultPtr Type Pointer to a double.</p>
M_CHAR_VALID_FLAG	<p>Description Read an array of flags listing the validity of individual string characters. Each flag is set depending on the user-specified acceptance score. To set the acceptance score value above which a character will be considered valid, call MocrControl() with the M_CHAR_ACCEPTANCE parameter. By default, all characters with a score greater or equal to 1 are valid.</p> <p>Returned Values M_TRUE or M_FALSE.</p> <p>ResultPtr Type Pointer to an array of double.</p>

M_CHAR_SCORE	Description Read an array of confidence scores for each individual string character.
	Returned Values Returned values range from 0% to 100%. The string score will probably never exactly reach 100%, and you should experiment to determine the range of scores corresponding to the desired level of confidence.
	ResultPtr Type Pointer to an array of double.
M_CHAR_POSITION_X	Description Read an array containing the X positions of each individual character.
	Returned Values 0 to X size of image.
	ResultPtr Type Pointer to an array of double.
M_CHAR_POSITION_Y	Description Read an array containing the Y positions of each individual character.
	Returned Values 0 to Y size of image.
	ResultPtr Type Pointer to an array of double.

To change the type of a returned value, the user can append one of the following suffixes to the **ResultToGet** value:

M_TYPE_CHAR	Returns data cast to char.
M_TYPE_SHORT	Returns data cast to short.
M_TYPE_LONG	Returns data cast to long.
M_TYPE_FLOAT	Returns data cast to float.
M_TYPE_DOUBLE	Returns data cast to double.

As an example: M_CHAR_POSITION_X is normally returned as an array of type double, but if you set **ResultToGet** to M_CHAR_POSITION_X + M_TYPE_LONG, the X coordinates of all string characters will be written to the array pointed to by **ResultPtr** as long values.

The **ResultPtr** parameter specifies the address of the location to which the results will be written. The type and size of the variable or array pointed to by **ResultPtr** must match the default result pointer type and size specified in the table above (unless the default has been overridden). If the variable is an array, its size must be equal to the number of characters read or verified, except for the M_STRING parameter which must have one extra entry since the string is null-terminated.

Example mocrread.c

See also MocrReadString(), MocrVerifyString(), MocrInquire(), MocrControl()

MocrHookFunction

Synopsis Hook a function to an event.

Format **MOCRHOOKFCTPTR MocrHookFunction(FontId, HookType, HookHandlerPtr, UserDataPtr)**

MIL_ID FontId;	Font identifier
long HookType;	Type of event to hook
MOCRHOOKFCTPT HookHandlerPtr;	Address of function to call when an event occurs
void *UserDataPtr;	User data pointer or M_NULL

Description This function allows you to attach or detach a user defined function to an event when the specified font is used. A type of event to which a user defined function can be hooked is the string validation during a read or verify operation. When this type of event is hooked, the **MocrReadString()** or **MocrVerifyString()** function will call the hooked function one or many times during the operation to validate the string that was read before writing it in the OCR result buffer. This function allows you 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.

Note that functions hooked to an event execute on a distinct thread. This permits the functions to run asynchronously from the operation that fired the event and from functions hooked to other events. Although there is a small queue to permit a certain amount of overlap, hooked functions should not take longer to execute than the period in which two of their associated events can occur. You cannot determine the instance of the event that fired the function, and even if this were possible, this information would generally not be very useful. Typically, a hooked function performs the minimum number of operations required and, if necessary, performs longer processes by launching other threads.

The **FontId** parameter specifies the identifier of the font.

The **HookType** parameter specifies the event type. This parameter can be set to the following. Note that a hooked function must be unhooked by combining the **HookType** parameter with M_UNHOOK.

M_STRING_VALIDATION	Specifies to hook a function that will be called to validate the read strings.
---------------------	--

The **HookHandlerPtr** parameter specifies the address of the function that should be called when an event occurs.

The value returned by the hook function must be a long containing the string validity status: either M_TRUE or M_FALSE. Note that MOCRHOOKFCTPTR, MFTYPE, and MPTYPE are reserved MIL predefine types for function and data pointers.

The hook handler function, pointed to by **HookHandlerPtr**, must be declared as follows:

long MFTYPE HookHandler(HookType, StringPtr, UserDataPtr);	
long HookType;	Type of event hooked
void MPTYPE *StringPtr;	Pointer to string to validate
void MPTYPE *UserDataPtr;	User data pointer passed to MocrHookFunction()

The **UserDataPtr** parameter specifies a pointer to a user defined data structure. This pointer will be passed to the hook handler function when the event occurs. If not used, this parameter should be set to M_NULL.

Return value A pointer to the previously hooked function of the same hook type is returned. M_NULL is returned if no previous function was hooked. This allows you to chain hook functions and to restore the old hook functions when unhooking.

See also **MocrModifyFont()**, **MocrSetConstraint()**

MocrImportFont

Synopsis Import font data from file on disk.

Format **void MocrImportFont(FileName, FileFormat, Operation, CharListString, FontId)**

MIL_TEXT_PTR FileName;	Name of font data source file
long FileFormat;	File format
long Operation;	Operation type
MIL_TEXT_PTR CharListString;	String containing list of characters
MIL_ID FontId;	Font identifier

Description This function imports font character representations from a file to initialize or overwrite an existing font. This function's main use is to initialize custom fonts.

The **FileName** parameter specifies the name of the file from which to retrieve the font character representation information.

The **FileFormat** parameter specifies how the data is stored in the file. Possible formats are:

M_MIL	MIL format image file.
M_TIFF	TIFF format image file.
M_FONT_ASCII	User-drawn ASCII file.

When either M_MIL or M_TIFF is selected as the file format, the characters in the string specified by **CharListString** are read from the image file. The font character data is read from this file assuming that the source image is a grid of character representations matching the specified font size. The character representations are read from left to right and top to bottom, and the number of characters in the source image must equal the number of characters in the list pointed to by **CharListString**.

Note that only one-band images can be used.

When `M_FONT_ASCII` is selected as **FileFormat**, font character data is to be presented as follows:

File representation								Comments
MIL_ASCII_FONT<CR>								Specifies ASCII file format
<CR>								Optional end-of-line(s)
CharValue 43<CR>								Identifier of start of first character, followed by a space, followed by the value (generally ASCII) associated with that character data
00	00	00	00	00	00	00	<CR>	Data of character matching box size X
00	00	00	FF	00	00	00	<CR>	Data of character matching box size X
00	00	00	FF	00	00	00	<CR>	Data of character matching box size X
00	FF	FF	FF	FF	FF	00	<CR>	Data of character matching box size X
00	00	00	FF	00	00	00	<CR>	Data of character matching box size X
00	00	00	FF	00	00	00	<CR>	Data of character matching box size X
00	00	00	00	00	00	00	<CR>	Data of character matching box size X
<CR>								Optional end-of-line(s)
CharValue 44<CR>								Identifier of start of next character, followed by a space, followed by the value (generally ASCII) associated with that character data
etc.								

For an example of a user-drawn ASCII file see the *semi.txt* file in the `\MIL\EXAMPLES` directory.

The **Operation** parameter indicates the type of import operation to be performed. It should be set to `M_LOAD_CHARACTER`.

The **CharListString** parameter specifies a null-terminated string giving the list of characters to be read from the font file. The number of characters in this list must match the number of character representations present in the source image. For the `M_FONT_ASCII` **FileFormat**, this parameter is ignored and must be set to `M_NULL` since the codes associated with each character representation are already included in the file.

The **FontId** parameter specifies the font identifier to which character representations will be written.

See also `MocrAllocFont()`, `MocrCopyFont()`, `MocrRestoreFont()`, `MocrSaveFont()`

MocrInquire

Synopsis Inquire about font character information.

Format **void MocrInquire(FontId, InquireType, UserVarPtr)**

MIL_ID FontId;	Font identifier
long InquireType;	Item to inquire about
void *UserVarPtr;	Pointer to variable for returned value

Description This function inquires about a specific item of an OCR's font character information.

The **FontId** parameter specifies the identifier of the font from which to read information.

The **InquireType** parameter specifies the particular item of the font's character information to inquire about. It can be set to one of the following values:

InquireType	Description
M_OWNER_SYSTEM	The MIL identifier (MIL_ID) of the system on which the font has been allocated (MocrAllocFont()). This inquire type must be combined with M_TYPE_MIL_ID.
M_FONT_TYPE	Font type (MocrAllocFont()).
M_CHAR_NUMBER	The number of characters that can be stored in the font (MocrAllocFont()).
M_CHAR_NUMBER_IN_FONT	The number of characters that are currently stored in the font.
M_CHAR_BOX_SIZE_X	The font character box X size (MocrAllocFont()).
M_CHAR_BOX_SIZE_Y	The font character box Y size (MocrAllocFont()).
M_CHAR_OFFSET_X	The font character X offset (MocrAllocFont()).
M_CHAR_OFFSET_Y	The font character Y offset (MocrAllocFont()).
M_CHAR_SIZE_X	Font character X size (MocrAllocFont()).
M_CHAR_SIZE_Y	Font character Y size (MocrAllocFont()).
M_CHAR_THICKNESS	The thickness of the font character (MocrAllocFont()).

InquireType	Description
M_STRING_LENGTH	The length of the string that the font is trained to read/verify (MocrAllocFont()).
M_STRING_LENGTH_MAX	The maximum string length that will be read/verified using the allocated font (MocrAllocFont()).
M_FOREGROUND_VALUE	The foreground value (MocrAllocFont()).
M_FONT_INITFLAG	The initialization flag passed at font allocation (MocrAllocFont() and MocrModifyFont()).
M_STRING_ACCEPTANCE	The minimum score for a read/verified string to be considered valid (MocrControl()).
M_CHAR_ACCEPTANCE	The minimum score for a read/verified character to be considered valid (MocrControl()).
M_CHAR_INVALID	The symbol used for unrecognized characters. When no special character has been defined by the user the returned value is M_NULL (MocrControl()).
M_TARGET_CHAR_SPACING	Intercharacter spacing, which is the distance between adjacent characters (MocrCalibrateFont() and MocrControl()).
M_TARGET_CHAR_SIZE_X	The target image character X size (MocrCalibrateFont() and MocrControl()).
M_TARGET_CHAR_SIZE_Y	The target image character Y size (MocrCalibrateFont() and MocrControl()).
M_SKIP_CONTRAST_ENHANCE	The contrast enhancement flag (MocrControl()).
M_SKIP_STRING_LOCATION	The string location flag (MocrControl()).
M_SPEED	The robustness/speed factor.
M_CHAR_IN_FONT	The characters present in the font. A null terminated array of character is returned.
M_CONSTRAINT_TYPE+n	The constraint character type for the nth position (MocrSetConstraint()).
M_CONSTRAINT+n	The constraint string for the nth position in the string. A null terminated array of characters is returned (MocrSetConstraint()).

The **UserVarPtr** parameter specifies the address in which results will be written. This parameter should be a pointer to a double, except in the case of the M_CONSTRAINT+n and M_CHAR_IN_FONT parameters, when it should be a pointer to an array of MIL_TEXT_CHAR. The type and size of the data pointed to by **UserVarPtr** should match the default result pointer

type and size specified above (if not overridden). For the M_CONSTRAINT+n and M_CHAR_IN_FONT parameters, **UserVarPtr** must point to an array of size equal to the number of characters in the font plus one extra entry. This is because the string is null-terminated.

It is possible to override the default **UserVarPtr** type by appending one of the following to the **InquireType** parameter:

M_TYPE_CHAR	Returns data cast to a char.
M_TYPE_SHORT	Returns data cast to a short.
M_TYPE_LONG	Returns data cast to a long.
M_TYPE_FLOAT	Returns data cast to a float.
M_TYPE_DOUBLE	Returns data cast to a double.
M_TYPE_MIL_ID	Returns data cast to a MIL_ID.

For example, setting **InquireType** equal to M_CHAR_SCALE_X +M_TYPE_LONG will force the font character X size to be written to the variable pointed by **UserVarPtr** as a long value.

See also **MocrAllocFont(), MocrControl(), MocrSetConstraint(), MocrCalibrateFont(), MocrModifyFont()**

MocrModifyFont

Synopsis Invert a font to match the target image characters.

Format MIL_ID MocrModifyFont(FontId, Operation, ControlValue)

MIL_ID FontId;	Font identifier
long Operation;	Operation
long ControlValue;	Control value

Description This function inverts an existing font. When inverting, this function changes the foreground value (see **MocrAllocFont()**) to match that of the target image characters.

The **FontId** parameter specifies the buffer identifier of the font to invert.

The **Operation** parameter specifies the type of operation to perform. It can only be set to M_INVERT.

The **ControlValue** parameter specifies the control value and must be set to M_DEFAULT.

See also MocrAllocFont(), MocrCalibrateFont(), MocrControl()

MocrReadString

Synopsis Read an unknown string from an image.

Format `void MocrReadString(ImageBufId, FontId, OcrResultId)`

MIL_ID ImageBufId;	Image buffer identifier
MIL_ID FontId;	Font buffer identifier
MIL_ID OcrResultId;	OCR result buffer identifier

Description This function reads an unknown string from the specified image using the specified font. All existing font controls and constraints (which can be set with **MocrControl()** and **MocrSetConstraint()**) are taken into account, and results are stored in the specified result buffer. Results can be read from the result buffer by using the **MocrGetResult()** function.

This function assumes that the string to be read has the same length as specified in the font, and that the target string characters have the same type, size, and spacing as the characters in the sample image used for calibration (see **MocrCalibrateFont()**).

The **ImageBufId** parameter specifies the buffer identifier of the image which contains the string to be read.

The **FontId** parameter specifies the buffer identifier of the font to use to read the string in the target image.

The **OcrResultId** parameter specifies the result buffer in which to place results of the read operation.

Example mocrread.c

See also **MocrCalibrateFont()**, **MocrGetResult()**, **MocrSetConstraint()**, **MocrControl()**, **MocrVerifyString()**

MocrRestoreFont

Synopsis Restore a font from disk.

Format **MIL_ID MocrRestoreFont(FileName, Operation, SystemId, FontIdPtr)**

MIL_TEXT_PTR FileName;	Font file name
long Operation;	Operation flag
MIL_ID SystemId;	System identifier
MIL_ID *FontIdPtr;	Storage location for font identifier

Description This function restores an OCR font previously saved with **MocrSaveFont()** from a file on disk, and returns a handle to it.

The **FileName** parameter specifies the font file name.

The **Operation** parameter specifies which font data to restore. It can be set to one of the following values:

M_RESTORE	Allocates a new font using the SystemId specified, and restores all of the font's character, control, and constraint data. The new font ID is written to the variable specified by the FontIdPtr parameter.
M_LOAD_CONTROL	Loads only font control data into the specified font. The font ID is read from the variable specified by the FontIdPtr parameter.
M_LOAD_CONSTRAINT	Loads only font constraint data into the specified font. The font ID is read from the variable specified by the FontIdPtr parameter.
M_DEFAULT	Same as M_RESTORE.

The **SystemId** parameter determines the system on which the buffer will be allocated when an M_RESTORE or M_DEFAULT operation is specified. This parameter must be given a valid system identifier. Alternatively, **SystemId** can be set to M_DEFAULT to have MIL automatically select the most appropriate system on which to allocate the buffer (this system might be the default Host system or another already allocated system). This parameter can also be set to M_DEFAULT_HOST, in which case the default Host system of the current MIL application will be selected for the allocation.

The **FontIdPtr** parameter specifies the address of the variable to/from which the font buffer identifier is to be written/read.

After performing an M_RESTORE operation, the function returns the font identifier and writes it in the variable pointed to by **FontIdPtr**. Since **MocrRestoreFont()** also returns the font identifier, this parameter can be set to M_NULL. If the restore operation fails, M_NULL is returned as the identifier.

When performing an M_LOAD_CONTROL or an M_LOAD_CONSTRAINT operation, the function reads the target font identifier from the variable pointed to by **FontIdPtr**. It then loads the specified data into the existing font buffer.

Return value This function returns the buffer identifier of the existing or newly allocated font. If allocation fails, M_NULL is returned as the identifier.

Example mocrread.c

See also **MocrImportFont(), MocrSaveFont(), MocrCopyFont()**

MocrSaveFont

Synopsis Save an existing font to disk.

Format `void MocrSaveFont(FileName, Operation, FontId)`

MIL_TEXT_PTR FileName;	Font file name
long Operation;	Operation flag
MIL_ID FontId;	Font identifier

Description This function saves an existing font to disk using the MIL font file format. The font's control, constraint and/or character data can all be saved; which data is saved depends on the value of the **Operation** parameter.

The **FileName** parameter specifies the file to which font data will be saved. If the font file already exists, its entire contents will be overwritten.

The **Operation** parameter specifies which data to save to disk. The font character data contains the grayscale representation of each character in the font. The font control data includes such settings as target character size and spacing. The font constraint data specifies which characters can appear at given positions in the search string. The **Operation** parameter can take one of the following values:

M_SAVE	Save all font data.
M_SAVE_CONSTRAINT	Save only font character constraint data.
M_SAVE_CONTROL	Save only font control data.
M_DEFAULT	Same as M_SAVE.

The **FontId** parameter specifies the buffer identifier of the font to be saved.

Example mocrread.c

See also `MocrAllocFont()`, `MocrRestoreFont()`, `MocrImportFont()`, `MocrCopyFont()`, `MocrControl()`, `MocrSetConstraint()`

MocrSetConstraint

Synopsis Set character position constraints.

Format `void MocrSetConstraint(FontId, CharPos, CharPosType, CharValidString)`

MIL_ID FontId;	Font identifier
long CharPos;	Character position to set
long CharPosType;	Grammatical constraint parameter
MIL_TEXT_PTR CharValidString;	String containing list of valid characters

Description This function specifies the character value constraints to apply at each position of the string in the target image, when using the specified font. It specifies which characters can appear at given positions in the string to be read, and associates these constraints with the font specified in the **FontId** parameter. This function can be used to increase speed and reliability when a string has a known format and obeys certain grammatical rules.

The **FontId** parameter specifies the buffer identifier of the font with which to associate the constraints.

The **CharPos** parameter specifies the character position in the string for which a constraint is being set. Valid values range from zero to the length of the string minus 1. The **CharPosType** parameter, in conjunction with the **CharValidString** parameter, specifies the type of character which can appear at the specified string position. It can be set to one of the following values:

M_DEFAULT	All characters present in the font accepted.
M_DIGIT	Only characters "0...9" (ASCII codes 48 to 57) accepted.
M_LETTER	Only characters "A...Z", "a...z" (ASCII codes 65 to 90 and 97 to 122) accepted.
M_LETTER+M_UPPERCASE	Only characters "A...Z" (ASCII codes 65 to 90) accepted.
M_LETTER+M_LOWERCASE	Only characters "a...z" (ASCII codes 97 to 122) accepted.

The **CharValidString** parameter is an optional, null-terminated string which gives an explicit list of valid characters for the specified position. If this parameter is set to M_NULL, all font characters matching the

CharPosType specification will be programmed into the font as valid. If **CharValidString** is not set to `M_NULL`, all specified characters must match the **CharPosType** parameter definition and must exist in the font.

Example `mocrread.c`

See also `MocrControl()`

MocrVerifyString

Synopsis Verify a known string in an image.

Format `void MocrVerifyString(ImageBufId, FontId, String, OcrResultId)`

MIL_ID ImageBufId;	Image buffer identifier
MIL_ID FontId;	Font buffer identifier
MIL_TEXT_PTR String;	Character string to be verified
MIL_ID OcrResultId;	OCR result buffer identifier

Description This function verifies that a known string can be properly read using the specified font. It can be used to quickly determine whether a known string is present in the target image, and to evaluate its quality. This operation can be performed more quickly with **MocrVerifyString()** than with **MocrReadString()**. After verification, the specified result buffer contains a Pass/Fail flag and a confidence score. The user can obtain results from the result buffer with the **MocrGetResult()** function.

The **ImageBufId** parameter specifies the buffer identifier of the target image containing the string to be verified.

The **FontId** parameter specifies the buffer identifier of the font to use to verify the string in the target image.

The **String** parameter contains the character string to be verified.

The **OcrResultId** parameter identifies the OCR result buffer in which to place the verification results.

See also `MocrReadString()`, `MocrGetResult()`

MpatAllocAutoModel

Synopsis Automatically allocate unique pattern matching models of the specified type, from a source image.

Format `MIL_ID MpatAllocAutoModel(SystemId, SrcImageBufId, SizeX, SizeY, PosUncertaintyX, PosUncertaintyY, ModelType, Mode, ModelIdArrayPtr)`

<code>MIL_ID SystemId;</code>	System identifier
<code>MIL_ID SrcImageBufId;</code>	Identifier of the source image from which to extract the model
<code>long SizeX;</code>	Model width
<code>long SizeY;</code>	Model height
<code>long PosUncertaintyX;</code>	Maximum X-axis displacement
<code>long PosUncertaintyY;</code>	Maximum Y-axis displacement
<code>long ModelType;</code>	Model type
<code>long Mode;</code>	Mode of operation
<code>MIL_ID *ModelIdArrayPtr;</code>	Storage location for model identifier(s)

Description This function searches for the specified number of most-suitable unique areas, of the specified dimensions, in the model's source image. From each area found, the function automatically allocates a model. If none are found, no model is allocated and an error is reported. It can take several seconds to find the best models (more for large or small images).

To be effective, the model's target image should be a typical target image. Therefore, **MpatAllocAutoModel()** is useful, for example, when you want to perform whole image alignment, for which allocation of a unique model is essential.

You can determine the offset of a model's origin relative to its source image, using the **MpatInquire()** function.

If the eventual model can appear at an angle, in most cases, it is better to allocate an `M_NORMALIZED + M_CIRCULAR_OVERSCAN` type of model and then use **MpatSetAngle()** to specify the angular range.

You can change a model's search parameters at any time, using the appropriate **MpatSet...()** command. When the model(s) is no longer required, you should release its memory, using **MpatFree()**.

The **SystemId** parameter specifies the system on which to allocate the models. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to allocate the models (the Host system or any already allocated system).

The **SrcImageBufId** parameter specifies the identifier of the image buffer from which to extract the models. This image should be representative of the target images.

The **SizeX** and **SizeY** parameters define the width and height of the required models.

The **PosUncertaintyX** and **PosUncertaintyY** parameters specify the maximum displacement (shift) expected between the reference position of the models in the source image and their position when found in the target images. This information is used to select models that are far enough from the image borders to be present in the target images. Set **PosUncertaintyX** and **PosUncertaintyY** to the expected maximum pixel displacement in the vertical and horizontal direction, respectively. You can set either of these parameters to M_DEFAULT if the models can be selected from anywhere in the whole width and/or height of the image, respectively. The default search region of each model is automatically set using the original position of the model, plus or minus the specified positional uncertainty.

The **ModelType** parameter specifies the type of the model.

M_NORMALIZED	A model used to search for the position, match score, and angle (if angular search is enabled) of a model occurrence in the specified image, using MpatFindModel() and MpatFindMultipleModel() .
M_NORMALIZED+ M_ORIENTATION	A model that can be used to search for the position, match score, and angle (if angular search is enabled) of a model occurrence with MpatFindModel() . Alternatively, the model can be used to find the global orientation of the model in a target image with MpatFindOrientation() . The model's source image should be a typical target image.

You can add the following setting to the above settings:

M_CIRCULAR_OVERSCAN	Extracts the model, as well as circular overscan data from the source image. The overscan data is determined by rotating the area from which to extract the model, about its center. The overscan data of an M_CIRCULAR_OVERSCAN type model is used if you search for an occurrence of the model at an angle.
---------------------	---

The **Mode** parameter controls the speed of the allocation process:

M_FAST (M_DEFAULT)	High-speed allocation of models.
M_BEST	High-precision allocation of models.

You can add the following setting to the **Mode** parameter:

M_MULTIPLE+ <i>n</i>	Allocates several models from the same image. Set <i>n</i> to the number of models to allocate. For example, set Mode to M_BEST+M_MULTIPLE+4 to find and allocate the four most unique models available in your image. Models can overlap within the image by half the size of the model (in x and y).
----------------------	--

The **ModelIdArrayPtr** parameter specifies the address of the array in which the model identifiers are to be written. The identifier is required to use a model with other pattern matching functions. Since **MpatAllocAutoModel()** also returns the model identifier, you can set this parameter to M_NULL when allocating a single model. If allocation fails, M_NULL is written as the identifier.

Return value The returned value is the first model identifier. If allocation fails, M_NULL is returned.

- Status**
- This function currently supports only unsigned 8-bit grayscale images.
 - You can only define models that respect the following condition:
 $(\text{maxvalue}^2 * \text{SizeX} * \text{SizeY}) < 2^{32}$; where *maxvalue* is the maximum pixel value (typically, 255) in the target image and the model. This restriction is imposed to avoid overflows in the internal 32-bit accumulators.
 - This function is not optimized for orientation type models. After allocation, you should check to make sure the model contains all significant edges of the target image.

Default Models are associated with a set of default search parameters. For an
values M_NORMALIZED model type, the model's search parameters are set to the
 following defaults:

Search region:	The default search region of each model is automatically set using the original position of the model, plus or minus the specified positional uncertainty.
Positional accuracy:	M_MEDIUM (typically ± 0.10 pixels)
Search number:	1
Search speed:	M_MEDIUM
Acceptance level:	70%
Certainty level:	80%
Search angle:	0°
Center of model (reference position):	A double value equal to: ((SizeX - 1) / 2.0, (SizeY - 1) / 2.0) (relative to the model origin).

Example mshift.c

See also **MpatFree()**, **MpatInquire()**, **MpatSet...()**, **MpatFindModel()**,
 MpatFindOrientation()

MpatAllocModel

Synopsis Allocate a pattern matching model from a source image.

Format `MIL_ID MpatAllocModel(SystemId, SrcImageBufId, OffX, OffY, SizeX, SizeY, ModelType, ModelIdPtr)`

MIL_ID SystemId;	System identifier
MIL_ID SrcImageBufId;	Source image buffer
long OffX;	X-coordinate of model origin in the source image
long OffY;	Y-coordinate of model origin in the source image
long SizeX;	Model width
long SizeY;	Model height
long ModelType;	Model type
MIL_ID *ModelIdPtr;	Storage location for model identifier

Description This function allocates a model, using data from the specified area of the model's source image.

You can change a model's search parameters at any time, using the appropriate **MpatSet...()** command. When the model is no longer required, you should release its memory, using **MpatFree()**.

When you only need the orientation of a large object and your target image is appropriate, it is fastest to use an M_ORIENTATION type model (**MpatFindOrientation()**).

For an M_NORMALIZED type model (with or without overscan data), define the angular range in which the model can appear using **MpatSetAngle()**. Angular search is fastest when performed with an M_CIRCULAR_OVERSCAN model; however, M_CIRCULAR_OVERSCAN should only be used when the region around the model is consistent. An example is the image of an integrated circuit. An M_NORMALIZED type model (without overscan data) is used usually for an image with an inconsistent surrounding region, such as an image of loose nuts and bolts lying on a metal sheet.

When preprocessing an `M_NORMALIZED` model (without overscan data) for which an angular search range is specified, rotated versions of the model are created assigning "don't care" pixels to regions that do not have corresponding data in the original model.

When preprocessing an `M_CIRCULAR_OVERSCAN` model for which an angular search range has been specified, a set of models is extracted from rotated versions of the `M_CIRCULAR_OVERSCAN` model, creating models that would appear upright if the target image were rotated. For this type of model, a larger region than the one defined is extracted from the model's source image so as to allow creation of models at different angles.

The **SystemId** parameter specifies the system on which to allocate the model. This parameter must be set to a valid system identifier, `M_DEFAULT_HOST` or `M_DEFAULT`. To use the default Host system of the current MIL application, specify `M_DEFAULT_HOST`. If you specify `M_DEFAULT`, MIL will select the most appropriate system on which to allocate the model (the Host system or any already allocated system).

The **SrcImageBufId** parameter specifies the identifier of the image buffer from which to extract the model (that is, the model's source image).

The **OffX** and **OffY** parameters specify the coordinates of the model origin in the source image (**SrcImageBufId**).

The **SizeX** and **SizeY** parameters specify the width and height of the model.

The specified model width (**SizeX**), height (**SizeY**), and position (**OffX**, **OffY**) must be valid in the source image.

The **ModelType** parameter specifies the type of the model.

<code>M_NORMALIZED</code>	A model used to search for the position, match score, and angle (if angular search is enabled) of a model occurrence in a target image, using MpatFindModel() and MpatFindMultipleModel() .
<code>M_ORIENTATION</code>	A model used to search for the global orientation of the model in a target image, using MpatFindOrientation() . Note, an <code>M_ORIENTATION</code> type model is created from the contours of the specified area in the model's source image. The model's source image should be a typical target image and the model should contain all significant edges of the target image. The source and target images should have a fairly uniform background.

M_NORMALIZED+ M_ORIENTATION	Combining the types allows the model to be used to search for the position, match score and angle (if angular search is enabled) of a model occurrence with MpatFindModel() . Alternatively, it allows the model to be used to search the global orientation of the model in the specified image with MpatFindOrientation() .
--------------------------------	---

You can add the following setting to the above settings:

M_CIRCULAR_OVERSCAN	<p>Extracts the model, as well as circular overscan data from the source image. The overscan data is determined by rotating the area from which to extract the model, about its center. The overscan data of an M_CIRCULAR_OVERSCAN type model is used if you search for an occurrence of the model at an angle.</p> <p>This type of model should only be used when the region around the model is consistent.</p> <p>The model must not be extracted from a region too close to the edge of the model's source image. In addition, the M_CIRCULAR_OVERSCAN model should be complex enough so that if models are created from it at the required angles, they are representative of the pattern being sought.</p>
---------------------	---

The **ModelIdPtr** parameter specifies the address of the variable in which the model identifier is to be written. This identifier is required to use the model with other pattern matching functions. Since **MpatAllocModel()** also returns the model identifier, you can set this parameter to M_NULL.

Return value The returned value is the first model identifier. If allocation fails, M_NULL is returned.

- Status**
- This function currently supports only unsigned 8-bit grayscale images.
 - You can only define models that respect the following condition:
 $(\text{maxvalue}^2 * \text{SizeX} * \text{SizeY}) < 2^{32}$; where *maxvalue* is the maximum pixel value (typically, 255) in the target image and the model. This restriction is imposed to avoid overflows in the internal 32-bit accumulators.

Default values Models are associated with a set of default search parameters. For an M_NORMALIZED model type, the model's search parameters are set to the following defaults:

Positional accuracy:	M_MEDIUM (typically ± 0.10 pixels)
Positional uncertainty (search region):	M_ALL (full image)
Search number:	1

Search speed:	M_MEDIUM
Acceptance level:	70%
Certainty level:	80%
Search angle:	Disabled
Center of model (reference position):	A double value equal to: ((SizeX - 1) / 2.0, (SizeY - 1) / 2.0) (relative to the model origin).

Examples morien2.c, mpatrot.c, msearch.c

See also **MpatFree()**, **MpatSet...**(), **MpatAllocAutoModel()**, **MpatFindModel()**, **MpatFindOrientation()**

MpatAllocResult

Synopsis Allocate a pattern matching result buffer.

Format MIL_ID MpatAllocResult(SystemId, NbEntries, PatResultIdPtr)

MIL_ID SystemId;	System identifier
long NbEntries;	Number of result buffer entries
MIL_ID *PatResultIdPtr;	Storage location for pattern matching result buffer identifier

Description This function allocates a result buffer with the specified number of entries. When the result buffer is no longer required, release its memory, using **MpatFree()**.

The **SystemId** parameter specifies the system on which the result buffer will be allocated. This parameter must be set to a valid system identifier, M_DEFAULT_HOST, or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to allocate the buffer (the Host system or any already allocated system).

The **NbEntries** parameter specifies the number of result entries to allocate. This value should be greater than or equal to the number of occurrences being sought (specified with **MpatSetNumber()**). If **NbEntries** is set to M_DEFAULT, the number of entries will be allocated dynamically to match the number of actual occurrences found at runtime; used in conjunction with an M_ALL search (**MpatSetNumber()**), this provides an efficient method of allocating the correct size result buffer.

The **PatResultIdPtr** specifies the address of the variable in which the pattern matching result buffer identifier is to be written. Since the **MpatAllocResult()** function also returns the pattern matching result buffer identifier, you can set this parameter to M_NULL. If allocation fails, M_NULL is written as the identifier.

Return value The returned value is the pattern matching result buffer identifier. If allocation fails, M_NULL is returned.

Examples morien1.c, morien2.c

See also MpatFree()

MpatAllocRotatedModel

Synopsis Rotate a pattern matching model.

Format **MIL_ID MpatAllocRotatedModel(SystemId, SrcModelId, Angle, InterpolationMode, ModelType, NewModelIdPtr)**

MIL_ID SystemId;	System identifier
MIL_ID SrcModelId;	Source model identifier
double Angle;	Angle of rotation
long InterpolationMode;	Interpolation mode used in rotation
long ModelType;	Type of model to rotate
MIL_ID *NewModelIdPtr;	Storage location for new rotated model identifier

Description This function allocates a new model and initializes it with the rotated version of the specified source model.

This function has the same effect as creating an M_NORMALIZED model (without M_CIRCULAR_OVERSCAN) for which a single angle is specified (that is, delta minimum and maximum are set to 0).

Note that the allocated model size is usually greater than the size of its source, due to the rectangular shape of the model and the nature of the rotation operation. The additional pixel locations produced by the rotation, which have no corresponding pixels in the source model, are set to “don't care” values. The size (X and Y) of the new model can be inquired, using **MpatInquire()**.

Note that the settings of the source model are copied to the new model. You can change any of the model's search parameters (using any appropriate **MpatSet...()** function), except its angle (**MpatSetAngle()** function).

Once you have a rotated version of a model, if you no longer require the original source model, it can be deleted with **MpatFree()**.

The **SystemId** parameter specifies the system on which to allocate the rotated model. This parameter must be set to a valid system identifier, M_DEFAULT_HOST or M_DEFAULT. To use the default Host system of the current MIL application, specify

M_DEFAULT_HOST. If you specify **M_DEFAULT**, MIL will select the most appropriate system on which to allocate the model (the Host system or any already allocated system).

The **SrcModelId** parameter specifies the identifier of the source model. It must be an **M_NORMALIZED** model without overscan data. A new rotated model is generated from this model.

The **Angle** parameter specifies the angle at which to rotate the model in a counter-clockwise direction. This parameter can be set to any value from 0° to 360°.

The **InterpolationMode** parameter specifies the mode of interpolation. This parameter can be set to one of the following:

M_NEAREST_NEIGHBOR	Nearest neighbor interpolation.
M_BILINEAR	Bilinear interpolation.
M_BICUBIC	Bicubic interpolation.
M_DEFAULT	Same as M_NEAREST_NEIGHBOR .

The **ModelType** parameter specifies the type of the new model:

M_NORMALIZED	A model used to search for the position and match score of a model occurrence in a target image, using MpatFindModel() and MpatFindMultipleModel() .
---------------------	--

The **NewModelIdPtr** parameter specifies the address of the variable in which to write the rotated model identifier. Since the **MpatAllocRotatedModel()** function also returns the model identifier, you can set this parameter to **M_NULL**.

Return value The returned value is the new model identifier. If the rotation fails, **M_NULL** is returned.

Default values Models are associated with a set of default search parameters. For an **M_NORMALIZED** model type, the model's search parameters are set to the following defaults:

Positional accuracy:	M_MEDIUM (typically ± 0.10 pixels)
Positional uncertainty (search region):	M_ALL (full image)
Search number:	1
Search speed:	M_MEDIUM

Acceptance level:	70%
Certainty level:	80%
Search angle:	0°
Center of model (reference position):	A double value equal to ((SizeX - 1) / 2, (SizeY - 1) / 2) (relative to the model origin).

See also **MpatSetPosition()**, **MpatSetSearchParameter()**, **MpatFree()**, **MpatInquire()**, **MpatFindModel()**

MpatCopy

Synopsis Copy a pattern matching model to an image buffer.

Format `void MpatCopy(ModelId, DestImageBufId, CopyMode)`

MIL_ID ModelId;	Model identifier
MIL_ID DestImageBufId;	Destination image buffer identifier
long CopyMode;	Copy mode to use

This function copies the specified model to the specified destination image buffer, starting at the top-left corner of the buffer. Once the model is copied to the destination buffer, it can then be displayed (if the destination buffer is displayable).

By making two calls to this function, one in which `M_DEFAULT` is used as the copy mode and the other in which `M_DONT_CARE` is used, it is possible to achieve the effect of overlaying the "don't care" pixels onto the original model.

The **ModelId** parameter specifies the identifier of the model that you want copied to the image buffer.

The **DestImageBufId** parameter specifies the identifier of the destination image buffer in which to place the model. You must ensure that the buffer is at least as large as the model. Note that the function only supports unsigned 8-bit grayscale images.

The **CopyMode** parameter specifies how the model will be copied.

M_DEFAULT	Copies the portion of the source image from which the model was extracted to the destination buffer. That is, the model as it was first defined before any "don't care" pixels where associated with it (see MpatSetDontCare()). Overscan data is not copied.
M_DONT_CARE	Copies only the "don't care" pixels of the model to the destination buffer. When copied, these pixels are given the value zero. To give these pixels a value other than zero, add the value to the <code>M_DONT_CARE</code> copy mode. Note, by default when using the <code>M_DONT_CARE</code> copy mode, any pixel value other than the "don't care" pixels in the destination image buffer will not be overwritten.
M_DONT_CARE+ M_CLEAR_BACKGROUND	Pixels in the resulting destination buffer that are not "don't care" pixels are cleared to zero.

Example The following example demonstrates how to copy "don't care" pixels as value 255 and set the other pixels in the destination buffer to zero.

```
MpatCopy(ModelId, ImageBufId, M_DONT_CARE +255 +M_CLEAR_BACKGROUND)
```

See also **MpatSetDontCare()**

MpatDraw

Synopsis Draw specific features of the model or result occurrence in the destination image buffer.

Format **void MpatDraw(GraphContId, ModelOrResultId, DestImageId, Operation, Index, ControlFlag)**

MIL_ID GraphContId;	Graphics context identifier
MIL_ID ModelOrResultId;	Identifier of model or result buffer
MIL_ID DestImageId;	Destination image buffer identifier
long Operation;	Drawing operation
long Index;	Index of the result
long ControlFlag;	Control flag

Description This function provides graphics utilities to draw specific model or result occurrence features in the destination buffer. **MpatDraw()** can be used with multiple results.

The **GraphContId** parameter specifies the graphics context to use. The graphics context must have been previously allocated using **MgraAlloc()**. Set this parameter to M_DEFAULT to use the default graphic context.

The **ModelOrResultId** parameter specifies the model or result buffer identifier from which to extract the features to draw.

The **DestImageId** parameter specifies the identifier of the destination image buffer into which to draw. You can also choose to annotate an image non-destructively, by drawing into its display's overlay buffer.

The **Operation** parameter specifies the type of operation to perform. Operations can be added together to draw multiple features at a time. For example, to draw both the result occurrence's bounding box and position, you would specify M_DRAW_BOX+M_DRAW_POSITION as the **Operation** parameter. The **Operation** parameter can be set to a combination of the following values:

Operation	Description
M_DRAW_BOX	Draws the bounding box of the model or of the result occurrence(s). For a result occurrence, the box is drawn at the coordinates at which its origin was found in the target image (not the reference position), respecting the occurrence's angle.

Operation	Description
M_DRAW_POSITION	Draws a cross at the model's reference position, or at the result occurrence(s) position and angle found in the target image (with respect to the reference position).
M_DRAW_IMAGE	Draws the region of the model source image from which the model was extracted. (equivalent to MpatCopy()) (for models only).
M_DRAW_DONT_CARES	Draws the model's don't care pixels (for models only).

The **Index** parameter specifies the index of the result occurrence to draw. M_DEFAULT always refers to all results. Note, when drawing features of a model, the index must be set to M_DEFAULT.

The **ControlFlag** parameter specifies, when drawing a model, whether to draw at the top-left corner of the destination image (M_DEFAULT) or at the offsets used to define the model region in the source image (M_ORIGINAL). When drawing features from a result buffer, this parameter must be set to M_DEFAULT.

See also **MgraAlloc()**

MpatFindModel

Synopsis Find a pattern matching model in the target image buffer.

Format `void MpatFindModel(ImageBufId, ModelId, PatResultId)`

MIL_ID ImageBufId;	Image buffer identifier
MIL_ID ModelId;	Model identifier
MIL_ID PatResultId;	Pattern matching result buffer identifier

Description This function finds occurrences of the specified model in the given image and returns the position of each occurrence. The search is performed using the model's current search parameters. Note, this function assumes that the model has been preprocessed (**MpatPreprocModel()**).

This function assumes that the model was extracted from a source image with the same scaling as the target image. In addition, if using a normalized grayscale non-rotational model, it assumes that the source image and the model's target image are of the same orientation (with approx. 5° tolerance). For angular search, use **MpatSetAngle()** to set the angular search range of the model.

The model's search parameters specify the maximum number of occurrences for which to look in the target image (**MpatSetNumber()**).

If a correlation has a match score equal or above the certainty level (**MpatSetCertainty()**), it is automatically considered an occurrence (default 80%), the remaining occurrences will be the best of those above the acceptance level (**MpatSetAcceptance()**).

Results are written to the specified result buffer. Use the **MpatGet..()** functions to read the results. See **MpatAllocResult()** for more information on allocating result buffers.

This function returns results in decreasing match-score order. This means that the most-likely occurrence is always returned first.

Patterns that are very close to the edge of the image might be found with lower match scores than usual due to edge effects. For this reason, you should use **MpatSetPosition()**, rather than a child buffer, to restrict the search to a portion of the image.

The **ImageBufId** parameter specifies the identifier of the target image. Note that this function only supports unsigned 8-bit grayscale images.

The **ModelId** parameter specifies the identifier of the model for which to search in the specified target image buffer.

The **PatResultId** parameter specifies the identifier of the pattern matching result buffer in which to store results.

Examples mpatrot.c, msearch.c, mshift.c

See also **MpatFindMultipleModel()**, **MpatGetNumber()**, **MpatGetResult()**, **MpatPreprocModel()**, **MpatSetAngle()**

MpatFindMultipleModel

Synopsis Find multiple pattern matching models in the target image buffer.

Format `void MpatFindMultipleModel(ImageBufId, ModelIdArray, PatResultIdArray, NumModels, SearchMode)`

MIL_ID ImageBufId;	Image buffer identifier
MIL_ID *ModelIdArray;	List of model identifiers
MIL_ID *PatResultIdArray;	List of result buffer identifiers
long NumModels;	Number of models in list
long SearchMode;	Search mode

Description 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. If you want to search for several models in a given image region, a single call to this function is more efficient than multiple calls to **MpatFindModel()**. This function assumes all models have been preprocessed.

This function assumes that each model was extracted from an image with the same scaling as the target image. In addition, if using a normalized grayscale non-rotational model, it assumes that the model's source image and the target image are at the same orientation (with approximately 5° tolerance). For angular search, use the **MpatSetAngle()** function to set the angular search range of the model.

Results are written to the specified result buffers. This function returns results in decreasing match-score order. This means that the most-likely occurrence is always returned first. Use the **MpatGet..()** functions to read the results. See **MpatAllocResult()** for more information on allocating result buffers.

Patterns that are very close to the edge of the image might be found with lower match scores than usual due to edge effects. For this reason, you should use **MpatSetPosition()**, rather than a child buffer, to restrict the search to a portion of the image. Note that all models must use the same search region.

The **ImageBufId** parameter specifies the identifier of the target image buffer. Note that this function only supports unsigned 8-bit grayscale images.

The **ModelIdArray** parameter specifies the address of the one-dimensional array that contains the identifiers of the models for which to search in the specified target image buffer. Note, all models must be of the same size and must use the same search region in the target image.

The **PatResultIdArray** parameter specifies the address of the one-dimensional array that contains the identifiers of the pattern matching result buffers in which to store results.

The **NumModels** parameter specifies the number of models.

The **SearchMode** parameter specifies the search mode and can be set to one of the following:

M_FIND_ALL_MODELS	Searches for all the specified models in the target image and writes the result of each model search in the corresponding buffer. The search is performed using each model's current search parameters. Each model's search parameters also determine the maximum number of occurrences for which to look in the target image (MpatSetNumber()). If a correlation has a match score above the model's certainty level (MpatSetCertainty()), it is automatically considered an occurrence (default 80%), the remaining occurrences will be the best of those above the acceptance level (MpatSetAcceptance()). Note, the results for the model in the ModelIdArray[n] are written in the corresponding result buffer PatResultArray[n] . Therefore, you must specify the same number of result buffers as the number of models.
M_FIND_BEST_MODELS	Searches for the specified models in the target image and writes the results which have the highest match scores in a single result buffer (PatResultArray[0]). Note, the number of different matches that are found depends on MpatSetNumber() , MpatSetCertainty() , and MpatSetAcceptance() of the first model in the array. The certainty, speed, angle and all other search parameters are also specified by the first model in the array; the search parameter settings of the other models are ignored. The model index of the model that generated the best match scores can be read from the result buffer using MpatGetResult(.., M_MODEL_INDEX,...) .

See also **MpatFindModel()**, **MpatGetNumber()**, **MpatGetResult()**

MpatFindOrientation

Synopsis Find the orientation of an image or of an object in an image.

Format **void MpatFindOrientation(ImageBufId, ModelId, FindResultId, ResultRange)**

MIL_ID ImageBufId;	Image buffer identifier
MIL_ID ModelId;	Model identifier
MIL_ID FindResultId;	The identifier of the result buffer
long ResultRange	Range of possible result angles

Description This function finds the orientation (angle, not position) of a whole unsigned 8-bit grayscale image or of an M_ORIENTATION-type model acquired from an unsigned 8-bit grayscale source image.

This method finds the orientation of a whole image, based on the dominant edges and their angular displacement from the image frame. If the image does not have dominant edges, its orientation is not well defined. In addition, if the image's background contains edges, the orientation of these edges might be found instead.

To find a specific object's orientation, a model must be provided. This model should be a single large object on a smooth uniform background allocated with an M_ORIENTATION model type, using **MpatAllocModel()**. This function will then search for the general contours of the object in a target image. You can request that multiple possible angles be returned to the result buffer, using **MpatSetNumber()**. Make sure you don't search for more matches than will fit in the result buffer. If you search for all occurrences (M_ALL) and the result buffer is too small, **MpatFindOrientation()** will return an error indicating that the buffer has overflowed.

Note, this function is designed for images with smooth edges, usually obtained when grabbing an image with a camera. It will not work well on an artificially-generated image unless the lines and edges are anti-aliased. In particular, it will not work on a nearest-neighbor resampled image, such as the result of **MimRotate()** with M_NEAREST_NEIGHBOR interpolation, unless the image is heavily smoothed.

The **ImageBufId** parameter specifies the identifier of an unsigned 8-bit grayscale target image buffer.

The **ModelId** parameter specifies the identifier of the model for which to search in the specified image buffer. The model must be of type `M_ORIENTATION`. If no model is specified (`M_NULL`), whole-image orientation is done, using the dominant edges of the image as the major axes.

The **FindResultId** parameter specifies the identifier of the result buffer in which to store results. The number of matches found can be obtained, using **MpatGetNumber()**. The match angles and scores can be obtained, using **MpatGetResult()**.

The **ResultRange** parameter specifies the range in which possible result angles can occur.

In whole-image orientation searches, edges serve as the axes for orientation purposes. The image can have unidirectional predominant edges (such as parallel stripes); in this case the result range should be set to `M_RESULT_RANGE_180`. Alternatively, the image can have bidirectional edges (such as an electronic wafer) whereby the two dominant edges should be perpendicular (90°). In this case, the result range can be set to `M_RESULT_RANGE_45` or `M_RESULT_RANGE_90`.

In model-orientation searches, a 360° result range is necessary to include all the rotational possibilities of the model.

The **ResultRange** parameter can be set to one of the following:

ResultRange	Description
<i>For whole-image orientation:</i>	
<code>M_RESULT_RANGE_45</code>	Searches and returns a value between -45° and 45° . Only available for images with bidirectional edges.
<code>M_RESULT_RANGE_90</code>	Searches and returns a value between 0° and 90° . Only available for images with bidirectional edges.
<code>M_RESULT_RANGE_180</code>	Searches and returns a value between 0° and 180° . Only available for images with unidirectional edges.
<i>For model orientation:</i>	
<code>M_RESULT_RANGE_360</code>	Searches and returns a value between 0° and 360° .

Examples `morien1.c`, `morien2.c`

See also `MpatGetNumber()`, `MpatGetResult()`, `MpatSetNumber()`

MpatFree

Synopsis Free a pattern matching model or a result buffer.

Format **void MpatFree(PatId)**

MIL_ID PatId;	Pattern matching model or result buffer identifier
---------------	--

Description This function deletes the specified pattern matching model or result buffer identifier and releases any memory associated with it.

The **PatId** parameter specifies the identifier of the pattern matching model or result buffer to free. These must have been successfully allocated (with **MpatAllocModel()**, **MpatAllocAutoModel()**, or **MpatAllocResult()**) prior to calling this function.

MpatGetNumber

Synopsis Get the number of model occurrences in the target image.

Format **long MpatGetNumber(PatResultId, CountPtr)**

MIL_ID PatResultId;	Pattern matching result buffer identifier
long *CountPtr;	Storage location for the count

Description This function is used to determine the number of matches found after searching for a model. It returns the number of occurrences of the model that were found with match scores equal to or above the model's acceptance level to a maximum of **MpatSetNumber()**. This number is the number of results that will be retrieved when you call **MpatGetResult()**.

A call to **MpatGetNumber()** should be made prior to checking any results with **MpatGetResult()**; in the case that no occurrences of the model were found, there would be no need to check the results. Also, the returned number will never be bigger than the number of entries allocated in the result buffer.

The **PatResultId** parameter specifies the identifier of the pattern matching result buffer that was used to store results obtained by **MpatFindModel()**, **MpatFindMultipleModel()**, or **MpatFindOrientation()**.

The **CountPtr** parameter specifies the address of the variable in which to write the number of occurrences. Since the **MpatGetNumber()** function also returns the requested information, you can set this parameter to **M_NULL**.

Return value The returned value is the number of occurrences of the model that were found at or above the acceptance level.

Examples mpatrot.c, msearch.c, mshift.c

See also **MpatSetNumber()**

MpatGetResult

Synopsis Get the pattern matching result values.

Format `void MpatGetResult(PatResultId, ResultType, UserArrayPtr)`

MIL_ID PatResultId;	Pattern matching result buffer identifier
long ResultType;	Type of results
double *UserArrayPtr;	Array for results

Description This function writes the match score, position of each occurrence, or angle of orientation stored in the specified result buffer, to the specified user array.

Note that if your target image has been associated with a calibration object, positional results are by default returned in real-world units. To have results returned in pixel units, use **McalControl()** with the `M_OUTPUT_COORDINATE_SYSTEM` control type set to `M_PIXEL`.

The **PatResultId** parameter specifies the identifier of the pattern matching result buffer that was used to store results obtained by **MpatFindModel()**, **MpatFindMultipleModel()**, or **MpatFindOrientation()**.

The **ResultType** parameter specifies the type of result that you want returned for each model occurrence. This parameter can be set to one of the following values:

ResultType	Description
M_SCORE	The match score of the occurrence (as a percentage).
M_POSITION_X	X-coordinate of the occurrences.
M_POSITION_Y	Y-coordinate of the occurrences.
M_ANGLE	The angle from an MpatFindModel() search with angle.
M_MODEL_INDEX	The index of the model, in the model identifier array, for which the occurrence was found when using MpatFindMultipleModel with <code>M_FIND_BEST_MODELS</code> .
M_ORIENTATION	The angle of orientation from MpatFindOrientation() .

The **UserArrayPtr** parameter specifies the address of the one-dimensional array in which to write the specified results. The array must be big enough to hold the number of results indicated by **MpatGetNumber()**. Results are returned in the user array as type "double".

Examples morien1.c, morien2.c, mpatrot.c, mshift.c

See also **MpatFindModel()**, **MpatFindMultipleModel()**, **MpatGetNumber()**, **MpatAllocResult()**

MpatInquire

Synopsis Inquire about the pattern matching model or the result buffer parameter setting.

Format **void MpatInquire(PatId, ParamToInquire, UserVarPtr)**

MIL_ID PatId;	Pattern matching model identifier or result buffer identifier
long ParamToInquire;	Parameter to inquire
void *UserVarPtr;	Storage location for inquired information

Description This function returns information about the specified model or result buffer. It is useful to determine, for example, the size of a model that has been restored from disk and its position in the model's source image.

The position (M_ORIGINAL_X and M_ORIGINAL_Y) can be directly compared with search results (by **MpatFindModel()** or **MpatFindMultipleModel()**), to calculate the displacement of target images relative to the model's source image.

The **PatId** parameter specifies the identifier of the model or result buffer for which to read the information.

The **ParamToInquire** parameter specifies the parameter about which to inquire. For a model identifier, this parameter can be set to one of the following values::

ParamToInquire	Description
M_ACCEPTANCE_THRESHOLD	Minimum acceptable match score to be considered as an occurrence of the model (set using MpatSetAcceptance()).
M_ALLOC_OFFSET_X	X-offset of model's top-left corner relative to the top-left corner of the model's source image (set using MpatAlloc...()).
M_ALLOC_OFFSET_Y	Y-offset of model's top-left corner relative to the top-left corner of the model's source image (set using MpatAlloc...()).
M_ALLOC_SIZE_X	Model width (set using MpatAlloc...()).
M_ALLOC_SIZE_Y	Model height (set using MpatAlloc...()).
M_ALLOC_TYPE	Model type (set using MpatAlloc...()).

ParamToInquire	Description
M_CENTER_X	X-offset of model's reference position relative to the top-left corner of model (set using MpatSetCenter()).
M_CENTER_Y	Y-offset of model's reference position relative to the top-left corner of model (set using MpatSetCenter()).
M_CERTAINTY_THRESHOLD	Match score at which an occurrence is assumed, without looking for better matches elsewhere in the image (set using MpatSetCertainty()).
M_COARSE_SEARCH_ACCEPTANCE	Minimum acceptable match score at all levels except the last level, to be considered as an occurrence of the model (set using MpatSetSearchParameter()).
M_EXTRA_CANDIDATES	Number of extra candidates to consider as possible candidates (set using MpatSetSearchParameter()).
M_FAST_FIND	Whether forcing or preventing fast peak finding is used (set using MpatSetSearchParameter()).
M_FIRST_LEVEL	The resolution level for the initial stage of the search (set using MpatSetSearchParameter()).
M_LAST_LEVEL	The resolution level for the final stage of the search (set using MpatSetSearchParameter()).
M_MIN_SPACING_X	The minimum spacing (in X) between two models in order for them to be considered distinct (set using MpatSetSearchParameter()).
M_MIN_SPACING_Y	The minimum spacing (in Y) between two models in order for them to be considered distinct (set using MpatSetSearchParameter()).
M_MODEL_STEP	Whether all or every second model pixel is used in the high-resolution stage of the search. (set using MpatSetSearchParameter()).
M_NUMBER_OF_OCCURRENCES	Number of model occurrences for which to search in the target image (set using MpatSetNumber()).

ParamToInquire	Description
M_ORIGINAL_X	X-offset of the model's reference position relative to the top-left corner of the model's source image (takes into account the MpatSetCenter() setting).
M_ORIGINAL_Y	Y-offset of the model's reference position relative to the top-left corner of the model's source image (takes into account the MpatSetCenter() setting).
M_OWNER_SYSTEM	The system on which the model is allocated.
M_POSITION_ACCURACY	Search position accuracy (set using MpatSetAccuracy()).
M_POSITION_START_X	X-coordinate of search region origin within target image (set using MpatSetPosition()).
M_POSITION_START_Y	Y-coordinate of search region origin within target image (set using MpatSetPosition()).
M_POSITION_UNCERTAINTY_X	Search region width (set using MpatSetPosition()).
M_POSITION_UNCERTAINTY_Y	Search region height (set using MpatSetPosition()).
M_PREPROCESSED	Whether or not the model is preprocessed (0 = no).
M_SEARCH_ANGLE	The value of the initial search angle (set using MpatSetAngle()).
M_SEARCH_ANGLE_ACCURACY	The angular accuracy (set using MpatSetAngle()).
M_SEARCH_ANGLE_DELTA_NEG	The difference that determines the lower limit of the search angle's range (set using MpatSetAngle()).
M_SEARCH_ANGLE_DELTA_POS	The difference that determines the upper limit of the search angle's range (set using MpatSetAngle()).
M_SEARCH_ANGLE_INTERPOLATION_MODE	The interpolation mode.
M_SEARCH_ANGLE_MODE	State of angular search mode (set using MpatSetAngle()).
M_SEARCH_ANGLE_TOLERANCE	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. (set using MpatSetAngle()).
M_SPEED	Model search speed (set using MpatSetSpeed()).

For a result buffer identifier, this parameter can be set to the following value:

M_NUMBER_OF_ENTRIES	The number of entries allocated in the result buffer.
M_TARGET_CACHING	Whether the pyramidal representation of the buffer is kept in the result buffer (set using MpatSetSearchParameter()).
M_OWNER_SYSTEM	The system on which the result buffer is allocated.

The **UserVarPtr** parameter specifies the address of the variable in which to write the requested information. Results are returned in the user variable as type "double". If you want results to be returned as type "long", combine the specified result type with M_TYPE_LONG (for example, M_ORIGINAL_X+M_TYPE_LONG).

MpatPreprocModel

Synopsis Preprocess a pattern matching model.

Format `void MpatPreprocModel(TypicalImageBufId, ModelId, Mode)`

MIL_ID TypicalImageBufId;	Typical background image buffer identifier (optional)
MIL_ID ModelId;	Model identifier
long Mode;	Preprocessing mode

This function preprocesses the specified model. It trains the system to search for the model in the most efficient manner (optionally within a specified typical image). The procedure is potentially quite lengthy (up to several seconds).

Call this function after all search parameters have been set. When you save, the model's preprocessing changes are stored with the model. Upon restoration, the model need not be preprocessed.

Note that if some of the model's search parameters are changed after a call to **MpatPreprocModel()**, the model must be preprocessed again. To inquire if your model is in a preprocessed state, use **MpatInquire()** with **M_PREPROCESSED**.

The **TypicalImageBufId** parameter specifies the identifier of a typical target image. The specified typical image will be used to refine and adapt the model to search on this typical background. You should only specify an image buffer if the model will always appear on such a background; otherwise, set this parameter to **M_NULL**.

The **ModelId** parameter specifies the identifier of the model to preprocess.

The **Mode** parameter specifies the preprocessing mode. Set this parameter to **M_DEFAULT**.

Examples mpatrot.c, msearch.c

MpatRead

Synopsis

Read a pattern matching model from an open file.

Format

MIL_ID MpatRead(SystemId, FileHandle, ModelIdPtr)

MIL_ID SystemId;	System identifier
FILE* FileHandle;	Model file handle
MIL_ID *ModelIdPtr;	Storage location for model identifier

Description

This function reads a model (previously saved with **MpatWrite()** or **MpatSave()**) from an open file and returns an identifier to it.

This function also restores all the model's search parameters that were in effect when the model was saved. If the model was preprocessed before saving, you do not need to preprocess it again.

The **SystemId** parameter specifies the system on which the pattern matching model resides. This parameter must be set to a valid system identifier, M_DEFAULT_HOST or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to allocate the buffer (the Host system or any already allocated system).

The **FileHandle** parameter specifies the handle of the open file (opened with the standard C function **fopen()**). Before calling this function, the file pointer must be positioned just before the start of a valid pattern matching model. After the function call, the file remains open and is positioned immediately after the model.

The **ModelIdPtr** parameter specifies the address of the variable in which the model identifier is to be written. Since the **MpatRead()** function also returns the model identifier, you can set this parameter to M_NULL. If allocation fails, M_NULL is written as the identifier.

Return value

The returned value is the model identifier. If allocation fails, M_NULL is returned.

See also

MpatWrite(), MpatSave(), MpatRestore()

MpatRestore

Synopsis Restore a pattern matching model from disk.

Format **MIL_ID MpatRestore(SystemId, FileName, ModelIdPtr)**

MIL_ID SystemId;	System identifier
MIL_TEXT_PTR FileName;	Model file name
MIL_ID *ModelIdPtr;	Storage location for model identifier

Description This function restores a model that was previously saved to a file, using **MpatSave()**. If the model was preprocessed before saving, you do not need to preprocess it again.

This function also restores all the model's search parameters that were in effect when the model was saved.

The **SystemId** parameter specifies the system on which to restore the model. This parameter must be set to a valid system identifier, M_DEFAULT_HOST or M_DEFAULT. To use the default Host system of the current MIL application, specify M_DEFAULT_HOST. If you specify M_DEFAULT, MIL will select the most appropriate system on which to allocate the buffer (the Host system or any already allocated system).

The **FileName** parameter specifies the model file name. The function handles (internally) the opening and closing of the file.

The **ModelIdPtr** parameter specifies the address of the variable in which to write the model identifier. Since the **MpatRestore()** function also returns the model identifier, you can set this parameter to M_NULL. If allocation fails, M_NULL is written as the identifier.

Return value The returned value is the model identifier. If allocation fails, M_NULL is returned.

Example mrestmod.c

See also **MpatSave()**, **MpatRead()**, **MpatWrite()**

MpatSave

Synopsis Save a pattern matching model to disk.

Format **void MpatSave(FileName, ModelId)**

MIL_TEXT_PTR FileName;	Model file name
MIL_ID ModelId;	Model identifier

Description This function saves all the information about the previously allocated model to disk, including all of the model's current search parameters values and any effects of preprocessing. Later, this information can be reloaded, using **MpatRestore()** or **MpatRead()**.

The **FileName** parameter specifies the model file name. If this file already exists, it will be overwritten. The function handles (internally) the opening and closing of this file.

The **ModelId** parameter specifies the identifier of the model to save.

See also **MpatRestore(), MpatWrite(), MpatRead()**

MpatSetAcceptance

Synopsis Set the acceptance level of a model.

Format void MpatSetAcceptance(ModelId, AcceptanceThreshold)

MIL_ID ModelId;	Model identifier
double AcceptanceThreshold;	Minimum acceptable correlation

Description This function sets the acceptance level for a match made with the specified model when it is sought in an image. If the correlation (match score) between the image and the model is less than this level, it is not considered a match (an occurrence). The default acceptance is set when the model is allocated.

The **ModelId** parameter specifies the identifier of the model for which to change the acceptance threshold search parameter.

The **AcceptanceThreshold** parameter specifies the acceptance level, as a percentage. A perfect match is 100%, no correlation is 0%. The match score depends on the image quality. You should experiment to decide what is a typical match score for your application.

See also MpatSetNumber(), MpatSetCertainty()

MpatSetAccuracy

Synopsis Set the positional accuracy of a model.

Format **void MpatSetAccuracy(ModelId, Accuracy)**

MIL_ID ModelId;	Model identifier
long Accuracy;	Required positional accuracy

Description This function sets the specified model's search parameter for positional accuracy and the complexity of the image. You can enhance speed performance by selecting a lower positional accuracy. Also, whenever the positional accuracy of a model changes, the effect of any preprocessing is undone. Therefore, if the accuracy is changed, call **MpatPreprocModel()** before searching for the model.

The positional accuracy is also slightly affected by the search speed (**MpatSetSpeed()**).

The **ModelId** parameter specifies the identifier of the model for which to change the positional accuracy search parameter.

The **Accuracy** parameter sets the required positional accuracy for the search. Set it to one of the following values:

M_LOW	Low accuracy (typically ± 0.20 pixels)
M_MEDIUM	Medium accuracy (typically ± 0.10 pixels)
M_HIGH	High accuracy (typically ± 0.05 pixels)

Note, the precision achieved is dependent on the quality of the model and the image (the tolerances listed above are typical for high-quality, low-noise images).

The method used to find a model with M_LOW accuracy more quickly can also produce match scores that are slightly lower than usual. If a precise match score is important to you, use at least medium accuracy.

Examples mpatrot.c, msearch.c

MpatSetAngle

Synopsis Set the angular search parameters of a model.

Format `void MpatSetAngle(ModelId, ControlType, ControlValue)`

MIL_ID ModelId;	Model identifier
long ControlType;	Type of control to set
double ControlValue;	Value associated with control type

Description This function sets the specified model's angular search parameters. Also, whenever the angular search parameters of a model change, the effect of any preprocessing of the model is undone. Therefore, if the search angle is changed, call **MpatPreprocModel()** before searching for the model.

The **ModelId** parameter specifies the identifier of the model for which to change the angular search parameter. This can be any type of model except an M_ORIENTATION type model or a model created using **MpatAllocRotatedModel()**.

The **ControlType** and **ControlValue** parameters specify the type of control to set and the value to which to set it.

By default, the **ControlType** M_SEARCH_ANGLE_MODE is disabled and the search is done at 0° (no rotation). When enabled, the model is searched at angle, respecting all search control parameters.

The following **ControlType/ControlValue** combinations can be selected:

ControlType	ControlValue	Description
M_SEARCH_ANGLE_MODE	M_ENABLE or M_DISABLE (M_DEFAULT)	Search at angle.
M_SEARCH_ANGLE	0.0° - 360.0° or M_DEFAULT (0.0°)	see *
M_SEARCH_ANGLE_DELTA_NEG	0.0° - 180.0° or M_DEFAULT (0.0°)	see *
M_SEARCH_ANGLE_DELTA_POS	0.0° - 180.0° or M_DEFAULT (0.0°)	see *
* These angles determine the range of angles to cover in the search operation: (M_SEARCH_ANGLE - M_SEARCH_ANGLE_DELTA_NEG) to (M_SEARCH_ANGLE + M_SEARCH_ANGLE_DELTA_POS) inclusively, starting at an angle close to M_SEARCH_ANGLE.		

ControlType	ControlValue	Description
M_SEARCH_ANGLE_TOLERANCE	0.1° - 180.0° or M_DEFAULT (5.0°)	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. For example, if a model can tolerate the target image being offset from its search angle by $\pm 2.5^\circ$, specify 5°. The specified tolerance determines the step angle. Note that very small tolerance values should only be used within small angular ranges, otherwise application performance can be adversely affected.
M_SEARCH_ANGLE_ACCURACY	0.1° - 180.0°, M_DISABLE, or M_DEFAULT	The required precision of the resulting angle. When set to M_DISABLE or M_DEFAULT, the angular accuracy equals the tolerance.
M_SEARCH_ANGLE_INTERPOLATION_MODE	M_NEAREST_NEIGHBOR (M_DEFAULT), M_BILINEAR, or M_BICUBIC	Determines the type of interpolation that is used when rotating the model.

Example mpatrot.c

See also **MpatSetAcceptance()**, **MpatFindModel()**, **MpatFindMultipleModel()**, **MpatFindOrientation()**

MpatSetCenter

Synopsis Set the reference position of a model.

Format `void MpatSetCenter(ModelId, OffX, OffY)`

MIL_ID ModelId;	Model identifier
double OffX;	X offset relative to model top-left corner
double OffY;	Y offset relative to model top-left corner

Description This function sets the reference position of the specified model. The default reference position is equal to $((SizeX - 1) / 2.0, (SizeY - 1) / 2.0)$ (relative to the model origin). All positional search results are based on this defined model's reference position, relative to the top-left corner of the image in which the search is performed.

The **ModelId** parameter specifies the identifier of the model for which to change the reference position.

The **OffX** and **OffY** parameters specify the offset of the new model's reference position relative to the model origin. Note that the reference position need not be in the model.

MpatSetCertainty

Synopsis Set the certainty level of a model.

Format **void MpatSetCertainty(ModelId, CertaintyThreshold)**

MIL_ID ModelId;	Model identifier
double CertaintyThreshold;	Certainty level (in percent)

Description This function sets the certainty level for a match made with the specified model when it is sought in an image. If the correlation (match score) between the image and the model is equal to or greater than the certainty level, a match is assumed without looking elsewhere in the image for a better match. The default certainty level is set when the model is allocated (**MpatAlloc...()**).

The **ModelId** parameter specifies the identifier of the model for which to change the certainty threshold search parameter.

The **CertaintyThreshold** parameter specifies the certainty level, as a percentage. If you set the certainty level too high (close to 100%), you slow down the search because you force the search algorithm to check the whole search region for the best match. A good certainty level is slightly lower than the expected score, so that the search can finish as soon as the match is found. However, if you set the certainty level too low, false matches might be found. Note that the certainty level must be equal to or greater than the acceptance level (**MpatSetAcceptance()**).

See also **MpatSetNumber(), MpatSetAcceptance(), MpatAlloc...()**

MpatSetDontCare

Synopsis Set the "don't care" pixels in a model.

Format `void MpatSetDontCare(ModelId, ImageBufId, OffX, OffY, Value)`

MIL_ID ModelId;	Model identifier
MIL_ID ImageBufId;	"Don't-care" image buffer identifier
long OffX;	Start of "don't care" mask in X direction
long OffY;	Start of "don't care" mask in Y direction
long Value;	Pixel value in image to be considered as "don't care" value

Description This function sets pixels in the specified model to a "don't care" state. The "don't care" pixels of the model will not be considered when searching for occurrences of the model in the target image (**MpatFindModel()** or **MpatFindMultipleModel()**). To determine which model pixels to set, a region of the specified image buffer, starting at the specified offset and equal in size to the model, is compared with the specified "don't care" value. If an image pixel is equal to this value, the corresponding pixel in the model is set to "don't care".

Note, each time this function is called, a new set of "don't care" pixels is assigned to the specified model, superseding any existing set. Therefore, multiple calls to this function do not have a cumulative effect. Also, whenever the "don't care" pixels in a model change, the effect of any preprocessing of the model is undone. Therefore, if the "don't care" pixels are changed, call **MpatPreprocModel()** before searching for the model.

The **ModelId** parameter specifies the identifier of the model in which to set "don't care" pixels.

The **ImageBufId** parameter specifies the identifier of the image buffer used to identify which pixels in the model will be set to "don't care". This buffer must be at least as large as the model.

The **OffX** and **OffY** parameters specify the X and Y offset from the upper-left corner of the specified image buffer to the upper-left corner of the pixel-value comparison area. The size of the comparison area is determined by the size of the model.

The **Value** parameter specifies the pixel value in the image buffer that determine the corresponding "don't care" pixels.

See also **MpatCopy()**, **MpatPreprocModel()**, **MpatFindModel()**, **MpatFindMultipleModel()**

MpatSetNumber

Synopsis Set the expected number of occurrences of a model.

Format void MpatSetNumber(ModelId, NbOccurrences)

MIL_ID ModelId;	Model identifier
long NbOccurrences;	Maximum number of occurrences to find

Description This function sets the number of occurrences of a model for which to search when using **MpatFindModel()** or **MpatFindMultipleModel()**. Occurrences with match scores that attain/exceed the certainty level (set with **MpatSetCertainty()**) are returned, up to the number specified in **NbOccurrences**. If such occurrences are fewer than the specified number, the remaining occurrences returned are the best of those that attain/exceed the acceptance level (set with **MpatSetAcceptance()**). The default is 1 (that is, find only the first occurrence with a match score that attains/exceeds the certainty level).

The **ModelId** parameter specifies the identifier of the model for which to set the expected number of occurrences.

The **NbOccurrences** parameter specifies the maximum number of occurrences for which to look in the target image. The number of occurrences should be less than or equal to the number of result buffer entries or some results may be lost. To return **all** matches that attain/exceed the acceptance level, set this parameter to M_ALL.

See also MpatSetAcceptance(), MpatSetCertainty(), MpatFindModel(), MpatFindMultipleModel(), MpatGetNumber(), MpatGetResult()

MpatSetPosition

Synopsis Set the search region of a model.

Format `void MpatSetPosition(ModelId, OffX, OffY, SizeX, SizeY)`

MIL_ID ModelId;	Model identifier
long OffX;	X-coordinate of search region origin within target image
long OffY;	Y-coordinate of search region origin within target image
long SizeX;	Width of search region
long SizeY;	Height of the search region

Description This function sets the specified model's search region. It limits the area in the target image in which to find the reference position of the model (set with **MpatSetCenter()**), and consequently increases the search speed. This function is useful when the model's reference position is expected to be found in a certain area. You can change the search region at any time (for example, when tracking an object, the search region could be changed before each call to **MpatFindModel()** or **MpatFindMultipleModel()**).

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.

Always use this function, rather than a child buffer, to restrict the search region. Other methods result in edge effects, and if the search region is as small as or smaller than the model, other methods produce invalid results.

The **ModelId** parameter specifies the identifier of the model for which to change the search region.

The **OffX** and **OffY** parameters specify the coordinates of the top-left corner of the search region in the target image.

The **SizeX** and **SizeY** parameters specify the width and height of search region. Note, it is valid to specify a search region that is smaller than the search model since you are setting the area in which to find the model's reference position.

To change some of the search parameters set with this function without affecting others, specify `M_NO_CHANGE` for those you do not want to change. For example, you can change **OffX** and **OffY** to the required values and keep **SizeX** and **SizeY** as is by setting them to `M_NO_CHANGE`.

To reset the search region size to the full image, set **OffX** and **OffY** to zero (0), and **SizeX** and **SizeY** to `M_ALL`.

MpatSetSearchParameter

Synopsis Set the internal search parameters of a model.

Format void MpatSetSearchParameter(PatId, Parameter, Value)

MIL_ID PatId;	Model identifier or result buffer identifier
long Parameter;	Parameter to be set
double Value;	Value to set parameter to

Description This function is for advanced users of the pattern matching module. It need not be used under most circumstances because the default settings usually provide the best results for search operations. However, if the default values do not satisfy the requirements of your application, this function can be used to set the model's internal search parameters. These internal search parameters are normally derived from the speed and accuracy settings (see **MpatSetSpeed()** and **MpatSetAccuracy()**), but this function gives the experienced user precise control over them. Refer to *Chapter 12: Searches, models, and search parameters* in the *Matrox Imaging Library User Guide* to gain a clearer understanding of how the search algorithm works.

Note that if some of the model's search parameters are changed after a call to **MpatPreprocModel()**, the model must be preprocessed again. To inquire if your model is in a preprocessed state, use the **MpatInquire()** function with M_PREPROCESSED.

Note that the model's internal parameters are all saved and restored with the model, just like the other parameters such as search region, speed, and accuracy.

The **PatId** parameter specifies the identifier of the model whose search parameter to change, or of the result buffer.

The **Parameter** specifies the search parameter to set.

The **Value** parameter specifies the value to which to set the specified parameter.

For a model identifier, this parameter can be set to one of the following values:

Parameter	Values	Description
M_ALL	M_DEFAULT	Determine all search parameters automatically from the current speed and accuracy settings.
M_FIRST_LEVEL	0 - 7	Resolution level for the initial stage (lowest resolution) of the search.
	M_DEFAULT	Determine first level automatically.
		Note: Level 0 is the original target image and each higher level is half the size (and resolution) of the previous one. If the specified level is not supported by the search algorithm, the highest possible level will be used. A higher first level speeds up the initial search but makes it less reliable, because the model might not retain enough distinctive features at such a low resolution.
M_LAST_LEVEL	0 - 7	Resolution level for the final stage (highest resolution) of the search.
	M_DEFAULT	Determine the last level automatically.
		Note: If the specified level is not supported by the search algorithm, the highest acceptable level will be used. Search score can also be less reliable for levels above 0, depending on the characteristics of the model.
M_MODEL_STEP	1	Use all model pixels in the correlation during the high resolution stage of the search.
	2	Use only every second model pixel (in both the X and Y directions) in the correlation during the high resolution stage of the search. This speeds up the last (high resolution) stage of the search, particularly for large models. The match score can be affected if the model has many fine features, but will tend not to be affected if the model has mainly coarse features.
	M_DEFAULT	Determine the model step automatically.
M_FAST_FIND	M_ENABLE	Force fast peak finding. The search algorithm attempts to find the peaks without checking every point. This is safe in most cases, but can cause matches to be missed for models that produce very narrow peaks.
	M_DISABLE	Prevent fast peak finding. The initial search (at the resolution level determined by M_FIRST_LEVEL) computes the correlation at every position in the search region. This guarantees that the biggest match peak will be found, and that it will be investigated first.
	M_DEFAULT	Pre-processing decides if fast peak finding is appropriate.

Parameter	Values	Description
M_ALLOC_OFFSET_X	any integer	Set the model's allocation X offset to the specified value. This might be useful when the original value is lost after rotating a model. You can use MpatInquire() to inquire about the current X offset.
M_ALLOC_OFFSET_Y	any integer	Set the model's allocation Y offset to the specified value. This might be useful when the original value is lost after rotating the model. You can use MpatInquire() to inquire about the current Y offset.
M_MIN_SPACING_X	1.0 to 100.0	Set the minimum spacing (in X) between two models in order for them to be considered distinct.
	M_DEFAULT	Default is 75% of the model width (SizeX of MpatAlloc...()).
M_MIN_SPACING_Y	1.0 to 100.0	Set the minimum spacing (in Y) between two models in order for them to be considered distinct.
	M_DEFAULT	Default is 75% of the model height (SizeY of MpatAlloc...()).
M_COARSE_SEARCH_ACCEPTANCE		
	1.0 to 100.0	Sets the coarse search acceptance threshold level to use for rejecting candidate model peaks at low resolution levels. This can speed up the search when some of the matches you request do not reach the certainty threshold, or when you request more matches than are really present in the image. The coarse search acceptance level should usually be set much lower than the search acceptance level. For example, a good level to set it to is about 20% to 30%. Note that if it is too low, you will not see any increase in speed. However, if it is too high, you risk rejecting real match peaks. Note that the coarse search acceptance threshold level is not saved and restored with the model. So, if you always want to use your own coarse search acceptance level, you should set it each time after restoring your model and before calling MpatFindModel() . Otherwise, the default coarse search acceptance level is used..
	M_DEFAULT	Determines the coarse search acceptance threshold level automatically.
M_EXTRA_CANDIDATES	integer (default = 0)	Set the number of extra candidates to consider. Normally, the search algorithm considers only a limited number of (best) scores as possible candidates to match when proceeding at the most sub-sampled stage. This parameter allows you to add robustness to the algorithm, by considering more candidates, without compromising too heavily on search speed.

For a result buffer identifier, this parameter can be set to the following value:

Parameter	Values	Description
M_TARGET_CACHING	M_ENABLE	When set to M_ENABLE, the pyramidal representation of the buffer (generated when MpatFindModel() or MpatFindMultipleModel() is called) is kept in the result buffer. 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.
	M_DISABLE	When set to M_DISABLE, the pyramidal representation of the buffer is generated each time MpatFindModel() or MpatFindMultipleModel() is called. Default is M_DISABLE.

See also **MpatSetSpeed()**, **MpatSetAccuracy()**, **MpatInquire()**

MpatSetSpeed

Synopsis Set search speed of a model.

Format **void MpatSetSpeed(ModelId, SpeedFactor)**

MIL_ID ModelId;	Model identifier
long SpeedFactor;	Search speed factor

Description This function specifies the required search speed when using **MpatFindModel()** or **MpatFindMultipleModel()**. At a higher speed, the search takes all reasonable short cuts; therefore, the search should be performed faster than at lower settings. Generally, the high speed setting should be used for better quality images or when using a simple model. Note, the high speed setting reduces positional accuracy very slightly. Try the low speed settings only if your image quality is particularly poor and you have encountered problems at higher speeds. Also, whenever the search speed parameter setting of a model changes, the effect of any preprocessing of the model is undone. Therefore, if the search speed is changed, call **MpatPreprocModel()** before searching for the model.

The **ModelId** parameter specifies the identifier of the model for which to change the speed parameter.

The **SpeedFactor** parameter specifies the search speed. Set this parameter to one of the following:

M_VERY_HIGH	Very high speed
M_HIGH	High speed
M_MEDIUM	Medium speed
M_LOW	Low speed
M_VERY_LOW	Very low speed

Examples mpatrot.c, msearch.c

See also **MpatPreprocModel()**

MpatWrite

Synopsis Write a pattern matching model to an open file.

Format `void MpatWrite(FileHandle, ModelId)`

FILE* FileHandle;	Model file handle
MIL_ID ModelId;	Model identifier

Description This function writes all the information about the previously allocated model to the current file position of an open file, including all of the model's current search parameter values and any effects of preprocessing. This information can later be reloaded with **MpatRead()**.

The **FileHandle** parameter specifies the handle of the open file (opened with the standard C function **fopen()**). The model is written starting at the current file position. After writing, the file remains open and is positioned immediately after the model just written.

The **ModelId** parameter specifies the identifier of the model to save.

See also **MpatRestore()**, **MpatWrite()**, **MpatRead()**

MsysAlloc

Synopsis Allocate a hardware system.

Format **MIL_ID MsysAlloc(SystemType, SystemNum, InitFlag, SystemIdPtr)**

void *SystemType;	Type of system to allocate
long SystemNum;	System number
long InitFlag;	Initialization flag
MIL_ID *SystemIdPtr;	Storage location for system identifier

Description This function allocates a hardware system (board set, Host system, and any available graphics controller) so that it can be used by subsequent MIL functions. Upon execution of this function, MIL ensures that it can open communication with the system before allocating it and generates an error if it cannot.

A system must be allocated before any buffers, displays, or digitizers can be allocated on it. Before allocating a system, an application must be allocated, using **MappAlloc()** or **MappAllocDefault()**.

Note, upon allocation of an application, a default Host system is automatically allocated. Rather than using **MsysAlloc()** to allocate a Host system, you can use this default Host system, by specifying M_DEFAULT_HOST wherever a Host system identifier is required.

When you no longer need a particular system, free it using **MsysFree()**.

The **SystemType** parameter specifies the type of system to allocate. Set this parameter to one of the following values:

SystemType	Type of system to allocate
M_SYSTEM_SETUP	System selected in the setup utility.
M_SYSTEM_HOST	Host type system.
M_SYSTEM_VGA	VGA type system.
M_SYSTEM_METEOR_II	Meteor-II type system.
M_SYSTEM_METEOR_II_1394	Meteor-II /1394 type system.
M_SYSTEM_METEOR_II_DIG	Meteor-II /Digital type system.
M_SYSTEM_METEOR_II_CL	Meteor-II /Camera Link type system.
M_SYSTEM_ORION	Orion type system.

SystemType	Type of system to allocate
M_SYSTEM_GENESIS	Genesis type system.
M_SYSTEM_CORONA_II	Corona-II type system.

The **SystemNum** parameter specifies the number (or rank) of the target board of the specified system type. This parameter can be set to one of the following:

M_DEFAULT	Default board.
M_DEV0	The first board of the specified system type.
...	The n th board of the specified system type.
M_DEV15	The sixteenth board of the specified system type.

The **InitFlag** parameter specifies the type of initialization to perform on the selected system. This parameter can be set to one of the following:

M_COMPLETE	Perform a complete initialization of the system: initialize the system to its default state and download any required resident software. At least one complete initialization is necessary after you power-up your system.
M_PARTIAL	Initialize the system with its default state, but do not download any resident software (which can take a few seconds).
M_DEFAULT	Same as M_COMPLETE.

You can control the use of DirectDraw for all displays allocated for the system. To do so, add one of the following to **InitFlag**:

M_DDRAW	Enable the use of DirectDraw by the system.
M_NO_DDRAW	Disable the use of DirectDraw by the system.

Refer to the *MIL/MIL-Lite Board-Specific Notes* for possible additional information that applies to your particular system.

The **SystemIdPtr** parameter specifies the address of the variable in which to write the system identifier. Since the **MsysAlloc()** function also returns the system identifier, you can set this parameter to M_NULL. If allocation fails, M_NULL is written as the identifier.

Return value The returned value is the system identifier. If allocation fails, M_NULL is returned.

See also **MsysFree()**

MsysControl

Synopsis Control system behavior.

Format void MsysControl(SystemId, ControlType, ControlValue)

MIL_ID SystemId;	System identifier
long ControlType;	Type of event to control
long ControlValue;	Flag to control event

Description This function controls the system behavior. For example, it can be used to control where buffers allocated on the specified system will be processed. Generally, when you allocate buffers on a specific system, processing is done on that system or on the Host system if it is more appropriate. However, you can use this function to force all processing on a specific system.

The **SystemId** parameter specifies the identifier of the system on which to set the control.

The **ControlType** and **ControlValue** parameters specify the type of event to control and the associated value, respectively. These parameters can be set to any valid control type and control value combination that is supported by the system (refer to the appropriate *MIL Board-Specific Notes* chapter), or to one of the following combinations:

ControlType	ControlValue & Description	
M_PROCESSING_SYSTEM	MIL identifier of the system to use for processing, cast to long.	Force the processing of buffers, allocated on the system specified by SystemId , to be performed by the system specified by the control value.
	M_DEFAULT_HOST	Force the processing of buffers, allocated on the system specified by SystemId , to be performed by the default Host system.
	M_DEFAULT	Re-establish the default processing system selected by MIL at system allocation.
	*Note, even when you force processing to be performed by a specific system, some operations might not execute successfully if the specific system does not completely support the requested operation. This can occur even if processing compensation is enabled.	

ControlType	ControlValue & Description	
M_LIVE_GRAB_MOVE_UPDATE	Specifies whether to copy the current image from its previous (window) location to the location of the displaced window before restarting the grab operation (the grab is stopped during window displacement). This is particularly useful when grabbing from a triggered camera, since a trigger is probably not issued as often as the window is displaced. Therefore, the window will be empty after its displacement unless M_LIVE_GRAB_MOVE_UPDATE is enabled.	
	M_ENABLE	Perform a copy between the windows. (Default for triggered cameras)
	M_DISABLE	Do not perform a copy between the windows. (Default for non-triggered cameras)
M_LIVE_GRAB_NO_TEARING	Specifies whether or not no-tearing mode is enabled with live grabs. This mode should be enabled before selecting any buffer to the display. This mode requires special hardware. A Matrox Millennium G400, G450, or G550 graphics controller must be used. If this control type is used and is not supported, an error will be produced.	
	M_ENABLE	No-tearing mode is enabled with live grabs.
	M_DISABLE	No-tearing mode is disabled with live grabs.(default)
M_LAST_GRAB_IN_TRUE_BUFFER	Specifies, for windowed displays (M_WINDOWED), whether a snapshot grab is automatically performed in the true grab buffer at the end of a live grab operation. You can override this default, in which case, the true buffer will not contain the grabbed data. This default can be overridden by setting the ControlType to M_DISABLE:	
	M_ENABLE	Grab last frame in true grab buffer (default).
	M_DISABLE	Don't grab last frame in true grab buffer.
M_NATIVE_MODE_ENTER	M_DEFAULT	Signal to MIL that the system is entering the system's native mode.
M_NATIVE_MODE_LEAVE	M_DEFAULT	Signal to MIL that the system is exiting the system's native mode.

ControlType	ControlValue & Description	
M_USE_MMX	Specifies whether MMX opcodes are used when processing is done on the specified system.	
	M_DEFAULT	Like M_ENABLE when an MMX processor is detected, otherwise like M_DISABLE.
	M_ENABLE	Use the MMX opcodes to accelerate processing.
	M_DISABLE	Never use the MMX opcodes.
M_USE_SSE	Control the use of SSE code when processing is done on the specified system.	
	M_DEFAULT	When an SSE processor is detected, this control type is similar to M_ENABLE; otherwise, it is similar to M_DISABLE.
	M_ENABLE	Use the SSE opcodes to accelerate processing. Note, an error will be generated if no SSE processor is detected or if the operating system does not support it.
	M_DISABLE	Never use the SSE opcodes.
M_USE_SSE2	Control the use of SSE2 code when processing is done on the specified system.	
	M_DEFAULT	M_ENABLE when a SSE2 processor is detected, otherwise M_DISABLE.
	M_DISABLE	Never use SSE2 opcodes.
	M_ENABLE	The SSE2 opcode will be used to accelerate processing. An error will be generated if no SSE2 processor is detected or if the OS does not support it.(One should use M_DEFAULT instead).

ControlType	ControlValue & Description	
If your graphics controller does not have non-destructive overlay capabilities, MIL can typically still grab live into a windowed display. When necessary, the grab will switch to pseudo-live (simulating a live grab by grabbing into the Host buffer and updating the display) to prevent the grab from overwriting another window. If there is an instance when automatic live-to-pseudo-live switching does not happen or you want to override the default behavior, you can use the following ControlType and ControlValue parameter settings.		
M_LIVE_GRAB	Specifies whether to perform a live grab whenever possible, or to force a pseudo-live grab, when grabbing continuously into a displayable buffer.	
	M_ENABLE	Live grab is enabled (default).
	M_DISABLE	Live grab is disabled.
M_PSEUDO_LIVE_GRAB_WHEN_OVERLAPPED	Set whether to pause the grab or switch to a pseudo-live grab when the live grab is interrupted due to one of the above-mentioned conditions (for example, if the window displaying the grab is overlapped by a menu).	
	M_ENABLE	Pseudo-live (default).
	M_DISABLE	Pause grab.
M_STOP_LIVE_GRAB_WHEN_DISABLED	Set whether or not to switch to a pseudo-live grab while the display window is disabled (for example, when a pop-up dialog box is opened):	
	M_ENABLE	Pseudo-live while the display window is disabled (default).
	M_DISABLE	Force live.
M_STOP_LIVE_GRAB_WHEN_INACTIVE	Set whether or not to switch to a pseudo-live grab while the display window is inactive (that is, while it does not have the focus):	
	M_ENABLE	Pseudo-live while the display window is inactive (default).
	M_DISABLE	Force live.
M_STOP_LIVE_GRAB_WHEN_MENU	Set whether or not to switch to a pseudo-live grab while an opened menu overlaps the display window:	
	M_ENABLE	Pseudo-live while menu overlaps the display window (default).
	M_DISABLE	Force live.

ControlType	ControlValue & Description	
M_PSEUDO_LIVE_GRAB	Specifies whether to perform a pseudo-live grab when a live grab is enabled but is not possible. If a live grab is enabled, and can be performed, it will take priority over a pseudo-live continuous grab, even if a pseudo-live grab is enabled. A continuous grab is done pseudo-live only when it is enabled and it is not possible to perform a live grab. If pseudo-live grabbing is disabled and a live grab cannot be performed, a continuous grab will be paused until conditions under which a live grab can be performed are achieved (or the grab times out).	
	M_ENABLE	Pseudo-live grab is enabled (default).
	M_DISABLE	Pseudo-live grab is disabled.
M_USE_FULL_OPTIMIZATION	Control the use of all optimizations available on the specified system.	
	M_DEFAULT	M_ENABLE
	M_ENABLE	All available optimization will be used to accelerate processing. No errors will be reported since only the optimization supported by the system will be activated.
	M_DISABLE	Disable all optimization. Only C++ code will be used (no SIMD optimization).

See also **MappGetError(), MappHookFunction(), MappControl()**

MsysFree

Synopsis Free a system.

Format **void MsysFree(SystemId)**

MIL_ID SystemId;	System identifier
------------------	-------------------

Description This function deallocates a system previously allocated with **MsysAlloc()**.

Prior to freeing a system, ensure that all buffers, displays, and digitizers allocated on the system are freed.

The **SystemId** parameter specifies the identifier of the system to free.

See also **MsysAlloc()**

MsysInquire

Synopsis Inquire about a system parameter setting.

Format **long MsysInquire(SystemId, InquireType, UserVarPtr)**

MIL_ID SystemId;	System identifier
long InquireType;	Type of information to inquire
void *UserVarPtr;	Storage location for inquired information

Description This function inquires about the specified system parameter setting.

The **SystemId** parameter specifies the system identifier.

The **InquireType** parameter specifies the system parameter about which to inquire. Some of the values are not supported by all platforms. This parameter can be set to one of the following values:

InquireType	Description
M_OWNER_APPLICATION	The MIL identifier (MIL_ID) of the application on which the system has been allocated.
M_SYSTEM_TYPE	The type of system allocated: M_SYSTEM_HOST_TYPE, M_SYSTEM_VGA_TYPE, M_SYSTEM_METEOR_II_1394_TYPE, M_SYSTEM_METEOR_II_TYPE, M_SYSTEM_METEOR_II_DIG_TYPE, M_SYSTEM_METEOR_II_CL_TYPE, M_SYSTEM_ORION_TYPE, M_SYSTEM_GENESIS_TYPE, or M_SYSTEM_CORONA_II_TYPE.
M_SYSTEM_NAME	This inquire type copies the system's name to the user-supplied array, as a string. Note that this inquire type is available when using any supported Matrox Imaging board.
M_SYSTEM_TYPE_PTR	Pointer to a function that can communicate with the system (board). This inquiry type returns the actual system type pointer that was passed to the MsysAlloc() function upon system allocation. It is preferable to use M_SYSTEM_TYPE_PTR to inquire about the type of system allocated.
M_NUMBER	Board number of the system (MsysAlloc()).
M_INIT_FLAG	System initialization flag (MsysAlloc()).
M_DISPLAY_NUM	Number of CRT controllers available on your Matrox imaging board.
M_DIGITIZER_NUM	Number of digitizers available on the system.
M_PROCESSOR_NUM	Number of processors available on the system.

InquireType	Description	
M_PROCESSING_SYSTEM_TYPE	Processing system type used to process buffers allocated on that system (MsysControl()). Either M_SYSTEM_HOST_TYPE, or M_SYSTEM_GENESIS_TYPE will be returned.	
M_PROCESSING_SYSTEM	Identifier of the processing system.	
M_LOCATION	Location of the specified system.	
	M_LOCAL	The system is located on the local computer.
	M_REMOTE	The system is located on a remote computer.
M_DCF_SUPPORTED	Whether the system supports downloadable digitizer configuration format (<i>.dcf</i>) files.	
M_USE_MMX	State of use of MMX code for processing on the specified system (M_ENABLE or M_DISABLE).	
M_USE_SSE	State of use of SSE code for processing on the specified system (M_ENABLE or M_DISABLE).	
M_PHYSICAL_ADDRESS_VGA	The physical address of the VGA frame buffer. If the VGA is not a Matrox VGA, M_NULL is returned.	
M_COMPRESSION_SUPPORTED	Whether the system supports compression and decompression of images (M_YES or M_NO). MIL-Lite does not support JPEG 2000 compression, and requires dedicated hardware for JPEG compression. Under the full version of MIL, compression and decompression is supported, whether or not dedicated hardware is present.	
M_LIVE_GRAB	Whether the live grab is enabled (M_ENABLE or M_DISABLE).	
M_PSEUDO_LIVE_GRAB	Whether the pseudo live grab is enabled (M_ENABLE or M_DISABLE).	
M_PSEUDO_LIVE_GRAB_WHEN_OVERLAPPED	A switch is made to a pseudo-live grab when the display window is overlapped by another window: M_ENABLE or M_DISABLE.	
M_STOP_LIVE_GRAB_WHEN_DISABLED	Grabbing is frozen when the display window is disabled: M_ENABLE or M_DISABLE.	
M_STOP_LIVE_GRAB_WHEN_INACTIVE	Grabbing is frozen when the display window is inactive: M_ENABLE or M_DISABLE.	

InquireType	Description
M_STOP_LIVE_GRAB_WHEN_MENU	Grabbing is frozen when the display window is overlapped by a menu: M_ENABLE or M_DISABLE.
M_LIVE_GRAB_NO_TEARING	Whether no-tearing mode is enabled with live grabs.
M_LIVE_GRAB_MOVE_UPDATE	Whether the live grab move update is enabled (M_ENABLE or M_DISABLE).
M_LAST_GRAB_IN_TRUE_BUFFER	A last grab is done to the true buffer at the end of a continuous grab: M_ENABLE or M_DISABLE.

The **UserVarPtr** parameter specifies the address of the variable in which the requested information is to be written. When **MsysInquire()** also returns the requested information, you can set this parameter to M_NULL.

The variable should be a pointer to a long except for the following inquire types:

- M_OWNER_APPLICATION and M_PROCESSING_SYSTEM_TYPE, which should be a pointer to a MIL_ID.
- M_SYSTEM_NAME, which should be a pointer to an array of MIL_TEXT_CHAR. The array of MIL_TEXT_CHAR must be large enough to hold the name of the system.
- M_SYSTEM_TYPE_PTR, which should be a void pointer.

Return value Except for M_SYSTEM_NAME, the returned value is the requested system information, cast to long. For M_SYSTEM_NAME, the returned value is M_NULL.

See also **MsysAlloc()**, **MsysControl()**

Appendices

Appendix A: The default setup configuration file

This appendix discusses the main defaults specified in the setup configuration file.

The default setup configuration file

When you use the **MappAllocDefault()** macro to initialize the global state of the library, open communication channels with any required hardware system, download any required resident software to this hardware, allocate an image buffer, display controller or digitizer, the macro uses the defaults specified in the *milsetup.h* file. This file is set up upon installation with the install utility. It is an ASCII file that can also be modified manually. You should review the contents of this file prior to using the **MappAllocDefault()** macro to ensure that the defaults are as required. You can modify these defaults to a preferred default setup. This appendix discusses each of the main defaults in detail so that you can modify them, if required, by altering their predefined values. For a complete listing of all the defaults, refer to the *milsetup.h* file.

The setup flag

```
/*..... */
/* SETUP SPECIFIED FLAG                      */
/* Activate or deactivate MIL use setup flag  */
#define M_MIL_USE_SETUP      1L
```

The `M_MIL_USE_SETUP` default determines whether *milsetup.h* has already been included. This default should always be set to 1L.

The native mode flag

```
/* ..... */
/* NATIVE MODE PROGRAMMING FLAG              */
/* Activate or deactivate native mode programming */
#define M_MIL_USE_NATIVE      1L
```

The `M_MIL_USE_NATIVE` default determines whether native mode code specific to a system can be used in the MIL application. When this default is set to 1L, MIL assumes that native-mode code may be used and will include associated prototypes and defines.

Default initialization flag

```
/*..... */
/* DEFAULT STATE INITIALIZATION FLAG          */
#define M_SETUP              M_COMPLETE
```

The `M_SETUP` default determines the type of initialization to perform if it is specified by the **MappAllocDefault() InitFlag** parameter. `M_SETUP` can be set to `M_COMPLETE` (initialize MIL and do a complete initialization of the specified system) or `M_PARTIAL` (initialize MIL but don't fully initialize the system). In general, set this parameter to `M_COMPLETE` if initialization time is not critical.

Default system

```

/*..... */
/* DEFAULT SYSTEM SPECIFICATION                      */
#define M_DEF_SYSTEM_TYPE                M_SYSTEM_PULSAR
#define M_DEF_SYSTEM_NUM                 M_DEV0
#define M_DEF_SYSTEM_SETUP              M_DEF_SYSTEM_TYPE

```

The above defaults determine the target system (or board) that will be allocated by **MappAllocDefault()**. The **MappAllocDefault()** macro calls the **MsysAlloc()** command to allocate the target system. `M_DEF_SYSTEM_TYPE` specifies the system type, `M_DEV_SYSTEM_NUM` specifies its device number in your Host system, and `M_DEF_SYSTEM_SETUP` can be used later as an **MsysAlloc()** parameter.

Default display

```

/*..... */
/* DEFAULT DISPLAY SPECIFICATION                      */
#define M_DEF_DISPLAY_NUM                M_DEFAULT
#define M_DEF_DISPLAY_FORMAT             MIL_TEXT("M_DEFAULT")
#define M_DEF_DISPLAY_INIT               M_DEFAULT
#define M_DEF_DISPLAY_SETUP              M_DEF_DISPLAY_FORMAT
#define M_DEF_DISPLAY_KEY_COLOR          0L
#define M_DEF_DISPLAY_KEY_ENABLE_ON_ALLOC 0L
#define M_DEF_DISPLAY_KEY_DISABLE_ON_FREE 0L

```

The above defaults determine the display type that will be allocated if the **MappAllocDefault() DisplayIdVarPtr** parameter is not set to `M_NULL`. **MappAllocDefault()** macro calls the **MdispAlloc()** command to allocate the display. `M_DEF_DISPLAY_NUM` specifies display number on your target system, and `M_DEF_DISPLAY_FORMAT` specifies the display format. `M_DEF_DISPLAY_INIT` should be set to `M_DEFAULT`.

Default digitizer

```

/* ..... */
/* DEFAULT DIGITIZER SPECIFICATION */
#define M_DEF_DIGITIZER_NUM      M_DEVO
#define M_DEF_DIGITIZER_FORMAT  "\\PULSARLIB\\DCF\\R170_LO.DCF"
#define M_DEF_DIGITIZER_INIT     M_DEFAULT
#define M_DEF_CAMERA_SETUP      M_DEF_DIGITIZER_FORMAT

```

The above defaults determine the digitizer type that will be allocated if the **MappAllocDefault()** **DigitizerIdVarPtr** parameter is not set to M_NULL. **MappAllocDefault()** macro calls the **MdigAlloc()** command to allocate the digitizer.

M_DEF_DIGITIZER_NUM specifies digitizer number on your target system, and M_DEF_DIGITIZER_FORMAT specifies the input data format (or camera output data format).

M_DEF_DIGITIZER_INIT should be set to M_DEFAULT.

Default image buffer

```

/* ..... */
/* DEFAULT IMAGE BUFFER SPECIFICATION */
#define M_DEF_IMAGE_NUMBANDS_MIN      1L
#define M_DEF_IMAGE_SIZE_X_MIN        512
#define M_DEF_IMAGE_SIZE_Y_MIN        480
#define M_DEF_IMAGE_SIZE_X_MAX        1280
#define M_DEF_IMAGE_SIZE_Y_MAX        1024
#define M_DEF_IMAGE_TYPE               8+M_UNSIGNED
#define M_DEF_IMAGE_ATTRIBUTE_MIN      M_IMAGE+M_PROC

```

The above defaults determine the image buffer that will be allocated if the **MappAllocDefault()** **ImageIdVarPtr** parameter is not set to M_NULL. By default, if a color digitizer was specified upon installation, a color buffer (three bands) will be allocated; otherwise, a monochrome buffer is allocated. The **MappAllocDefault()** macro calls the **MbufAlloc2d()** command to allocate a monochrome buffer or

MbufAllocColor() to allocate a color buffer. The buffer width and height are the maximum between the default display image dimensions M_DEF_IMAGE_SIZE_X_MIN and M_DEF_IMAGE_SIZE_Y_MIN and the default display format size, but never exceed M_DEF_IMAGE_SIZE_X_MAX and M_DEF_IMAGE_SIZE_Y_MAX. M_DEF_IMAGE_TYPE specifies the depth and range of the data buffer.

M_DEF_IMAGE_ATTRIBUTE_MIN specifies the minimum attributes for the buffer usage.

M_DEF_IMAGE_NUMBANDS_MIN specifies the number of color bands of the buffer.

When you do not want to use defaults

When you do not want to use **MappAllocDefault()**, you can individually specify the allocation of any MIL application, system, digitizer, buffer, or display. The individual allocations must respect the following:

- You must allocate the MIL application before using any other MIL function.
- You must allocate the MIL system after allocating the MIL application and before allocating any digitizer, buffer, or display. You can allocate multiple systems within an application.
- You can allocate multiple digitizers, buffers, or displays within a system.
- When freeing (de-allocating) individually, you must respect the reverse of the order required for allocation.

The following illustrates allocating individually, using a modification of the *mgrab.c* example (appearing in *Chapter 2* of the *User Guide*).

```
/* File name: mgrab.c
 * Synopsis: This program grabs an image from the default camera.
 */
#include <stdio.h>
#include <mil.h>

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

    /* Allocate an application. */
    MappAlloc(M_DEFAULT, &MilApplication);
    /* Allocate a system. */
    MsysAlloc(M_SYSTEM_METEOR-II, M_DEVO, M_COMPLETE, &MilSystem);
```

(cont. ...)

```
/* Allocate a digitizer. */
MdigAlloc(MilSystem, M_DEVO, M_CAMERA_SETUP, M_DEFAULT, &MilCamera);

/* Allocate a display. */
MdispAlloc(MilSystem, M_DEVO, M_DISPLAY_SETUP, M_DEFAULT,
           &MilDisplay);

/* Allocate a buffer. */
MbufAlloc2d(MilSystem, 640, 480, 8, M_IMAGE + M_PROC + M_GRAB + M_DISP,
           &MilImage);

/* Select a display. */
MdispSelect(MilDisplay, MilImage);

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

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

/* Release the buffer. */
MbufFree(MilImage);

/* Release the display. */
MdispFree(MilDisplay);

/* Release the digitizer. */
MdigFree(MilCamera);

/* Release the system. */
MsysFree(MilSystem);

/* Release the application.*/
MappFree(MilApplication);
}
```

Appendix B: The MIL Function Developer's Toolkit

This chapter covers the purpose and contents of the toolkit that provides a privileged interface with MIL.

The MIL Function Developer's Toolkit

The MIL Function Developer's Toolkit provides a privileged interface with MIL that allows MIL programmers to define commands to extend MIL's functionality.

You can create your own MIL-type functions (pseudo-MIL functions) and integrate them directly into the MIL library, where they behave like standard MIL functions (for example, respecting error handling and tracing). This is useful to create high-level packages on top of MIL or to extend the MIL library function set (for example, by adding new functions with specialized algorithms). Although pseudo-MIL functions can also integrate native mode functions, their inclusion makes the pseudo-MIL function non-portable to other platforms. The toolkit provides a series of functions (**Mfunc...()**) designed to facilitate the creation of pseudo-MIL functions.

An example using the Function Developer's Toolkit

In this example, we create a pseudo-MIL function that adds a constant to a LUT buffer and writes the result into the same buffer.

```

/*
 * File name: mnatfct.c
 * Synopsis: This example shows the use of the MIL native mode programmer's kit.
 *           It shows how the user can mix MIL code and
 *           user code to create a pseudo-MIL function.
 *           This example creates a function that finds the pointer
 *           to a MIL image buffer and uses it to directly access
 *           the data and add a constant.
 *           Note: The image must be 8 bit unsigned.
 */

/* headers. */
#include <stdio.h>
#include <mil.h>

#define MAX_DEPTH          8
#define DIMENSION_ERROR   1
#define IMAGE_FILE         M_IMAGE_PATH MT("Board.mim")

```

(cont...)

```

/* Pseudo-MIL Function definition. */
void AddConstToImage(MIL_ID MilImageId, unsigned long ConstantToAdd)
{
    MIL_ID      Func;
    long        ImageSizeX, ImageSizeY, ImagePitch, x, y, TempValue;
    unsigned char *ImageDataPtr;

    /* Prepare the start of the function and register the parameters. */
    Func = MfuncAlloc(MT("AddConstToImage"), 2, M_NULL, M_NULL, M_NULL, M_NULL, M_DEFAULT, M_NULL);
    MfuncParamId(Func, 1, MilImageId, M_IMAGE, M_IN+M_OUT);
    MfuncParamLong(Func, 2, ConstantToAdd);

    /* Mark the start of the function. */
    if (MfuncStart(Func))
    {
        /* Do the operation using a custom function and check to */
        /* not exceed the supported limits if required.          */

        if ((!MfuncParamCheck(Func)) ||
            (MbufInquire(MilImageId, M_SIZE_BIT, M_NULL) == MAX_DEPTH))
        {
            /* Read the image data pointer, size X, size Y and pitch. */
            MbufInquire(MilImageId, M_HOST_ADDRESS, &ImageDataPtr);
            MbufInquire(MilImageId, M_SIZE_X, &ImageSizeX);
            MbufInquire(MilImageId, M_SIZE_Y, &ImageSizeY);
            MbufInquire(MilImageId, M_PITCH, &ImagePitch);

            /* Add the constant to the image. */
            for (y= 0; y < ImageSizeY; y++) /* For each line */
            {
                for (x= 0; x < ImageSizeX; x++) /* for each pixel */
                {
                    /* Calculate the value to write */
                    TempValue = (long)ImageDataPtr[x] + (long)ConstantToAdd;

                    /* Write the value if no overflow else saturate */
                    if (TempValue <= 0xff)
                        ImageDataPtr[x] = (unsigned char)TempValue;
                    else
                        ImageDataPtr[x] = 0xff;
                }

                /* Move pointer to the next line */
                ImageDataPtr += ImagePitch;
            }
        }
        else
        {
            /* Report a MIL error. */
            MfuncErrorReport(Func, M_FUNC_ERROR+DIMENSION_ERROR,
                            MT("Buffer type not supported"),
                            MT("Depth is not 8 bit."),
                            M_NULL,
                            M_NULL
                            );
        }
    }
}

```

(cont. ...)

```

/* Signal to MIL that the image content was modified. */
MbufControl(MilImageId, M_MODIFIED, M_DEFAULT);

/* Mark the end of the function. */
MfuncFreeAndEnd(Func);
}

/* Main, to test the pseudo-MIL function. */
void main(void)
{
    MIL_ID MilApplication,    /* Application Identifier. */
    MilSystem,               /* System Identifier. */
    MilDisplay,              /* Display Identifier. */
    MilImage;                /* Image buffer Identifier. */

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

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

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

    /* Pause */
    printf("This example will create a custom pseudo-MIL function that \n");
    printf("process a MIL buffer using the pointer to its raw data.\n\n");
    printf("Reference image was loaded, press a key to continue.\n");
    getchar();

    /* Process the image directly with custom Pseudo-Mil function */
    AddConstToImage(MilImage, 0x40);

    /* Pause */
    printf("The white level of the image was augmented using a user made\n");
    printf("pseudo-MIL function accessing directly the image.\n");
    printf("Press a key to terminate.\n");
    getchar();

    /* Remove the image from display. */
    MdispDeselect(MilDisplay, MilImage);

    /* Release all allocated buffers. */
    MbufFree(MilImage);

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

```

MIL Function Developer's Toolkit Command Reference

The MIL Function Developer's Toolkit provides functions that allow you to create pseudo-MIL functions. The following table provides an overview of these functions.

MIL developer functions	Command parameters	Description
MfuncAlloc()	FunctionName, ParameterNumber	Allocate a pseudo-MIL function.
MfuncAllocId()	FunctionId, ObjectType, UserPtr	Allocate a pseudo-MIL object (a user-created object associated with a MIL identifier).
MfuncErrorReport()	FunctionId, ErrorCode, ErrorMessage, ErrorSubMessage1, ErrorSubMessage2, ErrorSubMessage3	Report an error message.
MfuncFreeAndEnd()	FunctionId	Free and end a pseudo-MIL function.
MfuncFreeId()	FunctionId, ObjectId	Free the MIL identifier associated with a pseudo-MIL object.
MfuncGetError()	FunctionId, ErrorType, ErrorVarPtr	Get error code or message.
MfuncIdGetObjectType()	FunctionId, ObjectId	Get the object type of a pseudo-MIL object.
MfuncIdGetUserPtr()	FunctionId, ObjectId	Get the user pointer associated with a pseudo-MIL object.
MfuncIdSetObjectType()	FunctionId, ObjectId, ObjectType	Assign a new object type to a pseudo-MIL object.
MfuncIdSetUserPtr()	FunctionId, ObjectId, UserPtr	Assign a new user pointer to a pseudo-MIL object.
MfuncModified()	BufId	Signal the modification of a MIL buffer.
MfuncParamCheck()	FunctionId	Read the MIL application parameter checking flag.
MfuncParamDouble()	FunctionId, ParamIndex, ParamValue	Register a double parameter.
MfuncParamId()	FunctionId, ParamIndex, ParamValue, ParamIs, ParamHasAttr	Register a MIL_ID parameter.

MIL developer functions	Command parameters	Description
MfuncParamLong()	FunctionId, ParamIndex, ParamValue	Register a long parameter.
MfuncParamPointer()	FunctionId, ParamIndex, ParamValue	Register a pointer parameter.
MfuncParamRegister()	FunctionId	Read MIL application parameter registering flag.
MfuncParamString()	FunctionId, ParamIndex, ParamValue	Register a null terminated string parameter.
MfuncStart()	FunctionId	Signal the start of a pseudo-MIL function.

MfuncAlloc

Synopsis Allocate a Pseudo-MIL function.

Format **MIL_ID MfuncAlloc(FunctionName, ParameterNumber, WorkFunctionPtr, WorkDllName, WorkFunctionName, WorkFunctionOpcode, InitFlag, FuncIdVarPtr)**

MIL_TEXT_PTR FunctionName;	Function name
long ParameterNumber;	Number of parameters passed
MFUNCFCTPTR WorkFunctionPtr;	Pointer to the work function to call
MIL_TEXT_PTR WorkDllName;	Name of the module containing the work function
MIL_TEXT_PTR WorkFunctionName;	Name of the work function to call in the module
long WorkFunctionOpcode;	Unique work function opcode
long InitFlag;	Initialization flag
MIL_ID *FuncIdVarPtr;	Pointer to a variable for the MIL identifier

Description This function allows you to associate the current user-created function (that is, the function calling **MfuncAlloc()**) with a MIL identifier and allocate it as a pseudo-MIL function. This function will then be considered as a standard MIL function, respecting all of MIL environment controls, such as tracing and error handling.

You must establish the existence of the pseudo-MIL function (with a call to **MfuncAlloc()**), before calling any other function. You must also register each parameter of this pseudo-MIL function by calling the appropriate **MfuncParam...()** function. Once this has been done, you must signal the actual start of the pseudo-MIL function by calling **MfuncStart()**.

Upon completion, you must signal the end of the pseudo-MIL function by calling **MfuncFreeAndEnd()**.

The **FunctionName** parameter is a null terminated string specifying the name of the current user-created function.

The **ParameterNumber** parameter is the number of parameters passed to the current user-created function.

The **WorkFunctionPtr**, **WorkDllName**, **WorkFunctionName**, **WorkFunctionOpcode**, **FuncIdVarPtr** parameters are reserved for future use, and should therefore be set to **M_NULL**.

The **InitFlag** parameter is reserved for future use, and should therefore be set to **M_DEFAULT**.

Return value The returned value is a MIL identifier for the function; **M_NULL** on error.

Example `mnatfct.c`

See also **MfuncStart()**, **MfuncFreeAndEnd()**, **MfuncParamDouble()**, **MfuncParamId()**, **MfuncParamLong()**, **MfuncParamPointer()**, **MfuncParamString()**

MfuncAllocId

Synopsis Allocate a MIL identifier for a user-created object.

Format **MIL_ID MfuncAllocId(FunctionId, ObjectType, UserPtr)**

MilId FunctionId;	Function identifier
long ObjectType;	Object type
void *UserPtr;	Pointer to the user-created object

Description This function allows you to allocate a MIL identifier and associate it with a user-created object (such as a structure, or an array). The object is then known as a pseudo-MIL object. This permits a user-created object to be recognized by MIL and treated as a standard MIL object, for such procedures as tracing or error handling.

The **FunctionId** parameter is the identifier of the pseudo-MIL function currently in use.

The **ObjectType** parameter identifies the type of MIL object being allocated. This type is a bit encoded value and must be composed of M_USER_OBJECT_1 or M_USER_OBJECT_2 with **one** of the 16 least significant bits set (for example, M_USER_OBJECT_1 + 0x1L). You should use different group types (M_USER_OBJECT_1 or M_USER_OBJECT_2) for objects that are to be used in different MIL modules.

The **UserPtr** parameter specifies the address of the user-created object that is to be associated with a MIL identifier. This object can be a structure, an array, or any other data type.

Return value The returned value is the allocated MIL identifier; M_NULL on error.

See also **MfuncFreeId()**, **MfuncParamId()**, **MfuncIdGetObjectType()**, **MfuncIdSetObjectType()**, **MfuncIdGetUserPtr()**, **MfuncIdSetUserPtr()**

MfuncErrorReport

Synopsis Report an error message.

Format **long MfuncErrorReport(FunctionId, ErrorCode, ErrorMessage, ErrorSubMessage1, ErrorSubMessage2, ErrorSubMessage3)**

MIL_ID FunctionId;	Function identifier
long ErrorCode;	Error code to log
MIL_TEXT_PTR ErrorMessage;	Error message to log
MIL_TEXT_PTR ErrorSubMessage1;	Sub-error message 1 to log
MIL_TEXT_PTR ErrorSubMessage2;	Sub-error message 2 to log
MIL_TEXT_PTR ErrorSubMessage3;	Sub-error message 3 to log

Description This function allows you to log an error message using the MIL error handling mechanism. When this function is called, MIL will treat your error as a normal MIL error. If error reporting is enabled, the error message will be printed and all the information will be logged by the MIL error handler. These errors can be read using the standard MIL error functions (**MappGetError()**).

If you report an error with an error code set to M_NULL, you will reset any pending internal error that a MIL function call, inside your pseudo-MIL function, might have generated. This is useful if you don't wish the MIL error message to be reported. If you don't clear these errors, or you don't report your own error, MIL will detect any pending error when executing **MfuncFreeAndEnd()** and report the error message, prefixed with the name of your pseudo-MIL function.

The **FunctionId** parameter is the MIL identifier of the pseudo-MIL function in use.

The **ErrorCode** parameter is the numeric code assigned to the pseudo-MIL function's group of error messages. It must be M_FUNC_ERROR or greater (M_FUNC_ERROR+offset), so that it does not conflict with MIL specific errors.

The **ErrorMessage** parameter and its sub-messages are null terminated strings specifying the text of your error message. If you do not want to use one of the sub-messages, M_NULL can be passed. The error message, or any sub-error message, must not be longer than M_ERROR_MESSAGE_SIZE characters (including the terminating null character).

Return value The returned value is M_NULL if an error occurred during the error log operation; otherwise, not null.

Example mnatfct.c

MfuncFreeAndEnd

Synopsis Free and end a Pseudo-MIL function.

Format **void MfuncFreeAndEnd(FunctionId)**

MIL_ID FunctionId;	Function identifier
--------------------	---------------------

Description This function signals the end of a pseudo-MIL function, and frees the identifier associated with it. It assumes that a corresponding call to **MfuncStart()** was previously made.

You must call this function to exit the pseudo-MIL function. When **MfuncFreeAndEnd()** is called, MIL will treat your function end as a standard MIL function end. Any pending error within the function will be reported and, if trace reporting is enabled, the trace message will be printed. You can control the trace behavior using the normal MIL trace control function (**MappControl()**).

The **FunctionId** parameter is the MIL identifier of the function to terminate.

See also **MfuncAlloc(), MfuncStart()**

MfuncFreeId

Synopsis Free the MIL identifier associated with a pseudo-MIL object.

Format void MfuncFreeId(FunctionId, ObjectId)

MIL_ID FunctionId;	Function identifier
MIL_ID ObjectId;	Object identifier

Description This function frees a MIL object identifier that was allocated with the **MfuncAllocId()** function.

The **FunctionId** parameter is the MIL identifier of the pseudo-MIL function in use.

The **ObjectId** parameter is the MIL identifier of the pseudo-MIL object to free.

See also MfuncAllocId()

MfuncGetError

Synopsis Get error code or message.

Format **long MfuncGetError(FunctionId, ErrorType, ErrorVarPtr)**

MIL_ID FunctionId;	Function identifier
long ErrorType;	Error type
void *ErrorVarPtr;	Pointer to a variable for the error

Description This function allows you to read an error code or message that was previously reported. This function can be used to check the success of a MIL function call inside a pseudo-MIL function.

The **FunctionId** parameter is the MIL identifier of the pseudo-MIL function in use.

The **ErrorType** parameter identifies the type of error you want to read. It must be set to one of the following:

M_INTERNAL	Error code returned by the last call to any MIL function. This code is reset to M_NULL_ERROR before each MIL function call and is set to a specific error code if an error occurs while executing the function. The error code is written in the location pointed to by ErrorVarPtr (when not M_NULL) as a long value and is also returned by MfuncGetError() .
M_INTERNAL_SUB_NB	Returns the number of error subcodes associated to the internal error. The number is written in the location pointed to by ErrorVarPtr (when not M_NULL) as a long value and is also returned by MfuncGetError() .
M_INTERNAL_SUB_1, ... M_INTERNAL_SUB_3	The nth error subcode associated to the current error. The error subcode is written in the location pointed to by ErrorVarPtr (when not M_NULL) as a long value and is also returned by MfuncGetError() .
M_INTERNAL_FCT	The function code associated to the current error. The function code is written in the location pointed to by ErrorVarPtr (when ErrorVarPtr is not M_NULL) as a long value and is also returned by MfuncGetError() .
M_INTERNAL_...+ M_MESSAGE	When M_MESSAGE is added to an M_INTERNAL... define, the function will return the string associated with specified error type. The string will be written in a character array pointed to by ErrorVarPtr . This array must be at least M_ERROR_MESSAGE_SIZE characters in size.

The **ErrorVarPtr** parameter is the address of the variable containing the error code or message.

To get the M_GLOBAL or M_CURRENT error, use the regular **MappGetError()** function.

Return value The returned value is an error code or sub-error code; otherwise, M_NULL.

MfuncIdGetObjectType

Synopsis Get the object type of a pseudo-MIL object.

Format **long MfuncIdGetObjectType(FunctionId, ObjectId)**

MIL_ID FunctionId;	Function identifier
MIL_ID ObjectId;	Object identifier

Description This function retrieves the object type of an object that was allocated with the **MfuncAllocId()** function.

The **FunctionId** parameter is the MIL identifier of the pseudo-MIL function in use.

The **ObjectId** parameter is the MIL identifier of the pseudo-MIL object.

Return value The returned value is the object type of the specified object. When the MIL_ID is not valid, M_NULL is returned if the Id value is less than the greater valid Id; M_INVALID if the Id value is greater than the greater valid Id.

See also **MfuncAllocId(), MfuncIdSetObjectType()**

MfuncIdGetUserPtr

Synopsis Get the user pointer of a pseudo-MIL object.

Format `void* MfuncIdGetUserPtr(FunctionId, ObjectId)`

MIL_ID FunctionId;	Function identifier
MIL_ID ObjectId;	Object identifier

Description This function obtains the user pointer of an object that was allocated with the **MfuncAllocId()** function.

The **FunctionId** parameter is the MIL identifier of the pseudo-MIL function in use.

The **ObjectId** parameter is the MIL identifier of the pseudo-MIL object.

Return value The returned value is the user pointer of the specified object.

See also **MfuncAllocId(), MfuncIdSetUserPtr()**

MfuncIdSetObjectType

Synopsis Assign a new object type to a pseudo-MIL object.

Format **void MfuncIdSetObjectType(FunctionId, ObjectId, ObjectType)**

MIL_ID FunctionId;	Function identifier
MIL_ID ObjectId;	Object identifier
long ObjectType;	New object type

Description This function assigns a new object type to an object that was allocated with the **MfuncAllocId()** function.

The **FunctionId** parameter is the MIL identifier of the pseudo-MIL function in use.

The **ObjectId** parameter is the MIL identifier of the pseudo-MIL object.

The **ObjectType** parameter is the new object type to be assigned to the specified object. This type is a bit encoded value and must be composed of M_USER_OBJECT_1 or M_USER_OBJECT_2 with **one** of the 16 least significant bits set (for example, M_USER_OBJECT_1 + 0x1L).

See also **MfuncAllocId(), MfuncIdGetObjectType()**

MfuncIdSetUserPtr

Synopsis Assign a new pointer to a pseudo-MIL object.

Format **void MfuncIdSetUserPtr(FunctionId, ObjectId, UserPtr)**

MIL_ID FunctionId;	Function identifier
MIL_ID ObjectId;	Object identifier
Void *UserPtr;	New user pointer

Description This function assigns a new user pointer to an object that was allocated with the **MfuncAllocId()** function.

The **FunctionId** parameter is the MIL identifier of the pseudo-MIL function in use.

The **ObjectId** parameter is the MIL identifier of the pseudo-MIL object.

The **UserPtr** parameter is the new user pointer to assign to the specified object.

See also **MfuncAllocId()**, **MfuncIdGetUserPtr()**

MfuncModified

Synopsis

Signal the modification of a MIL buffer.

Format

long MfuncModified(BufId)

MIL_ID BufId;	Buffer identifier
---------------	-------------------

Description

This function must be used to signal to MIL that the identified buffer has been modified (altered). MIL will then increment the modification count of that MIL buffer. This count is used by some MIL functions to manage automatic updates. The current value of the count is accessible via **MbufInquire()**.

The **BufId** parameter is the MIL identifier of the buffer that has been modified.

Return value

The returned value is M_NULL if successful; otherwise, an error was found.

MfuncParamCheck

Synopsis Read the MIL application parameter checking flag.

Format `long MfuncParamCheck(FunctionId)`

MIL_ID FunctionId;	Function identifier
--------------------	---------------------

Description This function allows you to read the MIL application parameter checking flag, which has been set with the **MappControl()** function. The **MfuncParamCheck()** function can be used to determine if the parameters of the specified pseudo-MIL function must be checked. This is typically used when you want to save the parameter checking time for a time-critical pseudo-MIL function.

The **FunctionId** parameter is the identifier of the pseudo-MIL function in use.

Return value The returned value is M_NULL if no parameter checking is required; otherwise, checking is required.

Example mnatfct.c

See also **MappControl()**

MfuncParamDouble

Synopsis Register a double parameter.

Format **void MfuncParamDouble(FunctionId, ParamIndex, ParamValue)**

MIL_ID FunctionId;	Function identifier
long ParamIndex;	Parameter index
double ParamValue;	Parameter value

Description This function allows you to register a double parameter of the current pseudo-MIL function. The **MfuncParamDouble()** function should be called after a call to **MfuncAlloc()** and before a call to **MfuncStart()**.

The **FunctionId** parameter is the identifier of the current pseudo-MIL function.

The **ParamIndex** parameter is the index of the parameter within the function parameter list. The index of the first parameter is considered to be one.

The **ParamValue** parameter is the value of the double parameter.

See also **MfuncAlloc(), MfuncStart(), MfuncFreeAndEnd(), MfuncParamCheck(), MfuncParamId(), MfuncParamLong(), MfuncParamPointer(), MfuncParamString()**

MfuncParamId

Synopsis Register a MIL_ID parameter.

Format `void MfuncParamId(FunctionId, ParamIndex, ParamValue, ParamIs, ParamHasAttr)`

MIL_ID FunctionId;	Function identifier
long ParamIndex;	Parameter index
MIL_ID ParamValue;	Parameter value
long ParamIs;	Type of MIL object represented
long ParamHasAttr;	Attribute the MIL object must have

Description This function allows you to register a MIL_ID parameter of the specified pseudo-MIL function. The **MfuncParamId()** function should be called after a call to **MfuncAlloc()** and before a call to **MfuncStart()**.

The **FunctionId** parameter is the identifier of the pseudo-MIL function that received the parameter.

The **ParamIndex** parameter is the index of the parameter within the function parameter list. The index of the first parameter is considered to be one.

The **ParamValue** parameter is the value of the MIL_ID parameter.

The **ParamIs** parameter specifies the type of MIL object. It must be one, or more, of the following types to be considered valid:

M_IMAGE	M_LUT	M_STRUCT_ELEMENT
M_KERNEL	M_HIST_LIST	M_PROJ_LIST
M_EVENT_LIST	M_COUNT_LIST	M_EXTREME_LIST
M_DISPLAY	M_DIGITIZER	M_ARRAY
M_APPLICATION	M_SYSTEM	M_GRAPHIC_CONTEXT
M_BLOB_RESULT	M_BLOB_FEATURE_LIST	M_PAT_MODEL
M_PAT_RESULT	M_OCR_FONT	M_OCR_RESULT
M_MEAS_MARKER	M_MEAS_RESULT	M_MEAS_CONTEXT
M_USER_OBJECT_1	M_USER_OBJECT_2	

The **ParamHasAttr** parameter specifies what kind of attribute the MIL object must have, in order to be considered a valid MIL_ID parameter for the specified function. Either M_IN or M_OUT (or both) **must** be specified, to indicate if the buffer is used for input or output. Optionally, you **can** specify one or more additional attributes from the following list: M_GRAPH, M_DISP, M_GRAB, M_PROC.

Note that the arguments tagged as M_OUT will have their internal modification count incremented to signal that they have been modified.

Example `mnatfct.c`

See also `MfuncAlloc()`, `MfuncStart()`, `MfuncFreeAndEnd()`, `MfuncParamDouble()`, `MfuncParamLong()`, `MfuncParamPointer()`, `MfuncParamString()`

MfuncParamLong

Synopsis Register a long parameter.

Format `void MfuncParamLong(FunctionId, ParamIndex, ParamValue)`

MIL_ID FunctionId;	Function identifier
long ParamIndex;	Parameter index
long ParamValue;	Parameter value

Description This function allows you to register a long parameter of the current pseudo-MIL function. The **MfuncParamLong()** function should be called after a call to **MfuncAlloc()** and before a call to **MfuncStart()**.

The **FunctionId** parameter is the identifier of the current pseudo-MIL function.

The **ParamIndex** parameter is the index of the parameter within the function parameter list. The index of the first parameter is considered to be one.

The **ParamValue** parameter is the value of the long parameter.

See also **MfuncAlloc()**, **MfuncStart()**, **MfuncFreeAndEnd()**, **MfuncParamDouble()**, **MfuncParamId()**, **MfuncParamPointer()**, **MfuncParamString()**

MfuncParamPointer

Synopsis Register a pointer parameter.

Format **void MfuncParamPointer(FunctionId, ParamIndex, *ParamValue, Size, ParamAttribute)**

MIL_ID FunctionId;	Function identifier
long ParamIndex;	Parameter index
void *ParamValue;	Parameter value
long Size;	Size of data pointed to by the ParamValue parameter
long ParamAttribute;	Parameter Attribute

Description This function allows you to register a pointer parameter of the current pseudo-MIL function. The **MfuncParamPointer()** function should be called after a call to **MfuncAlloc()** and before a call to **MfuncStart()**.

The **FunctionId** parameter is the identifier of the current pseudo-MIL function.

The **ParamIndex** parameter is the index of the parameter within the pseudo-MIL function's parameter list. The index of the first parameter is considered to be one.

The **ParamValue** parameter is the value of the pointer parameter.

The **Size** parameter is the size of the data, in bytes, pointed to by the **ParamValue** parameter.

The **ParamAttribute** parameter is the attribute of the pointer parameter. It can be equal to M_IN or M_OUT.

See also **MfuncParamDouble(), MfuncParamId(), MfuncParamLong(), MfuncParamString()**

MfuncParamRegister

Synopsis Read the MIL application parameter registering flag.

Format `long MfuncParamRegister(FunctionId)`

MIL_ID FunctionId;	Function identifier
--------------------	---------------------

Description This function allows you to read the MIL application parameter registering flag. This function can be used to know if the parameters of the specified pseudo-MIL function must be registered. This is typically used when you want to save the parameter registration time for some time-critical pseudo-MIL functions.

The **FunctionId** parameter is the identifier of the pseudo-MIL function in use.

Return value The returned value is M_NULL if no parameter registering is required; otherwise, registering is required.

MfuncParamString

Synopsis Register a null terminated string parameter.

Format `void MfuncParamString(FunctionId, ParamIndex,
*ParamValue,SizeOfData, ParamAttribute)`

MIL_ID FunctionId;	Function identifier
long ParamIndex;	Parameter index
MIL_TEXT_PTR ParamValue;	Parameter value
long Size;	Size of data being pointed to by the ParamValue parameter
long ParamAttribute;	Parameter attribute

Description This function allows you to register a null terminated string parameter of the current pseudo-MIL function. The **MfuncParamString()** function should be called after a call to **MfuncAlloc()** and before a call to **MfuncStart()**.

The **FunctionId** parameter is the identifier of the current pseudo-MIL function.

The **ParamIndex** parameter is the index of the parameter within the pseudo-MIL function's parameter list. The index of the first parameter is considered to be one.

The **ParamValue** parameter is the value of the string parameter.

The **Size** parameter is the size of the data, in bytes, pointed to by the **ParamValue** parameter.

The **ParamAttribute** parameter is the attribute of the string parameter. It can be equal to M_IN or M_OUT.

See also **MfuncParamDouble()**, **MfuncParamId()**, **MfuncParamLong()**, **MfuncParamPointer()**

MfuncStart

Synopsis Signal the start of a pseudo-MIL function.

Format **long MfuncStart(FunctionId)**

MIL_ID FunctionId;	Function identifier
--------------------	---------------------

Description This function signals to MIL the actual start of the specified pseudo-MIL function. When this function is called, MIL will treat your function start as a standard MIL function start. If trace reporting is enabled, the trace message will be printed. You can control the trace behavior using the normal MIL trace function (**MappControl()**).

Note that if a MIL identifier was registered in the function parameter list with **MfuncParamId()**, the validity of that identifier will be checked during **MfuncStart()** execution, and a MIL error will be reported if that identifier is not valid.

The **FunctionId** parameter is the MIL identifier of the pseudo-MIL function to start.

Return value The returned value is M_NULL if an error occurred; otherwise, not null.

Example mnatfct.c

See also **MfuncAlloc()**, **MfuncFreeAndEnd()**, **MfuncParamId()**, **MappControl()**

Index

A

- absolute value
 - image 325
 - result of operation 130
- acquisition
 - attribute 100, 104, 108, 144, 148
 - continuous 257
 - image 256
- adding, image 325, 328
- alignment 28
- allocate
 - application 35
 - buffers 11
 - child buffer 116, 118, 120–121
 - code object 223
 - defaults 10, 37
 - digitizer 11, 240
 - display 11, 270
 - feature list 60
 - graphics context 303
 - measurement context 397
 - measurement marker 399
 - measurement result buffer 402
 - multi-band buffer 108
 - OCR result buffer 472
 - one-dimensional buffer 100
 - pattern matching model, automatic 499
 - pattern matching model, manual 503
 - pattern matching result buffer 507
 - processing result buffer 323
 - pseudo-MIL function 575
 - pseudo-mil identifier 577
 - result buffer 61
 - system 9, 550
 - thread 42
 - two-dimensional buffer 104
- analog reference levels 268
- angle
 - marker 410
 - measurement marker 404
 - search 535

- application
 - allocate 35
 - control environment 40
 - control module 13
 - free 45
 - inquire environment 56
 - pseudo-MIL, parameter checking flag 589
 - pseudo-MIL, parameter registering flag 595
 - starting 9
- arcs, draw 305
- arcs, draw filled 306
- area 90
- arithmetic operations 325, 328
- attributes, data buffer usage 100, 104, 108, 144, 148
- avi files 154, 158, 178

B

- background color
 - associate to graphics context 307
 - inquire 317
- binarize 330
- bit-shift image 383
- blanking, display 279
- blob analysis
 - calculate 63
 - controls 65
 - feature list 60
 - free result buffer 72
 - inquire 81
 - module 14
 - processing mode 65
 - result buffer 61
 - results 75, 77
 - run-length encoding 79
 - selecting blobs 87
- blob chains 66, 71, 91, 96
- blob identifier image
 - labelled 82
- blobs
 - area 90
 - border-touching 83
 - box coordinates 90
 - breadth 90
 - center of gravity 90
 - compactness 90
 - contact points 90

- control 65
- convex perimeter 90
- elongation 90
- Euler number 90
- Feret diameter 90, 97
- fill 70
- intercepts 90
- label 82
- label value 73, 90
- labelling 363
- length 90
- maximum pixel value 90
- mean pixel value 90
- minimum pixel value 90
- number of 74
- number of holes 90
- perimeter 90
- pixel sum 90
- reconstruction 83
- roughness 90
- runs 90
- secondary angle 90
- select 87
- select feature 90
- select moment 98
- standard deviation 90
- sum of squares 90
- thickening 384
- thinning 385
- zone of influence 395
- borders
 - blobs touching 83

C

- calculate
 - blobs 63
 - measurements 403
- camera
 - specification 240
- cell size of code 226
- characters, text 22, 322
- child buffers
 - allocate 116, 118, 120–121
 - ancestor buffer 182
 - attributes 116, 118, 120–121
 - multiple dimensions 116, 118, 121
 - offset 116, 118, 120–121
 - parent buffer 182
 - physical space 116, 118, 120–121
 - purpose 11
 - returned coordinates 116, 118, 120–121
 - type 116, 118, 120–121
- circles, draw 305
- clear
 - buffer 123, 308
 - display 279
- clipping
 - data 135
 - point-to-point 332
- closing operation 334
- code object, allocating 223
- code results, obtaining 230
- codes
 - allocating code object 223
 - inquiring about code objects 232
 - reading 234
 - setting code objects 225
 - types 223
 - writing 19, 236–238
- coefficients, warping 298
- color band 108, 144, 148
- color images
 - allocate buffer 108
 - allocate child buffer 116, 118
 - color conversion 337
 - copy 136, 138
 - create buffer 148
 - displaying 292–293
 - grabbing 256–257
 - loading 188, 200
 - saving 188, 200
- column profile 376
- command reference
 - order 32
 - quick reference 13
 - status section 32
- commands
 - Function DevelopersToolkit' 573
 - MIL, command summary 13
 - predefined constants 32
 - pseudo-MIL 570
- communication channels 9
- compactness, blob 90
- comparative operations 325, 328

- compensation
 - memory 41
 - processing 41
- complex operations 367
- connectivity
 - code 335
 - mapping 335
- contact points 90
- continuous grab 257
- contrast
 - marker/background 428
- contrast variation, markers 428
- control
 - application environment settings 40
 - blobs 65
 - buffer features 124
 - code object 225
 - digitizer 243
 - display 274
 - graphic context 310
 - measurement context 405
 - messages, error 40, 42
 - neighborhood operation 130
 - parameter checking 41
 - processing compensation 41
 - system processing 552
 - thread 42
 - timer 59
 - trace mechanism 40
- conversion
 - color 337
 - data format 155, 174
- convex perimeter 90
- coordinates
 - measurement marker 410
- copy
 - clip, and 135
 - color band 136, 138
 - conditional 141
 - data 134, 136, 138, 141, 161–162, 164, 168, 189–190, 192, 196–197
 - data line 194
 - fonts, OCR 477
 - mask 143
 - model 511

D

- data allocation and access module 15
- data buffer
 - allocation 100, 104, 108, 144, 148
 - ancestor 182
 - attributes 100, 104, 108, 144, 148
 - clear 123, 308
 - clip border 135
 - color band 108, 144, 148
 - control 124
 - copy 134
 - copy color 136, 138
 - copy theoretical line 168
 - defined 11
 - depth 100, 104, 108, 144, 148
 - export data 155
 - free 160
 - import data 174
 - inquire 153, 181, 362
 - load 155
 - multiple dimensions 108, 148
 - parent 182
 - pseudo-MIL, modification 588
 - put data 189–190, 192, 196–197
 - range 100, 104, 108, 144, 148
 - restore 198
 - retrieve data 161–162, 164, 170–171
 - save 155, 200
 - sign 100, 104, 108, 144, 148
 - two-dimensional 144
 - type 108, 144, 148
 - write data 194
- data format, input device 240
- data generation
 - LUT 297
 - module 21
- data objects, manipulation concepts 10
- debugging 12
- defaults
 - application 37
 - display 565
 - free 46
 - image buffer 566
 - initialization flag 564
 - input device 566
 - setup 564
 - system 565

- differences
 - image, count 342
 - image, find 325, 328
- digitizer
 - allocate 240
 - control 243
 - data format 240
 - event hook 260
 - free 255
 - input channel 242
 - inquire 263
 - LUT 267
 - number 240
 - reference levels 268
- dilation
 - advanced 368
 - basic 343
 - closing operation 334
 - opening operation 370
- display
 - allocation 270
 - control behavior 274
 - control module 20
 - format 270
 - free 280
 - image buffer 292–293
 - inquire 283
 - LUT 287
 - number 270
 - pan 291
 - scroll 291
 - zoom 294
- distance
 - inter-marker 404
- distance transformation 344
- dividing, image 325, 328
- don't care pixels 539
- dots, draw 312
- DrawDIBDraw()
 - VGA 271–272

E

- edge
 - detection 346
 - rising/falling 247–248
- edge extractors
 - Laplacian 339

- elongation, blob 90
- encoding strings 19, 236–238
- encoding type 226
- erosion
 - advanced 368
 - basic 349
 - closing operation 334
 - opening operation 370
- error correction type 227
- error reporting
 - appendix 11
 - code 47
 - hook 53, 281
 - message control 40, 42
 - messages 12, 47
 - pseudo-MIL function 578, 582
 - screen 11
 - suberror code 47
 - use 11
- Euler number 90
- event, locate 365
- examples
 - Function DeveloppersToolkit' 570
 - pseudo-function development 570
- export data buffer 155
- extreme value, find 350

F

- feature list, blob analysis 60
- Feret diameter
 - blob feature 90, 97
 - elongation 90
 - mean 90
 - minimum/maximum 90
 - minimum/maximum angle 90
 - number of 65
- field grabbing 247
- file format 155
- files
 - avi 154, 158, 178
 - semi.txt 487
- fill
 - blobs 70
- filled-in shapes 306
 - boundary-type seed fill 313
- filter
 - rank 377

- find
 - buffer extremes 350
 - marker 409
 - model 515
 - multiple models 517
 - orientation, image/object 519
- first-order polynomial warpings 298, 390
- floating-point buffers
 - edge detection restrictions 348
- font
 - associate to graphics context 314
 - inquire 317
 - scale 315
 - size 315
- foreground color
 - associate to graphics context 22, 309
 - inquire 317
- foreground, blobs 65
- free
 - application 45
 - application defaults 46
 - buffer, blob analysis result 72
 - buffer, data 160
 - buffer, measurement result 412
 - buffer, OCR 479
 - buffer, pattern matching 521
 - code object 229
 - defaults 10, 46
 - digitizer 255
 - display 280
 - graphics context 22, 316
 - image processing result buffer 353
 - marker 412
 - measurement context 412
 - model 521
 - pseudo-MIL function 580
 - pseudo-MIL identifier 581
 - system 557
- function
 - development 570
 - hook 12, 53, 260, 281
 - pseudo-MIL, allocate 575
 - pseudo-MIL, example 570
 - pseudo-MIL, free 580
 - pseudo-MIL, start 597
- Function Developers Toolkit 569

- Function DevelopersToolkit
 - command summary 573
 - example 570
- functions See also, commands 32

G

- Genesis
 - system 551
- geometric model finder
 - score 456
 - target score 456
- geometric transforms 22
- get
 - blob label value 73
 - number of blobs 74
 - number of model occurrences 522
 - result, blob analysis 75, 77
 - result, code read/write 230
 - result, image processing 354, 356
 - result, pattern recognition 523
- global library state 564
- grab
 - continuous 257
 - halt 259
 - image 256
 - scale 244
 - wait 258
- gradient, intensity or angle 346
- graphics
 - arcs, draw 305
 - boundary type seed fill 313
 - buffer, clear 22, 308
 - circles, draw 305
 - dots, draw 312
 - filled elliptic arcs, draw 306
 - filled rectangles, draw 22, 321
 - lines, draw 22, 319
 - rectangles, draw 320
 - text, write 22, 322
- graphics context
 - allocate 303
 - background color, associate 307
 - control 310
 - default 304
 - font scale, associate 315

- foreground color, associate 22, 309
- free 22, 316
- inquire 317
- text font, associate 314

H

- halt grabbing 259
- histogram
 - equalization 359
 - generate 358
- hit or miss pattern matching 368
- holes
 - extract 83
 - fill 83
- hook
 - digitizer event 260
 - error 53, 281
 - get information 50
 - to an event 53
 - to OCR event 484
 - trace 53, 281
 - user-defined function 12
- Host
 - communication 9
 - screen 11
- host
 - default system 35, 38, 60–61, 100, 104, 108, 176, 303, 323, 397, 399, 402, 423, 500, 504, 507–508, 530–531, 550
 - system 297, 550
- hue 119, 136, 139, 163, 165, 191, 193, 337
- HLS 116

I

- identifier, MIL objects 32
- image
 - differences, count 342
 - projection 376
 - resizing 379
 - rotation 381
- image buffer
 - clear 22, 123, 308
 - removing from display 279
 - select for display 292
 - select window for display 293

- image processing
 - module 22
 - result buffer 323
- import data 174
- import font, OCR 486
- initialization
 - default 10, 37
 - system 9
- input device
 - control module 19
 - reference level 265
- input signal 245
- inquire
 - application environment 56
 - blob analysis 81
 - code object 232
 - data buffer 153, 181, 362
 - digitizer 263
 - display 283
 - font, OCR 488
 - graphics context 317
 - marker 419
 - measurement context 419
 - measurement result buffer 419
 - pattern recognition model 525
 - system 172, 558
- inspection, using pattern matching 28
- intensity
 - HLS 337
- intercepts 90
- interpolation mode 352, 379, 382, 509

K

- kernels
 - usage 339
- keying
 - inquire 283

L

- labelling
 - blobs 82, 363
- Laplacian edge detection 339
- lattice, blobs 65
- length
 - blob 90
 - inter-marker 404

- line equation
 - between markers 404
- lines
 - draw 22, 319
- load
 - data 188
- logical operations 325
- luminance
 - HLS 116, 119, 136, 139, 163, 165, 191, 193, 337
- LUTs
 - data generation 295, 297
 - display 287
 - input 267
 - mapping, point-to-point 367
 - warping 390

M

- macros 33
- Mapp...() 13, 35
- MappAlloc() 9, 35
- MappAllocDefault() 10, 37, 564
 - example 570
- MappControl() 11–12, 40
- MappControlThread() 42
- MappFree() 9, 45
- MappFreeDefault() 10, 46
 - example 570
- MappGetError() 12, 47
- MappGetHookInfo() 50
- MappHookFunction() 12, 53
- mapping 367
- MappInquire() 56
- MappModfiy() 58
- MappTimer() 59
- marker
 - allocation 399
 - characteristics 425
 - contrast 428
 - contrast variation 428
 - default characteristics 400
 - find 409
 - free 412
 - inquire 419
 - measurement box 428
 - measuring with 403, 409
 - orientation 430

- parameter, set 425
- polarity 430
- restore 423
- save 424
- stripe tolerance 432
- stripe width 432
- mask, copy 143
- match, morphological 368
- matrix-defined warpings
 - generating LUTs for 299
- maximum
 - blob label value 81
 - pixel value 90
- Mblob...() 14
- MblobAllocFeatureList() 60
- MblobAllocResult() 61
- MblobCalculate() 63
- MblobControl() 65
- MblobDraw() 67
- MblobFill() 66, 70
- MblobFree() 72
- MblobGetLabel() 66, 73
- MblobGetNumber() 74
- MblobGetResult() 75
- MblobGetResultSingle() 77
- MblobGetRuns() 66, 79
- MblobInquire() 81
- MblobLabel() 66, 82
- MblobReconstruct() 83
- MblobSelect() 87
- MblobSelectFeature() 90
- MblobSelectFeret() 97
- MblobSelectMoment() 98
- Mbuf...() 15
- MbufAlloc1d() 11, 100
 - example 570
- MbufAlloc2d() 11, 104
- MbufAllocColor() 11, 108
- MbufBayer() 113
- MbufChild1d() 120
- MbufChild2d() 121
- MbufChildColor() 116
- MbufChildColor2d 118
- MbufClear() 123
- MbufControl() 124
- MbufControlNeighborhood() 130
- MbufCopy() 134
- MbufCopyClip() 135

MbufCopyColor()	136	McalTransformImage()	204, 208, 210, 219, 222
MbufCopyColor2d()	138	McalTransformResult()	216, 221
MbufCopyCond()	141	McodeAlloc()	223
MbufCopyMask()	143	McodeControl()	223, 225
MbufCreate2d()	144	McodeFree()	229
MbufCreateColor()	148	McodeGetResult()	230
MbufDiskInquire()	153	McodeInquire()	232
MbufExport()	155	McodeRead()	234
MbufExportSequence()	158	McodeRestore()	236
MbufFree()	160	McodeSave()	237
example	570	McodeWrite()	19, 236–238
MbufGet()	161	Mdig...()	19
example	570	MdigAlloc()	240
MbufGet1d()	170	MdigChannel()	242
MbufGet2d()	171	MdigControl()	243
MbufGetColor()	162	MdigFocus()	252
MbufGetColor2d()	164	MdigFree()	11, 255
MbufGetHookInfo()	166	MdigGrab()	256
MbufGetLine()	168	MdigGrabContinuous()	257
MbufHookFunction	172	MdigGrabWait()	258
MbufHookFunction()	172	MdigHalt()	259
MbufImport()	174	MdigHookFunction()	260
MbufImportSequence()	178	MdigInquire()	263
MbufInquire()	181	MdigLut()	267
example	570	MdigReference()	268
MbufLoad()	188	Mdisp...()	20
example	570	MdispAlloc()	270
MbufPut()	189	MdispControl()	274
example	570	MdispDeselect()	279
MbufPut1d()	196	MdispFree()	11, 280
MbufPut2d()	197	MdispHookFunction()	281
MbufPutColor()	190	MdispInquire()	283
MbufPutColor2d()	192	MdispLut()	287
MbufPutLine()	194	MdispOverlayKey()	289
MbufRestore()	198	MdispPan()	291
MbufSave()	200	MdispSelect()	292
McalAlloc()	201	MdispSelectWindow()	293
McalAssociate()	202	MdispZoom()	294
McalControl()	202, 204	mean pixel value	90
McalFree()	205	measurement box	428
McalGrid()	201, 204, 206, 209	measurement context	
McalInquire()	208	allocate	397
McalList()	201, 209, 211	control parameter	405
McalRelativeOrigin()	213	default values	398
McalRestore()	214	free	412
McalSave()	215	inquire	419
McalTransformCoordinate()	216, 222		
McalTransformCoordinateList()	217		

- measurements
 - angle 404, 410
 - distance, inter-marker 404
 - line equation 404
 - markers, using 403
 - module 25
 - pattern matching 28
 - position 410–411
 - position variation 410–411
 - result buffer 402
 - result buffer, free 412
 - result buffer, inquire 419
 - results 413, 418
 - width variation, stripe 411
 - width, stripe 410
- memory
 - compensation 41
- Meteor
 - system 550
- MfuncAlloc() 575, 590–591, 593–594
 - example 570
- MfuncAllocId() 577, 584, 586–587
- MfuncErrorReport() 578
 - example 570
- MfuncFreeAndEnd() 578, 580
 - example 570
- MfuncFreeId() 581
- MfuncGetError() 582
- MfuncIdGetObjectType() 584
- MfuncIdGetUserPtr() 585
- MfuncIdSetObjectType() 586
- MfuncIdSetUserPtr() 587
- MfuncModified() 588
- MfuncParamChar()
 - example 570
- MfuncParamCheck() 589
 - example 570
- MfuncParamDouble() 590
- MfuncParamId() 591, 597
 - example 570
- MfuncParamLong() 593
- MfuncParamPointer() 594
- MfuncParamRegister() 595
- MfuncParamString() 596
- MfuncStart() 575, 580, 590–591, 593–594, 596–597
 - example 570
- Mgen...() 21
- MgenLutFunction() 295
- MgenLutRamp() 297
 - example 570
- MgenWarpParameter() 298, 390
- Mgra...() 21
- MgraAlloc() 303
- MgraArc() 305
- MgraArcFill() 306
- mgrab.c 567
- MgraBackColor() 307
- MgraClear() 308
- MgraColor() 309
- MgraControl() 310
- MgraDot() 312
- MgraFill() 313
- MgraFont() 314
- MgraFontScale() 315
- MgraFree() 316
- MgraInquire() 317
- MgraLine() 319
- MgraRect() 320
- MgraRectFill() 321
- MgraText() 322
- MIL
 - file format 155
 - objects 32, 58
 - running application 9
 - structure 8
- MIL modules
 - application 13
 - blob analysis 14
 - data allocation and access 15
 - data generation 21
 - digitizer control 19
 - display allocation 20
 - display control 20
 - graphics 21
 - image processing 22
 - measurements 25
 - optical character recognition 27
 - pattern matching 28
 - system device 30
- mil.h 10, 33
- milsetup.h 10, 37, 46, 564
- Mim...() 22
- MimAllocResult() 323
- MimArith() 325
- MimArithMultiple() 328

- MimBinarize() 330
- MimClip() 332
- MimClose() 334
- MimConnectMap() 335
- MimConvert() 337
- MimConvolve() 339
- MimCountDifference() 342
- MimDilate() 343
- MimDistance() 344
- MimEdgeDetect() 346
- MimErode() 349
- MimFindExtreme() 350
 - results 323
- MimFlip() 352
- MimFree() 353
- MimGetResult() 354
- MimGetResult1d() 356
- MimHistogram() 358
- MimHistogramEqualize() 359
- MimInquire() 362
- MimLabel() 363
- MimLocateEvent() 365
- MimLutMap() 367
 - example 570
- MimMorphic() 368
- MimOpen() 370
- MimProject() 376
- MimRank() 377
- MimResize() 379
- MimRotate() 381
- MimShift() 383
- MimThick() 384
- MimThin() 385
- MimTranslate() 389
- MimWarp() 298, 390
- MimWatershed() 393
- minimum pixel value 90
- MmeasAllocContext() 397
- MmeasAllocMarker() 399
- MmeasAllocResult() 402
- MmeasCalculate() 403
- MmeasControl() 405
- MmeasDraw() 406
- MmeasFindMarker() 409
- MmeasFree() 412
- MmeasGetResult() 413
- MmeasGetResultSingle() 418
- MmeasInquire() 419
- MmeasRestoreMarker() 423
- MmeasSaveMarker() 424
- MmeasSetMarker() 425
- MmodAlloc() 433
- MmodAllocResult() 435
- MmodControl() 436
- MmodDefine() 447
- MmodDraw() 450
- MmodFind() 452
- MmodFree() 454
- MmodGetResult() 455
- MmodInquire() 458
- MmodMask() 463
- MmodPreprocess() 465
- MmodRestore() 466
- MmodSave() 468
- mnatfct.c 570
- Mocr...() 27
- MocrAllocFont() 469, 479, 488–489, 491
- MocrAllocResult() 472, 479
- MocrCalibrateFont() 473, 489, 491–492
- MocrControl() 470, 473, 475, 489, 491–492
- MocrCopyFont() 477
- MocrFree() 472, 479
- MocrGetResult() 480, 492, 498
- MocrHookFunction() 484
- MocrImportFont() 486
- MocrInquire() 488
- MocrModifyFont() 489, 491
- MocrReadString() 492, 498
- MocrRestoreFont() 493
- MocrSaveFont() 493, 495
- MocrSetConstraint() 489, 492, 496
- MocrVerifyString() 498
- model
 - acceptance level 533
 - allocation, automatic 499
 - allocation, manual 503
 - angle of search 535
 - center 542
 - copy to image buffer 511
 - don't care pixels 511, 539
 - find 515
 - find multiple 517
 - find orientation 519
 - free 521
 - inquire 525
 - number of matches 522, 541

- positional accuracy 534
- preprocess 529
- read 530
- restore 531
- rotate 508
- save 532, 549
- search parameters 544
- morphological operations
 - custom 368
 - standard 22, 368
- Mpat...() 28
- MpatAllocAutoModel() 499
- MpatAllocModel() 503
- MpatAllocResult() 507
- MpatAllocRotatedModel() 508
- MpatCopy() 511
- MpatDraw() 513
- MpatFindModel() 515
- MpatFindMultipleModel() 517
- MpatFindOrientation() 519
- MpatFree() 521
- MpatGetNumber() 522
- MpatGetResult() 523
- MpatInquire() 525
- MpatPreprocModel() 529
- MpatRead() 530
- MpatRestore() 531
- MpatSave() 532
- MpatSetAcceptance() 533
- MpatSetAccuracy() 534
- MpatSetAngle() 535
- MpatSetCenter() 537, 542
- MpatSetCertainty() 538
- MpatSetDontCare() 539
- MpatSetNumber() 541
- MpatSetPosition() 542
- MpatSetSearchParameter() 544
- MpatSetSpeed() 548
- MpatWrite() 549
- Msys...() 30
- MsysAlloc() 9, 550
- MsysControl() 552
- MsysFree() 9, 557
- MsysInquire() 558
- multiple
 - operations 328
- multiplying, image 325, 328

N

- native mode 569
 - flag 564
 - portability 570
- neighborhood
 - kernels 130
 - operations 130
- normalization factor 130
- number of
 - holes 90
 - model matches 522
- number of cells in code 225

O

- object identifier 32
- object type
 - pseudo-MIL function 584
 - pseudo-MIL, assign 586
- occurrences, of model 522
- OCR
 - module 27
- open communication 9
- opening operation 370
- operation flags 130
- optical character recognition
 - acceptance levels 475
 - allocating result buffer 472
 - calibrating fonts 473
 - character constraints 496
 - character dimensions 475
 - contrast enhancement 476
 - copying fonts 477
 - erasing characters 475
 - freeing buffers 479
 - hooking functions 484
 - importing fonts 486
 - inquiring about fonts 488
 - inverting fonts 491
 - processing controls 470, 475
 - reading results 480
 - reading strings 492
 - restoring fonts 493
 - saving fonts 495
 - semi.txt file 487

- speeding up 476
- string location 476
- unrecognized characters 475
- verifying strings 498
- orientation
 - image 519
 - marker 430
 - object 519
- overscan
 - pixels 130

P

- panning, display 291
- parameter
 - double, pseudo-MIL 590
 - long, pseudo-MIL 593
 - MIL_ID, pseudo-MIL 591
 - null-terminated string, pseudo-MIL 596
 - pointer, pseudo-MIL 594
- parameter checking control 41
- parent buffer 11
- pattern matching
 - acceptance level 533
 - angle of search 535
 - module 28
 - number of matches 541
 - positional accuracy 534
 - result buffer 507
 - results 523
 - search position 542
- perimeter
 - blob 90
- perspective warpings 298
- pixel
 - aspect ratio 65, 405
 - don't care 511
 - location 365
 - offset 130
 - sum 90
- pointer
 - pseudo-MIL object 585
 - pseudo-MIL object, assign 587
- point-to-point operations 22
- polarity
 - edge/stripe 430
- portability, native mode 570
- position of marker 410

- position variation
 - marker 410
- positional accuracy 534
- preprocess
 - model 529
- processing
 - attribute 100, 104, 108, 144, 148
 - compensation 552
 - control 41
 - system, force 552
- profile 376
- projecting an image 376
- pseudo-MIL commands 570
- pseudo-MIL functions 570, 575
- put data
 - 1D data buffer 196
 - 2D data buffer 197
 - data buffer 189–190, 192

R

- rank filters 377
- read model 530
- reading codes 234
- reconstruct object from seed 84
- rectangles, draw 320
- rectangles, draw filled 22, 321
- reference level
 - black/white 265, 268
 - controls 268
 - digitizer 265
 - input channel 265
- reporting errors 11
- resident software, required 564
- resize image 298, 379
- restore
 - data buffer 198
 - fonts, OCR 493
 - marker 423
 - model 531
- result buffer
 - blob analysis 61, 75
 - free 72
 - image processing 323, 354, 356
 - measurement 402, 413, 418
 - pattern matching 507, 523

- retrieve data
 - 1D data buffer 170
 - 2D data buffer 171
 - color bands 162, 164
 - data buffer 161–162, 164
- rotate
 - image 298, 381
 - model 508
- roughness, blob 90
- row profile 376

S

- saturation
 - HLS 116, 119, 136, 139, 163, 165, 191, 193, 337
 - operation result 130
- save
 - data 155, 200
 - fonts, OCR 495
 - marker 424
 - model 532, 549
- scale, input 244
- scrolling, display 291
- search
 - parameters, set 544
 - position 542
- seed fill, boundary-type 313
- select
 - blob features 90
 - blob Feret diameter 97
 - blob moment 98
 - blobs to calculate 87
 - digitizer input channel 242
 - image to display 292
- semi.txt file, OCR 487
- setup flag 564
- sharpen image 339
- shearing images 298
- shift
 - bits 383
 - image 389
- spatial filtering operations
 - custom 339
 - usage 22
- standard deviation 90
- statistical operations 22
- stop grabbing 259

- structure, MIL 8
- structuring elements
 - custom 368
 - morphological transformation 368
- subtracting, image 325, 328
- sum of squares 90
- synchronization
 - of grab 246
 - with grab end 246
- system
 - allocation 550
 - control behavior 552
 - default setup configuration 564
 - device 9–10
 - free 557
 - Genesis 551
 - Host 550
 - inquire 172, 317, 558
 - Meteor 550
 - module 30
 - number 550
 - type 550
 - VGA 550

T

- text
 - character font 314
 - character size 315
 - write 22, 322
- theoretical data line 168, 194
- thickening 368, 384
- thinning 368, 385
- thread
 - allocate or control 42
- thresholding 330
- TIFF file format 155
- timer control 59
- toolkit
 - Function Developers' 569
 - Native Mode Programmers' 569
- trace
 - application 12
 - hook 53, 281
 - mechanism control 40
- transformation LUT 359

- transformations
 - generating LUTs for 299
- transforming data 155, 174
- translation, image 298, 389

V

- VCF (Video Configuration Format) 270
- VGA
 - system 550

W

- wait, grab 258
- warpings 298, 390
 - generating LUTs for 299
- watershed transforms 393
- width
 - stripe 410, 432
- width variation 411, 432
- writing codes 19, 236–238

X

- xfontscale, inquire 317

Y

- yfontscale, inquire 317

Z

- zone of influence 395
- zoom
 - display 294

Product Support

Product Assistance Request Form

[illegible]

