

MIL-Lite

version 7

User Guide and Command Reference

Manual no. 10514-801-0710

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

| | |
|--|----|
| <i>Chapter 1: Getting started</i> | 13 |
| The MIL-Lite package | 14 |
| MIL and the Intel MMX/SSE technologies | 16 |
| System requirements | 17 |
| Getting started | 18 |
| Installation | 19 |
| Building an application | 21 |

| | |
|--|----|
| <i>Chapter 2: Allocating an image buffer and grabbing images</i> | 27 |
| Getting started | 28 |
| Allocating and displaying an image buffer | 29 |
| Grabbing images | 32 |

| | |
|---|----|
| <i>Chapter 3: Specifying and managing your data buffers</i> | 35 |
| Data buffers | 36 |
| Target system | 37 |
| Specifying the dimensions of a data buffer | 37 |
| Data type and depth | 38 |
| Attribute | 38 |
| Manipulating and controlling certain data buffer areas | 42 |
| Child buffers | 42 |
| Copying specific buffer areas | 43 |
| Managing data buffers | 44 |
| Controlling how color image buffers are stored | 46 |

| | |
|---|-----------|
| RGB buffers | 47 |
| Binary buffers | 49 |
| YUV buffers | 49 |
| YUV16 Packed | 50 |
| YUV9 Planar | 51 |
| YUV12 Planar | 51 |
| YUV16 Planar | 52 |
| YUV24 Planar | 53 |
| Child YUV buffers | 53 |
| Accessing a MIL buffer directly. | 54 |
| Mapping a data buffer to user-allocated memory | 55 |
| Pixel conventions | 58 |
| Using buffers with the Bayer color filter | 59 |
| Using MIL to convert the image | 60 |
| How the Bayer image gets converted | 62 |
| White balancing your Bayer images | 64 |
| <hr/> | |
| <i>Chapter 4: Lookup tables</i> | <i>67</i> |
| Lookup tables | 68 |
| LUTs and data buffers | 69 |
| Loading and generating data into LUTs | 69 |
| Generating data directly into the LUT buffer | 69 |
| Loading LUTs with precalculated data | 70 |
| Using LUTs | 71 |
| Displaying using LUTs. | 71 |
| LUTs and digitizers | 72 |

Chapter 5: Displaying an image73

| | |
|---|-----|
| Displaying an image | 74 |
| Types of displays | 75 |
| Windowed display | 75 |
| Auxiliary display | 76 |
| Display number | 79 |
| Display size and depth | 79 |
| Displaying buffers of different data depths | 80 |
| Removing a buffer from the display | 81 |
| Displaying multiple buffers | 82 |
| Panning, scrolling, and zooming | 85 |
| Annotating the displayed image non-destructively | 86 |
| Using GDI annotations | 88 |
| Displaying an image in a user-defined window | 90 |
| Using MdispSelectWindow() | 90 |
| Palettes and output LUTs for windowed display (256-color). | 94 |
| Reference material: Windows palettes and physical output LUTs. | 94 |
| Default palette settings | 96 |
| Changing the default LUT values | 97 |
| CPU-assisted display. | 100 |

Chapter 6: Generating graphics 101

| | |
|----------------------------------|-----|
| MIL and graphics | 102 |
| Preparing for graphics | 102 |
| Drawing graphics | 104 |
| Writing text | 106 |

Chapter 7: Grabbing with your digitizer 107

| | |
|---|-----|
| Cameras and input devices | 108 |
| The data format | 109 |
| The digitizer number | 110 |
| Multiple cameras | 110 |
| Grabbing a single field | 111 |
| Line-scan cameras | 111 |
| Grabbing to the display | 112 |
| Live and pseudo-live continuous grabs. . . | 112 |
| Live transfer to the display. | 113 |
| Pseudo-live transfers to the display | 113 |
| Screen Tearing. | 116 |
| Reference levels, lookup tables, and scaling . | 117 |
| Black and white reference levels | 117 |
| Color image reference levels. | 119 |
| Mapping grabbed data through a LUT . . . | 119 |
| Scaling | 119 |
| Optimizing application performance when grabbing | 121 |
| Grab mode | 121 |
| Double buffering | 122 |
| Multiple buffering | 124 |

| | |
|--|------------|
| Grabbing a sequence of frames in real-time. | 125 |
| Grabbing with triggers and exposures | 125 |
| Asynchronous reset mode. | 126 |
| Triggers and exposures. | 127 |
| Software triggers. | 130 |
| <hr/> | |
| <i>Chapter 8: Color</i> | <i>131</i> |
| Dealing with color | 132 |
| Grabbing. | 132 |
| Displaying. | 134 |
| Saving and loading color images | 135 |
| How to manage your color buffer. | 135 |
| <hr/> | |
| <i>Chapter 9: JPEG compression</i> | <i>139</i> |
| Introduction | 140 |
| General steps | 141 |
| Controlling a JPEG compression | 143 |
| JPEG lossless | 143 |
| JPEG lossy | 144 |
| Restart markers | 145 |
| Improving results | 146 |
| Working with tables. | 147 |
| Inquiring values in default tables | 147 |
| Using your own table | 148 |

Chapter 10: Data manipulation with multiple systems 149

 Data manipulation with multiple systems. . . 150

Chapter 11: Using MIL with multi-processing and under multi-thread systems 151

 Multi-processing 152

 Multi-threading 153

 MIL and multi-threading 154

Chapter 12: Using MIL with Native Mode Functions . 163

 Integrating native functions with MIL code . . 164

 Portability 164

 Signaling MIL about Native Mode use. . . . 164

 A native mode example. 165

Chapter 13: Distribution 169

 Distribution of MIL-Lite-based applications . 170

 Redistributing MIL-Lite run-time DLL files
 and device drivers with your application. . . . 170

 Redistributing directly from the
 MIL-Lite CD. 170

 Redistributing using your own setup
 program. 171

 Normal redistribution using your
 custom CD. 171

 Silent redistribution 172

 Response file parameters 173

 Debugging the response file 175

| | |
|--|------------|
| Important notes for Windows 98/Me users | 176 |
| Important notes for Windows NT/2000 users | 176 |
| Uninstallation | 177 |
| MIL and MIL-Lite licenses | 178 |
| <hr/> | |
| Chapter 14 : Programming with MIL. | 181 |
| A MIL overview | 182 |
| Starting your MIL application | 183 |
| Header file and libraries | 184 |
| MIL object manipulation concepts. | 184 |
| Error handling and reporting | 185 |
| Tracing an application | 186 |
| A quick command reference | 187 |
| The application allocation and control module | 187 |
| The buffer allocation and access module. . | 188 |
| The digitizer allocation and control module | 190 |
| The display allocation and control module | 191 |
| The basic data generation module. | 192 |
| The basic graphics module | 192 |
| The system allocation and inquiry module | 193 |
| <hr/> | |
| Chapter 15: The command reference descriptions. . | 195 |
| The reference description notes | 196 |

Appendix A: The default setup configuration file . . . 409

The default setup configuration file 410

When you do not want to use defaults 413

Appendix B: The MIL Function Developer's Toolkit 415

The MIL Function Developer's Toolkit 416

An example using the Function Developer's Toolkit 416

MIL Function Developer's Toolkit Command Reference 419

Index

Product Support

Part I:
Using
MIL-Lite



To gain portable keys to...
image acquisition
image display

Chapter 1: Getting started

This chapter presents the MIL-Lite package features. It also explains the installation process and how to run a MIL-Lite application program.

The MIL-Lite package

MIL-Lite is a subset of MIL, the Matrox Imaging Library package. It includes all the MIL features for acquisition, data manipulation, graphics, and display control. Since all MIL-Lite features are identical to those in MIL, we use the word "MIL" to represent "MIL-Lite" throughout this manual.

The MIL package is a hardware-independent modular 32-bit imaging library. In general, MIL can manipulate either binary, grayscale, or color images.



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

Image acquisition

Images can be loaded from disk or acquired from the wide range of supported input devices (if hardware permits) and can be stored in your platform's storage area. Sequences of images can also be loaded and saved in .avi format. A Bayer filter is also included in MIL, which allows you to grab images with cameras using Bayer filters, and then convert them into 3-band color or single-band monochrome images.

Graphics capabilities

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

Creating your own MIL functions

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

MIL objects

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

Image pixel depth

The MIL package can:

- Grab up to 16-bit grayscale images, or color images
- Display 1, 8, or 16-bit grayscale or color images (if the platform supports it).

MIL documentation's word usage

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

Command descriptions

Descriptions of the individual commands are found in the *Command Reference* part of this manual.

MIL and the Intel MMX/SSE technologies

MIL has been optimized, in assembly language, to take advantage of Intel MMX acceleration and Streaming SIMD Extensions (SSE).

MMX

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

SSE

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

System requirements

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

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

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

Supported compilers

The MIL CD includes MIL libraries that support the Microsoft Visual C++ 6.0 (service pack 5) compiler under Windows NT 4.0 (service pack 6), Windows 98 SE, Windows Me, and Windows 2000. The CD also includes ActiveMIL ActiveX controls for Microsoft Visual Basic 6.0 (service pack 5) and Microsoft Visual C++ 6.0 (service pack 5) RAD tools. The service pack indicated in parentheses denotes the actual platform used for testing.

Getting started

Getting started

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

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

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

Installation

To install your MIL software, place the installation CD in an appropriate drive. The *setup.exe* program will run automatically.

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

- The drive and directory on which to install the program.
- Your development tool.
- The type of Matrox hardware installed in your computer (for example, Matrox Corona-II). Note that under Windows 98/Me/2000 the boards have to be installed before the Matrox frame grabber drivers are installed.
- Whether to install the MGA drivers. This will only be asked if you have a Matrox Imaging board with a display section or a Matrox graphics board, and the drivers to be installed are newer than the drivers already on your computer.
- The digitizer and display format to load into the default setup file, *milsetup.h*.
- Whether by default, displayed images should be displayed in a window on the Windows desktop or without a window on an auxiliary screen (a non-Windows desktop screen). If the answer is the latter, you will be asked for the **video configuration format (VCF)** to use by default. Auxiliary screens require either two graphics controllers or a DualHead graphics controller that integrates two CRT controllers.
- The amount of DMA linear non-paged memory to reserve for grab buffers. The amount of reserved DMA memory also establishes the amount of remaining RAM available to your operating system.

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

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

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

MIL Configuration utility

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

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

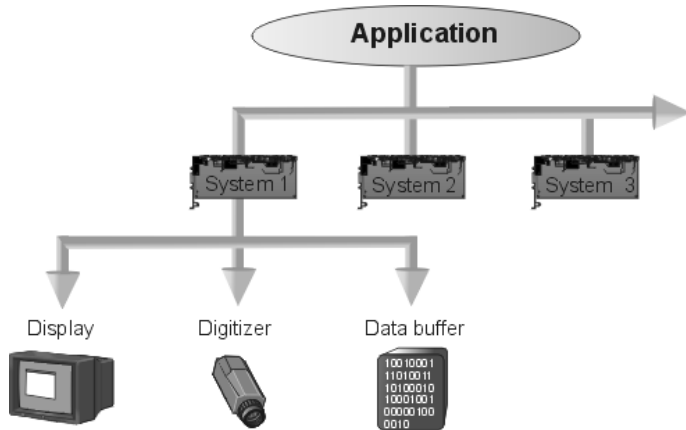
Note that MIL's 30-day evaluation license can only be installed once. Any attempt to tamper with the PC's calendar, before the date of expiry, will disable MIL. In that event, MIL can only be re-used once a license is obtained.

Building an application

Initialization

At the beginning of each application, you must:

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



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

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

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

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

Multiple systems

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

The default image buffer

If a color digitizer configuration format (DCF) was specified upon installation, the default image buffer is defined as a color buffer (RGB) in the *milsetup.h* file. Note, most examples in this manual assume that the default image buffer is a monochrome buffer. You will have to modify the examples appropriately in order to run them with color defaults. For more details on dealing with color, see Chapter 8.

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

Error reporting

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

Compiling and linking

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

| MIL Libraries | |
|----------------------|--------------------|
| Library | Description |
| mil.lib | Core library |
| milvga.lib | VGA library. |

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

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

Testing installation

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

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

```

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

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

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

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

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

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

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

```

Communicating properly?

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

Examples in general

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

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

Chapter 2: Allocating an image buffer and grabbing images

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

Getting started

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

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

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

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

Later in this manual, we discuss changing the current input channel, how to specify a different digitizer format, and how to allocate different types of image buffer. With that knowledge, you can return to this chapter and modify the examples. Chapter 8 discusses dealing with color in detail.

Allocating and displaying an image buffer

Allocating an image buffer

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

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

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

Displaying an image buffer



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

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

```

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

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

#define IMAGE_DEPTH 8L

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

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

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

```

(cont...)

```

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

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

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

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

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

```

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

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

Displaying multiple buffers

With MIL, you can also display multiple buffers. This is discussed later in the manual, in *Chapter 5: Displaying an image*.

Grabbing images

Grabbing an image



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

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

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

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

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

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

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

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

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


Allocate the grab image buffer on the same system, and of the same data format type, as the digitizer. For color input devices, use color image buffers (see *Chapter 8: Color*).

By default, when *MdigGrab()* is issued, it grabs a complete frame of data. Use *MdigControl()* to control the number of frames or fields grabbed by *MdigGrab()*. To control the digitizer, see *Chapter 7: Grabbing with your digitizer*.

*Continuous grabbing
and adjusting your
camera*

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

This is discussed in greater detail in *Chapter 7: Grabbing with your digitizer*. The following example is of adjusting a camera using a continuous grab.

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

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

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

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

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

(cont...)

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

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

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

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

Chapter 3: Specifying and managing your data buffers

This chapter discusses data buffers in detail. It shows you how to allocate and manage data buffers, and how to restrict an operation to a portion of a data buffer by using child buffers. It shows you how YUV buffers are stored, how to create a user-defined buffer, and how MIL defines the pixel reference position. It shows you how to grab images with a Bayer camera and restore the color information.

Data buffers

Data buffers

In this manual, the term *data buffer* is used loosely to refer to the most general type of data buffer (storage area) that is allocated by the MIL package and operated on by most MIL functions. For example, a data buffer can be a buffer for image data or one for lookup table (LUT) data. Besides data buffers, there are also other buffers (for example, result buffers), which are specific to a particular group of functions. These types of buffers are discussed in the chapters describing their related functions.

Allocating data buffers

All data buffers must be allocated before a function can access them. You can allocate a monochrome buffer using *MbufAlloc1d()*, *MbufAlloc2d()*, or *MbufAllocColor()*. You allocate a color buffer using *MbufAllocColor()*.

When allocating a data buffer, you must specify its:

- Target system.
- Dimensions.
- Data type and depth.
- Attribute.

Controlling specific parts

You can manipulate or control specific parts of data buffers by allocating and using child buffers. A child buffer is a subset of the parent buffer (a specific area of the parent buffer). Although any change made to the child buffer data affects the parent buffer, the buffer is considered a data buffer in its own right; wherever the parent buffer can be used, you can use the child buffer instead to affect only a part of the buffer. All results are returned relative to the child buffer coordinates rather than the parent buffer.

Target system

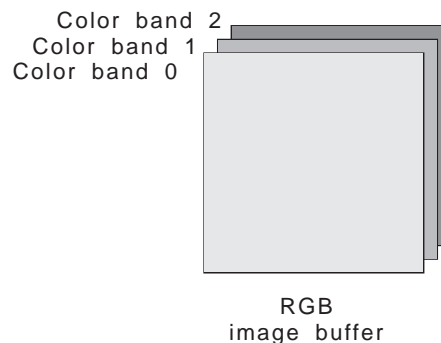
A data buffer is allocated on the specified system. If the `M_DEFAULT_HOST` system is specified, the default Host system of the current MIL application will be used. If `M_DEFAULT` is specified, MIL will select the most appropriate system on which to allocate the data buffer (it can be the default Host system or any currently allocated system).

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

Specifying the dimensions of a data buffer

Data buffers can have up to three dimensions: an x, y, and color band dimension. Most data buffers have an x dimension (for example, LUT buffers) or an x and y dimension (for example, monochrome image buffers). The color-band dimension has been provided to allow you to store data for each color component used to represent an image; when allocating color buffers, each band will be of the same data depth and type.

Once you finish using a data buffer, you should release its memory space, using *MbufFree()*.



Certain MIL functions support manipulating multi-band image buffers. See *Chapter 8: Color* for details on handling color image buffers.

Data type and depth

Data type and depth

The data depth of a buffer indicates the number of bits per band in the buffer (1, 8, 16, 32). The data type of a buffer indicates how its data is internally represented (that is, whether the data is considered signed, unsigned, or floating-point). Supported combinations are: 1-bit packed binary; 8-, 16-, and 32-bit integer (signed and unsigned); and 32-bit floating-point. If a function can only operate on data buffers of certain depths, this is explicitly stated in the command's description, otherwise the function can be used with any combination of data buffers (the *The MIL-Lite User Guide and Command Reference* manual).

Packed binary buffers

The packed binary data format represents each pixel by a single bit, in a state of 0 or 1. Therefore, 8 pixels can be packed in a single byte (known as an 8-bit data unit); that is, in a format eight times smaller than an 8-bit image.

Integer and floating-point buffers

In general, the fewer bits per pixel in a buffer, the faster an operation can be performed on the buffer. Packed binary buffers are the fastest to process. When you need to use integer buffers, use 8 bits per pixel when possible, 16 bits if necessary, and 32 bits as a last resort. When you need non-integer values, extra precision, or a greater dynamic range, you can use floating-point data buffers.

Attribute

Buffer type and usage

The data buffer attribute indicates the buffer type and its intended usage. MIL uses this information to determine the most appropriate location in physical memory in which to allocate the buffer, and how to handle the buffer. A data buffer can be one of the following types:

- M_IMAGE (image buffer).
- M_LUT (lookup table buffer).

- M_KERNEL (kernel buffer for convolution functions).
- M_STRUCT_ELEMENT (structuring element buffer for morphology functions).

Allocating an image buffer

When allocating an image buffer (M_IMAGE), you must give more information about its intended usage. An image buffer can be any combination of the following:

- A buffer that can be displayed (M_DISP).
- A buffer in which data can be grabbed (M_GRAB).
- A buffer in which data is stored in a compressed format (M_COMPRESS).

For example, to allocate an image buffer that can be displayed and used for processing, its attribute should be given as:

M_IMAGE + M_DISP + M_GRAB

In general, buffers are allocated in Host memory instead of on-board memory by default. This is because on-board memory is limited in size and Host memory can be accessed much faster than on-board memory. However, if the system has an on-board processor, the buffer is allocated on-board by default. These defaults can be overridden by using the *MbufAlloc...()* M_ON_BOARD and M_OFF_BOARD attributes.

Grab buffers

Buffers with an attribute of M_GRAB are allocated in DMA memory, which is physically contiguous and always present. This is also known as non-paged memory. An advantage to non-paged memory is that a bus mastering device can write to it without the help of the CPU.

If a system does not support grab buffers (for example, M_HOST_SYSTEM), you could still allocate a buffer on such a system in physically contiguous and always present memory by giving it an M_NON_PAGED attribute instead.

Displayable buffers

When a displayable buffer is allocated and selected for display (*MbufAlloc...()* with M_DISP, and then *MdispSelect()*), two buffers are maintained internally: one in Host memory for processing purposes, the other in a frame buffer (maintained directly or through a DIB) for display purposes (not necessarily the same size). When the Host buffer is modified, its associated

buffer in the frame buffer is automatically updated. When displaying a buffer, both the buffer and the display should be allocated on the same system.

When grabbing a single frame into a displayable buffer, MIL grabs into the Host memory version of the buffer and then updates the display of the buffer. When grabbing continuously, the grab is made directly to the frame buffer and then at the end of the grab, the Host buffer is updated.

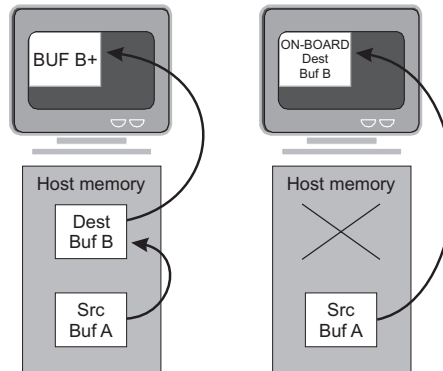
Overriding the default allocation sequence

On boards with a display section, you can override the default buffer allocation sequence and force allocation only in the frame buffer using the *MbufAlloc...()* M_ON_BOARD attribute. In general, the buffer is allocated in the non-displayable area of the frame buffer. When the M_DISP attribute is specified for auxiliary displays, the buffer will be in the displayable area. You can allocate only one M_DISP+M_ON_BOARD buffer and one M_OVR+M_ON_BOARD buffer unless stated in the *MIL/MIL-Lite Board Specific Notes* manual.

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

Overriding the default allocation sequence is useful when allocating a displayable buffer for any auxiliary display. If you are not using the displayable buffer for processing or are only

using it as a destination, storing the buffer on-board will avoid the extra copy operation to the display without the penalty of slowing down processing.



Even if it is not in the displayed area of the frame buffer, the image buffer depth and display depth must be the same.

Internal format of the buffer

It is also possible to force the internal representation of a data buffer using internal storage format specifiers, such as `M_PACKED` or `M_PLANAR`, which force the data buffer to be in a packed or planar format, respectively. Refer to *MbufAllocColor()* for a complete list of internal format specifiers.

Insufficient memory

If there is insufficient memory of the appropriate type to allocate a buffer with the specified attributes, the function generates an error and does not allocate the buffer.

Inappropriate data buffer usage

If you try to use a data buffer in a situation that is not appropriate for its allocated attribute, an error message is generated and the operation is not performed. For example, if you try to display a buffer without an `M_DISP` attribute with *MdispSelect()*, an error message will be generated.

Manipulating and controlling certain data buffer areas

You can manipulate or control specific parts of a data buffer by creating a child buffer within it or by copying specific parts of it to another buffer.

Child buffers

Child buffers are subsets of parent buffers

A child buffer is a subset (or region of interest) of a given data buffer (known as the parent buffer). Child buffers occupy a specific area of the parent buffer. Since this area is part of the same physical space as the parent buffer, changes made to the child buffer affect the parent buffer and vice versa.

Allocating child buffers

The child buffer is considered a data buffer in its own right. Like its parent buffer, a child buffer must be allocated so that it can be associated with an identifier and recognized as an entity by the MIL package. Allocate a monochrome child buffer using *MbufChild1d()* or *MbufChild2d()*. To allocate a child buffer consisting of only one of the color bands of a multi-band image buffer, use *MbufChildColor()* or *MbufChildColor2d()*. Note, as a subset of the parent buffer, a child buffer cannot exceed the bounds of its parent in any dimension. For example, a color buffer cannot be created from a monochrome parent buffer.

A child buffer takes on the same attributes and type as the parent buffer. In general, any operation that can be performed on the parent buffer can also be performed on the child buffer.

Allocate a child buffer by specifying its size and offset with respect to each of the parent buffer dimensions. After, when using the child image buffer, any specified or returned coordinates are relative to the child's top-left corner.

As with any MIL data buffer, once you have finished using a child data buffer, you must delete it, using *MbufFree()*.

One major benefit of the child buffer is being able to handle several buffers simultaneously, in contexts where normally only one buffer can be handled. For example, for auxiliary displays, you can only display one buffer at a time. However, you might want to display the source and destination buffer of an operation simultaneously. You can get around this situation by allocating a displayable image buffer as large as the display, then allocating two child buffers from this buffer. You can then use one as the source data buffer and one as the destination. When the parent buffer is selected on the display (*MdispSelect()*), both the source and the destination child buffers can be seen.

Copying specific buffer areas

As an alternative to using a child buffer, you can restrict operations to specific areas or bits of a **buffer** (child or parent) by copying the required portions to another buffer. You can copy data from any type of data buffer to another using any of the following functions. For example:

- Copy an image buffer to another buffer at the specified offset, using *MbufCopyClip()*. Data that falls outside of the destination buffer will be automatically clipped.
- Copy specific non-sequential areas to another buffer based on a conditional buffer, using *MbufCopyCond()*. Source buffer data is copied to the destination buffer if corresponding data in the specified conditional buffer satisfies the copy condition. Other data in the destination buffer is left unaffected.
- Copy specific non-consecutive bits to another buffer based on a mask, using *MbufCopyMask()*. Only destination bits that correspond to non-zero bits in the mask are modified with source bits.
- Copy a single band of a multi-color band buffer to or from a single-band buffer, using *MbufCopyColor()* or *MbufCopyColor2d()*. This allows you to operate on a single color band of a buffer.

If the source buffer depth is greater than that of the destination, the most significant bits are truncated when the data is copied into the destination. If the source is signed and the destination depth is greater than the source, the source data is sign-extended when it is copied into the destination.

MbufCopy() copies the entire buffer into another buffer, while the other commands copy only portions of a buffer.

Managing data buffers

Besides the copy functions discussed in the previous section, MIL provides several other data buffer management functions. These allow you to transfer data between an array and a buffer, load data into a buffer (or a sequence of buffers), and save a buffer (or a sequence of buffers) to disk.

Putting and retrieving data

You can put data from an array into a data buffer, using *MbufPut()*, *MbufPut1d()*, *MbufPut2d()*, *MbufPutColor()*, or *MbufPutColor2d()*. *MbufPut()* puts data in the entire buffer, while *MbufPutColor()* or *MbufPutColor2d()* put data into one or all color bands of a multi-band buffer. The other two commands allow you to put data in a selected area of a monochrome buffer, respectively.

In addition, you can retrieve data from a data buffer and place it into an array, using *MbufGet()*, *MbufGet1d()*, *MbufGet2d()*, *MbufGetColor()*, or *MbufGetColor2d()*. *MbufGet()* gets data from the entire buffer, while *MbufGetColor()* or *MbufGetColor2d()* get data from one or all bands of a multi-band buffer. The other two commands, like their ‘put in buffer’ counterparts, allow you to get data from a selected area of a monochrome, respectively.

❖ Note that you can also access the contents of a MIL buffer from an array by using *MbufInquire()*. Inquire the Host address of the buffer, and then using a pointer access the buffer as an array. This is discussed in more detail later.

Loading a data buffer

You can load data, using one of two methods:

- Load data into an automatically allocated MIL data buffer, using *MbufImport()* with M_RESTORE, or using *MbufRestore()*.
- Load data into a previously allocated MIL data buffer, using *MbufImport()* with M_LOAD or using *MbufLoad()*.

These commands internally handle the opening and closing of the file. With *MbufImport()*, you can specify the file's format. *MbufLoad()* and *MbufRestore()* will read the data in the file to determine the format, therefore they might take more time to return a result.

Saving a data buffer

You can save a data buffer to disk, using *MbufExport()* or *MbufSave()*. *MbufExport()* is the most general of these commands and can save data in any MIL-supported file format. *MbufSave()* can only save data in an M_MIL file format.

These functions internally handle opening and closing the file. If the given file name already exists, the file will be overwritten.

Loading and saving a sequence of data buffers

You can import or export a sequence of image buffers to a file using *MbufImportSequence()* or *MbufExportSequence()*, respectively. The available file formats are: standard AVI DIB format, MJPEG format, and proprietary AVI MIL format.

Controlling how color image buffers are stored

A color image buffer's internal representation can be either in a planar or packed format. When allocating the buffer, if its attribute is also set to `M_PLANAR`, the pixels are stored in planes (for example, RRR GGG BBB). When allocating the buffer, if its attribute is set to `M_PACKED`, each pixel is stored as one unit containing all its components (for example, RGB RGB RGB).

MIL automatically selects the most appropriate format, according to the specified intended usage attribute. If an image buffer is allocated in one format, and a general processing function requiring another format is called, the function will automatically convert the data to the required format and re-convert it back to its original format upon completion. To change a buffer's default internal storage format, change the internal storage part of the attribute parameter for *MbufAllocColor()*. Note that it might be slower to process buffers with `M_PACKED` attributes.

In general, packed formats are mostly used for display purposes; when selecting a buffer's attribute as `M_DISP`, the default internal representation is usually packed. This configuration allows for faster transfers to display sections that handle packed data (for example, VGA). However, if the display section of your board has dedicated red, green, and blue frame buffer planes, the buffer is allocated in planar format.

Planar formats are generally preferred for processing. Here, the buffer stores each pixel as three component planes (for example, RRR, GGG, BBB). Processing is done on each of the components separately.

When allocating an image buffer with more than one attribute, for example, `M_DISP` and `M_PROC`, the buffer's internal storage requirements for the display will take precedence over other attributes.

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

RGB buffers

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

RGB data formats

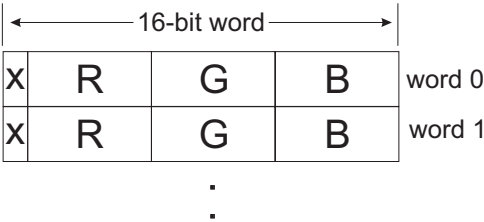
BGR24 packed (M_BGR24+M_PACKED) is a format whereby each pixel is internally stored as three consecutive bytes in little-endian order, that is:

| | |
|--------|---|
| Byte 0 | B |
| Byte 1 | G |
| Byte 2 | R |
| Byte 3 | B |
| Byte 4 | G |
| Byte 5 | R |
| | . |
| | . |
| | . |

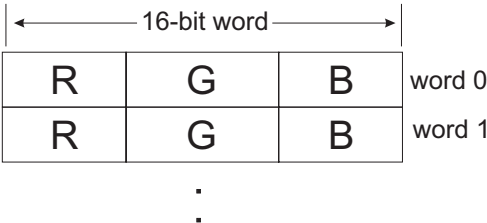
BGR32 packed (M_BGR32+M_PACKED) is a format whereby each pixel is internally stored as four consecutive bytes, in little-endian order. The most-significant byte is a "don't care" byte, as shown below:

| | |
|--------|---|
| Byte 0 | B |
| Byte 1 | G |
| Byte 2 | R |
| Byte 3 | X |
| Byte 4 | B |
| Byte 5 | G |
| Byte 6 | R |
| Byte 7 | X |
| | . |
| | . |
| | . |

RGB15 packed (M_RGB15+M_PACKED) is a format whereby each pixel is internally stored as a 16-bit word with a 5-bit blue value (least significant), a 5-bit green value, a 5-bit red value, and a "don't care" bit (most significant), in little-endian order, as shown below. Note that when accessing an M_RGB15+M_PACKED buffer as a 3-band 8-bit buffer, the least significant bits of each band are set to 0.



RGB16 packed (M_RGB16+M_PACKED) is a format whereby each pixel is internally stored as a 16-bit word with a 5-bit blue value (least significant), a 6-bit green value, and a 5-bit red value (most significant), in little-endian order, as shown below. Note that when accessing an M_RGB16+M_PACKED buffer as a 3-band 8-bit buffer, the least significant bits of each band are set to 0.



RGB planar are formats whereby the color components of all the pixels are stored contiguously: (RRR..., BBB..., GGG...).

Binary buffers

Binary buffers have a different internal storage format than other types of buffers: eight pixels are stored in one byte. The leftmost pixel of an image is the least significant bit that is stored in memory.

YUV buffers

YUV is a compressed format in which Y is the grayscale component (luminance) and U and V are the color components. MIL supports grabbing, loading, or saving images in a YUV color format.

Although any general processing operation can be performed on YUV buffers, allocating them for processing purposes is not recommended because MIL is configured to process RGB color data only. However, MIL will automatically convert YUV buffer data to RGB for all general processing operations (including conversion for display), and re-convert it to YUV upon completion.

All YUV formats are supported even on the Host system. However, only some systems support grabbing into YUV buffers. See the *MIL/MIL-Lite Board-Specific Notes* manual to determine if grabbing into YUV buffers is supported on your system.

YUV buffers must be allocated as 3-band 8-bit buffers, however, the actual number of bits per pixel will differ depending on the YUV format selected.

The supported YUV formats are:

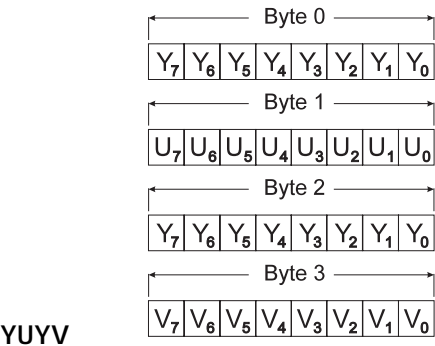
- YUV16 Packed
- YUV9 Planar
- YUV12 Planar
- YUV16 Planar

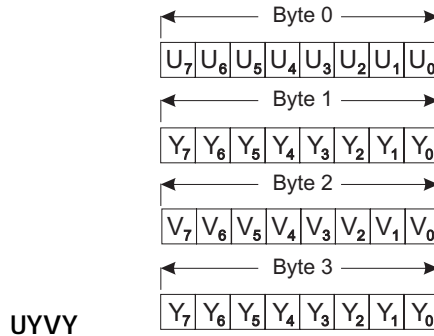
YUV16 Packed

YUV16 Packed or YUV 4:2:2 (M_YUV16+M_PACKED) is an interleaved data format. Although each pixel has a corresponding one byte Y (luminance component), each pair of pixels share the same one byte U (chrominance U) and the same one byte V (chrominance V). Since a pair (two pixels) is represented by 4 bytes, each pixel has an average of 16 bits per pixel.

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

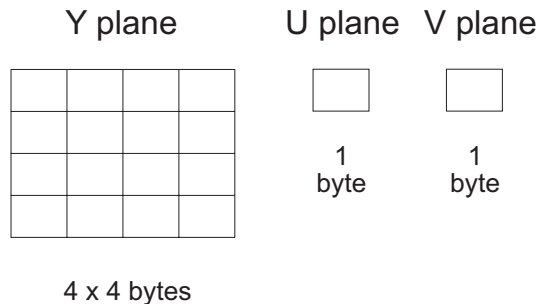
When you allocate an M_YUV16+M_PACKED buffer, MIL allocates the buffer in the format that is most suitable for the selected platform and the specified buffer attributes. You can, however, force a format using the M_YUV16_YUYV or M_YUV16_UYVY control types. When the buffer has an M_GRAB attribute, forcing an inappropriate format generates an error. When the buffer has an M_DISP attribute, if you force the buffer in the other YUV format, then CPU intervention is required to perform the automatic conversion. See the *MIL/MIL-Lite Board Specific Notes* for supported data formats.





YUV9 Planar

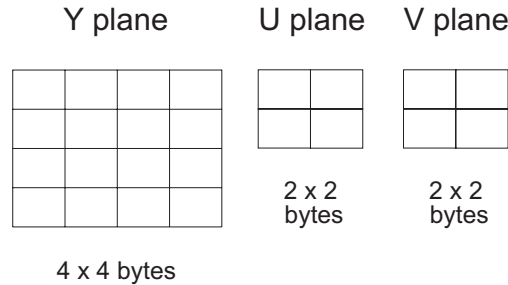
YUV9 Planar (M_YUV9+M_PLANAR) is a planar format whose components have a depth of one byte but are not of the same size. Although each pixel has a corresponding 1 byte Y (luminance) component, each block of 16 pixels share the same one byte of U (chrominance U) and the same one byte of V (chrominance V). Since the 16 pixels are represented by 18 bytes, each pixel has an average 9 bits. For example, a block of 16 pixels has the following: 16 bytes Y and 1 byte each of U and V.



YUV12 Planar

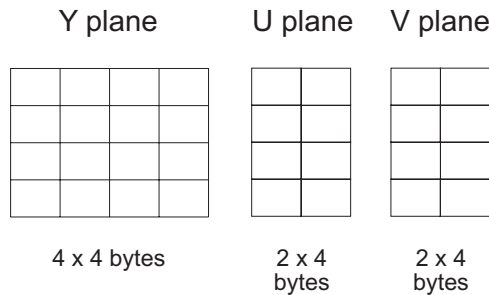
YUV12 Planar (M_YUV12+M_PLANAR) is a planar format whose components have a depth of one byte but are not of the same size. Although each pixel has a corresponding 1 byte Y (luminance) component, each block of 4 pixels share the same one byte of U (chrominance U) and the same one byte of V

(chrominance V). Since the 16 pixels are represented by 24 bytes, each pixel has an average of 12 bits. For example, a block of 16 pixels has the following: 16 bytes Y and 4 bytes each of U and V.



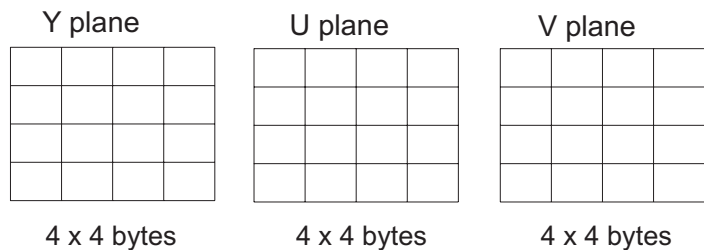
YUV16 Planar

YUV16 Planar (M_YUV16+M_PLANAR) is a planar format whose components have a depth of one byte but are not of the same size. Although each pixel has a corresponding 1 byte Y (luminance) component, each block of 2 pixels share the same 1 byte of U (chrominance U) and the same 1 byte of V (chrominance V). Since the 16 pixels are represented by 32 bytes, each pixel has an average 16 bits. For example, a block of 16 pixels has the following: 16 bytes Y and 8 bytes each of U and V.



YUV24 Planar

YUV24 Planar (M_YUV24+M_PLANAR) is an uncompressed planar format whose components have a depth of one byte and are of equal size. Each pixel has a corresponding 1 byte Y (luminance) component, 1 byte U component (chrominance U), and 1 byte V component (chrominance V). Since the 16 pixels are represented by 48 bytes, each pixel has an average 24 bits. For example, a block of 16 pixels has the following: 16 bytes Y and 16 bytes each of U and V.



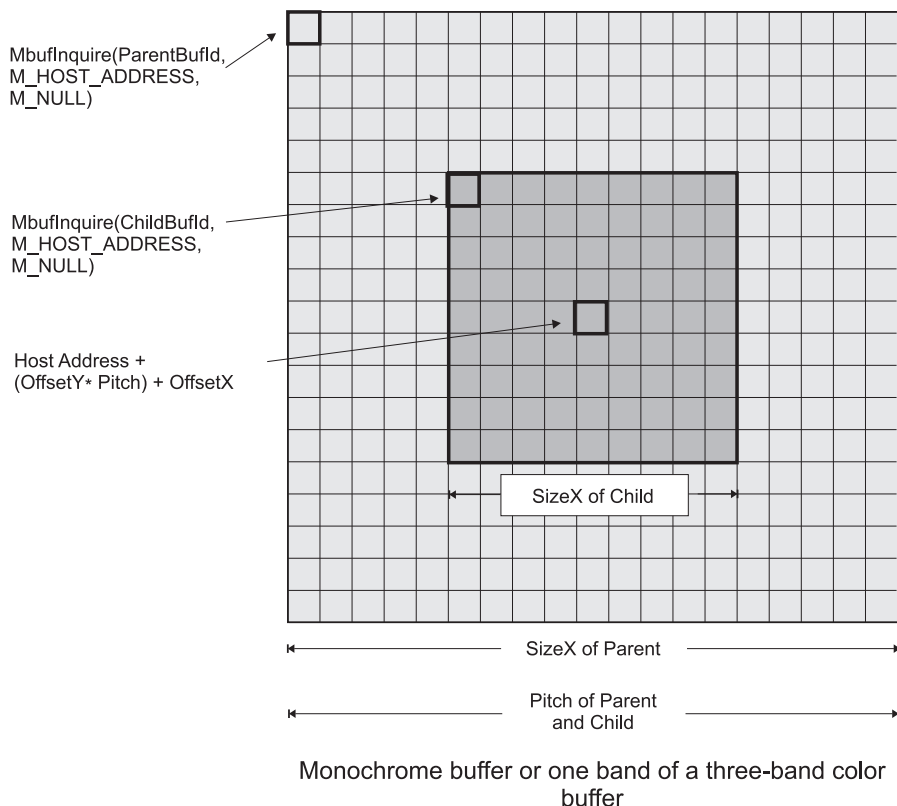
Child YUV buffers

You can create child buffers from YUV buffers in the same way as RGB child buffers. When creating YUV child buffers, MIL will keep the proportions of the U and V bands with respect to the Y band. For example, if your YUV9 Planar Y band is a size of 256 x 256 pixels, the U and V bands will be 1/4 the size of the Y band in each dimension (width and height): 64 x 64 pixels, which is 1/16 the size of the Y band. If a child buffer is 16 x 16 pixels, then the U and V bands will be 4 x 4 pixels. In other words, the 4 x 4 U and V bands (16 pixels) is 1/16 the size of the Y band (256 pixels).

Accessing a MIL buffer directly

If needed, a MIL buffer's contents can be accessed directly. For instance, if you want to calculate the average value of the pixels of your image, you could create a custom algorithm. The algorithm could be applied directly to the buffer without having to copy the contents of the MIL buffer into a user-allocated array (*MbufAlloc()*) by using *MbufGet()* and *MbufPut()*. To do so would be more efficient and might improve the performance of the custom algorithm.

In order to access the MIL buffer directly, the buffer's address and pitch must be known. Once you know this, you will be able to access them directly for optimum performance.



Address

The address of a parent or child buffer can be returned using *MbufInquire()*. Selecting `M_HOST_ADDRESS` will return a logical address, while `M_PHYSICAL_ADDRESS` will return a physical address. In either case, the first address of the buffer you are specifying will be the top left-most pixel in the image. Knowing the pitch and the depth of the buffer will tell you the address of the following row.

Pitch

The pitch of a buffer is the number of units between the beginnings of any two adjacent lines of the buffer's data and can be measured in pixels or bytes. Note that in some instances, the pitch in bytes will be more accurate than in pixels. If the last pixel falls outside of a 32-bit boundary (required by Windows), the start of the next row will be located at the beginning of the next 32-bit boundary; this process is called internal padding. When measuring the pitch in pixels, the padding can be counted as "extra" pixels, depending on the depth of the pixels. This will result in an inaccurate pitch.

Mapping a data buffer to user-allocated memory

Instead of allocating new memory to a buffer using *MbufAlloc...()*, you can create a buffer from the memory at a specified location, using *MbufCreate2d()* to create a monochrome data buffer and *MbufCreateColor()* to create a color data buffer. In these cases, MIL does not allocate any memory; it uses the memory that you provide.

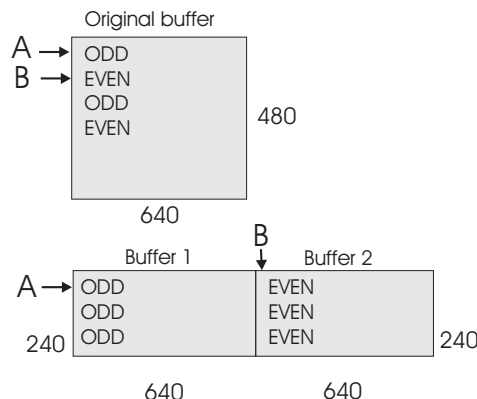
When creating a buffer with *MbufCreateColor()*, you must pass an array of pointers to the addresses of the data. For packed color buffers, you must pass an array of one pointer; for planar buffers, you must pass an array with the same number of pointers as the number of bands in the buffer. When creating a buffer with *MbufCreate2d()*, you must pass the address of the data. The address(es) can be either logical or physical. If you want to use the buffer for grabbing, the address(es) must be physical (grab buffers must be allocated in physically contiguous and always present memory, that is, non-paged). The *MbufCreate...()* functions must be used with caution because, when using physical addresses, these functions

provide direct manipulation of any of your PC's memory mapped devices; when using logical addresses, memory protection errors could result.

You can use *MbufInquire()* with the M_HOST_ADDRESS or M_PHYSICAL_ADDRESS control type to determine the Host's logical address or the physical address of a buffer's data, respectively. Note that the physical address is not necessarily an address in Host memory. It could be an address in on-board memory. If an on-board buffer is mapped to the Host, you can use the *MbufInquire()* function with the M_HOST_ADDRESS inquire type to determine the Host address to which it is mapped.

There are several instances when memory mapping is useful. A particularly useful instance is when processing and displaying an interlaced grab in a time critical application. In this case, you could use a displayable buffer to store and display the grabbed data. Then, to process each field as it is grabbed, you could use a buffer that is mapped to the odd field of the displayable buffer (Buffer 1) and a buffer that is mapped to the even field (Buffer 2).

Create Buffers 1 and 2 as follows:



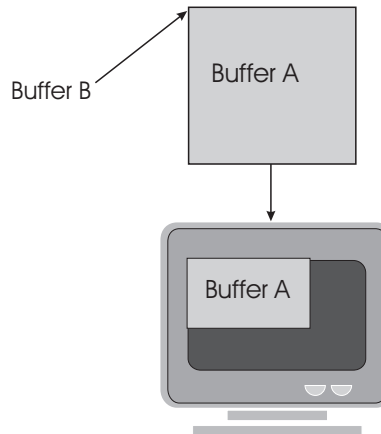
■ Buffer 1: (Odd field)

- Size = 640 x 240 (i.e., half height)
- Pitch = 1280 (i.e., to skip to the next field)
- Address = Address A (i.e., first pixel of the first row)

■ Buffer 2: (Even field)

- Size = 640 x 240 (i.e., half height)
- Pitch = 1280 (i.e., to skip to the next field)
- Address = Address B (i.e., first pixel of the second row)

In general, MIL automatically issues a display update after a displayed buffer has been modified. However, if a buffer selected on the display is modified using a mapped buffer, its display is not updated until you notify it of the change using *MbufControl(...M_MODIFIED...)*.



See *Chapter 10: Data Manipulation with multiple systems* for another instance where creating buffers is useful.

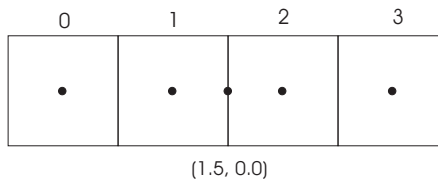
Pixel conventions

The center of a pixel is important for all MIL functions which return positional results with subpixel accuracy. The reference position of a pixel is its center, and the resulting subpixel coordinates are with respect to the pixel's center.

With this in mind, the coordinates of the center of an image can always be found using the following formula:

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

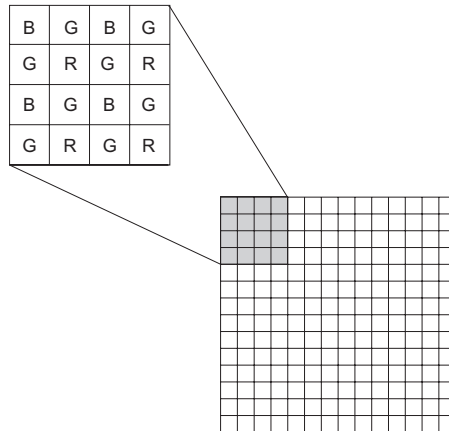
For example, the following image contains 4 pixels. If the formula is applied, the center of the image is found at (1.5, 0).



Using buffers with the Bayer color filter

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

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



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

Using MIL to convert the image

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

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

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

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

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

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

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

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

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

(cont...)

```

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

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

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

/* Allocate an array for the white balance coefficients. */
MilWBCoefficients = MbufAlloc1d(MilSystem, 3, 32+M_FLOAT, M_ARRAY, M_NULL);

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

MdispSelect(MilDisplay, MilImageDisp);

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

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

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

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

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

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

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

getchar();

```

(cont...)

```

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

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

getch();

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

getchar();

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

```

How the Bayer image gets converted

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

| | |
|---|---|
| G | B |
| R | G |

| | |
|---|---|
| B | G |
| G | R |

| | |
|---|---|
| G | R |
| B | G |

| | |
|---|---|
| R | G |
| G | B |

You must specify the pattern that is used by your camera when calling *MbufBayer()*. Since the green pixels are always diagonal to each other, specifying the starting two pixels of the pattern defines the pattern uniquely. Consult your camera's

documentation or contact the manufacturer if you are unsure; the Bayer image will not be converted properly if you specify the wrong pattern. If you cannot obtain information regarding the pattern of your camera, try all of *MbufBayer()*'s supported patterns to find the correct one.

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

| | | | | |
|-----------------|-----------------|-----------------|-----------------|---|
| B ₁ | G ₂ | B ₃ | G ₄ | : |
| G ₁₁ | R ₁₂ | G ₁₃ | R ₁₄ | : |
| B ₂₁ | G ₂₂ | B ₂₃ | G ₂₄ | : |
| G ₃₁ | R ₃₂ | G ₃₃ | R ₃₄ | : |
| : | : | : | : | : |

| | | | | |
|-----------------|-----------------|-----------------|-----------------|---|
| B ₁ | G ₂ | B ₃ | G ₄ | : |
| G ₁₁ | R ₁₂ | G ₁₃ | R ₁₄ | : |
| B ₂₁ | G ₂₂ | B ₂₃ | G ₂₄ | : |
| G ₃₁ | R ₃₂ | G ₃₃ | R ₃₄ | : |
| : | : | : | : | : |

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

| | | | | |
|-----------------|-----------------|-----------------|-----------------|---|
| B ₁ | G ₂ | B ₃ | G ₄ | : |
| G ₁₁ | R ₁₂ | G ₁₃ | R ₁₄ | : |
| B ₂₁ | G ₂₂ | B ₂₃ | G ₂₄ | : |
| G ₃₁ | R ₃₂ | G ₃₃ | R ₃₄ | : |
| : | : | : | : | : |

| | | | | |
|-----------------|-----------------|-----------------|-----------------|---|
| B ₁ | G ₂ | B ₃ | G ₄ | : |
| G ₁₁ | R ₁₂ | G ₁₃ | R ₁₄ | : |
| B ₂₁ | G ₂₂ | B ₂₃ | G ₂₄ | : |
| G ₃₁ | R ₃₂ | G ₃₃ | R ₃₄ | : |
| : | : | : | : | : |

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

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

White balancing your Bayer images

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

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

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

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

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

RGB images

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

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

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

YUV images

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

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

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

Monochrome images

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

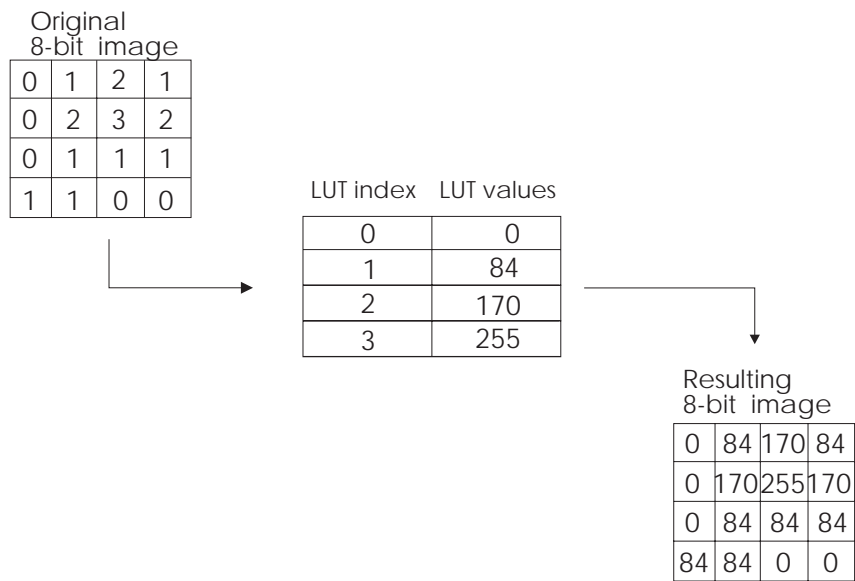
Note that if your image is grabbed in dark conditions, you might not only want to white balance your image, which will adjust the colors in your image, but you might also want to adjust the intensity. For monochrome and YUV images you can pass a greater value as the first element of the array if your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

Chapter 4: Lookup tables

This chapter describes lookup tables (LUTs). It shows you how to generate and modify them and briefly discusses how to use them.

Lookup tables

Lookup tables (LUTs) are collections of memory locations that are used to map data to pre-calculated values. They can easily reduce a multi-step or complex operation to a single-step LUT mapping.



If the hardware system permits, you can use LUTs to precondition input data at acquisition time, before it is stored in an image buffer. LUTs can also be used (hardware system permitting) to adjust the color contrast and intensity of an image upon display, without affecting the actual data.

LUTs and data buffers

LUT buffers

The MIL package represents LUTs as LUT data buffers. As with any other data buffer, LUT buffers must be allocated before they are used. A LUT buffer can be loaded, stored, or copied to another buffer (not necessarily to another LUT buffer) or to disk. You can also allocate child LUT buffers. When a LUT buffer is no longer required, you should free its memory space, using *MbufFree()*.

Allocating LUT buffers

LUT buffers are typically one-dimensional data buffers created with *MbufAlloc1d()* (single row). However, you can allocate a color RGB LUT, using *MbufAllocColor()*. In this case, set the number of bands to 3 (for RGB), the y-dimension to 1, and the x-dimension to have enough entries to represent the full range of possible values of the image buffer.

Loading and generating data into LUTs

With MIL, you can generate data directly into a LUT buffer or calculate the data and then load it in a LUT buffer.

Generating data directly into the LUT buffer

You can generate general data directly into a LUT buffer, using *MgenLutRamp()* or *MgenLutFunction()*.

The *MgenLutRamp()* command generates a value for each LUT index within the specified index range. The difference between the start value and the end value divided by the number of entries specified by the index range produces the increment. The increment is then used to generate the remaining entries of the index range.

If the increment is positive, *MgenLutRamp()* generates a ramp. If the increment is negative, the command generates an inverse ramp. If the increment is equal to zero, it loads the entire LUT range with the given start value.

The *MgenLutFunction()* command generates a value for each LUT index within the specified index range according to a specified mathematical function. The functions available are: M_LOG, M_EXP, M_SIN, M_COS, M_TAN, and M_QUAD. The specified start value is used as the initial X value in the equation. The remaining entries of the index range are generated by incrementing the value of X by 1 for each index.

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

Color LUTs

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

To load each color band with different data, you would have to generate the data into three separate one-dimensional LUT buffers, then copy each buffer to the appropriate color band of the color LUT buffer, using *MbufCopyColor()*.

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

Loading LUTs with precalculated data

More complex LUTs

There are several ways to generate more complex LUTs. Most of these, however, involve pre-calculating the data, then loading it into the LUT buffer:

- Calculate data, using your Host system, and then load it into the LUT, using *MbufPut()*, *MbufPut1d()*, or *MbufPutColor()*.
- Generate data into another data buffer, using MIL commands other than *MgenLutRamp()*, then copy the data to the LUT buffer, using *MbufCopy()* or *MbufCopyColor()*.
- Load previously saved LUT data from disk to the LUT buffer (*MbufLoad()*). Note, when loading data from disk, there should be enough data for each dimension of the LUT buffer.
- Restore a previously saved LUT, using *MbufRestore()*. Note, this command actually performs the LUT buffer allocation.

Using LUTs

In MIL, LUTs can be used in different circumstances:

- when displaying data (if supported by hardware)
- when acquiring data from a digitizer (if supported by hardware)

In each of these cases, if you want only a certain portion or palette of the LUT to be used, allocate the palette as a child buffer, and then specify the child LUT buffer identifier instead of its parent.

Refer to the documentation accompanying your target system device to determine under what circumstances it supports LUTs.

Displaying using LUTs

When you want to map a displayable image buffer through a LUT prior to displaying it, you need to associate the LUT buffer with the display, using *MdispLut()*. If this feature is supported by the hardware, it allows you to adjust the color contrast and intensity upon display without affecting the actual image data in memory.

The LUT buffer must match the pixel depth, and should either have the same number of color bands as the display or have a single color band. In the case of a single band, the same data is loaded into each of the display color LUTs.

Monochromatic effect

If you associate a one-band LUT buffer with a display, the same data is loaded in each output channel LUT, and the same data is routed to each output channel LUT. This produces a monochromatic effect when displaying a single-band image.

Pseudo-color effect

If you associate a three-band color LUT buffer (RGB) with a display, each LUT buffer color band is loaded in the corresponding output channel LUT. When displaying a single-band image, the same data is sent to each LUT. This produces a pseudo-color effect on the display.

True color effect

As mentioned above, if you associate a one-band LUT buffer with a display, the same LUT buffer data is loaded in each of the available output channel LUTs upon display. Although the same LUT values are used, you obtain a true color effect upon display of a color image because, typically, each image color band does not contain the same data. You generally want this image and LUT configuration when performing gamma correction to compensate for your monitor.

Finally, as is expected, associating a three-band color LUT with a display creates a true-color effect upon display of a color image.

Displaying image buffers with an associated LUT is further discussed in *Chapter 5: Displaying an image*.

LUTs and digitizers

Associating a LUT to a digitizer

Using MIL, you can map data from a digitizer through LUTs during image acquisition (if the device supports a LUT). This requires that you associate the LUT to the digitizer, using *MdigLut()*. The LUT buffer must match the pixel depth of the device. In addition, it should either have the same number of color bands as the digitizer or have a single color band.

Chapter 5: Displaying an image

This chapter discusses the display of image buffers, in detail. It shows you how to display several images simultaneously, and discusses some of the special effects that can be applied to a displayable image buffer.

Displaying an image

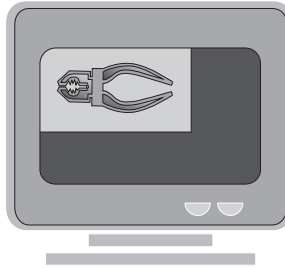
Whether or not your imaging board has a display section, MIL can display images. It will use the most appropriate graphics controller in your computer for display purposes. If your imaging board has a display section, and it is available, MIL will typically use it for display purposes.

Displayable image buffers

To display an image buffer, the buffer must have been allocated with a displayable attribute (`M_DISP`). In addition, a display must have been allocated using *MdispAlloc()* or *MappAllocDefault()*. Note that the buffer and the display should be allocated on the same system.

Selecting a buffer for display

Once a buffer and a display have been allocated, use *MdispSelect()* to select the image buffer to display. The buffer is displayed in a dedicated window or at the top-left corner of an auxiliary screen. If the specified image buffer is smaller in size than the display, the border outside the buffer is blacked out. If the specified image buffer is larger in size than the display, the right and bottom part of the buffer, the part that exceeds the display size, is not displayed.



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

Frame buffer

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

Types of displays

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

- With a windowed border, which is called a **windowed display** (M_WINDOWED).
- Without a windowed border on a dedicated screen, which is called an **auxiliary display** (M_AUXILIARY).

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

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

Windowed display

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

Extended desktop

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

All windowed displays (M_WINDOWED) are displayed in their own MIL default window (or, as will be seen later, in a user-allocated window). This window is transparently tracked

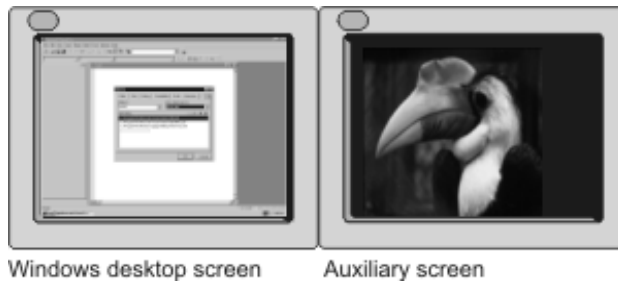
and updated with the image buffer selected to the display; that is, if the window moves or is occluded, the window is automatically updated with the image buffer accordingly.

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

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

Auxiliary display

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



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

Auxiliary displays are supported with the following minimum resources:

- Two graphics controllers.
- One DualHead graphics controller that integrates two CRT controllers (for example, Matrox G400, G450, or G550).
- ❖ Note that the graphics controller can be on, or apart from, the Matrox imaging board. Also, some boards might have special features or limitations regarding auxiliary displays; please see the *MIL/MIL-Lite Board Specific manual*.

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

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

Video output format

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

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

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

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

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

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

Windows NT

Under Windows NT, if the auxiliary screen is driven by the same board as the Windows desktop screen, the desktop's resolution must be larger than the resolution of the auxiliary display. For example, to allocate an auxiliary display with a resolution of 1024x768x32, the Windows desktop's resolution must be at least 1152x864x32. Note that if the Windows desktop screen and the auxiliary screen are controlled by two separate graphics controllers that have their own frame buffers, this restriction does not apply.

Matrox Millennium G400, G450, G550

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

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

Display number

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

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

Display size and depth

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

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

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

Displaying buffers of different data depths

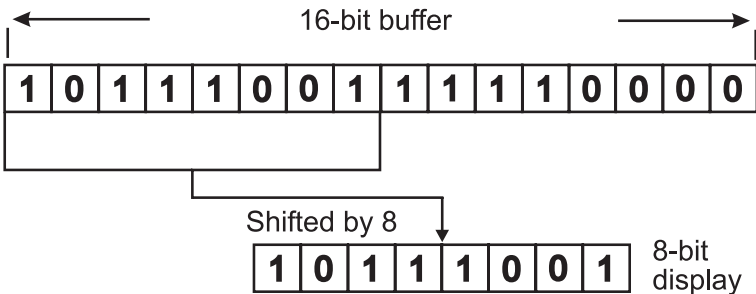
Displayable image buffers usually have a depth of 8-bits (or 3-band 8-bits, in the case of color images). For windowed displays, you can display images of other depths (for example, 1-bit or 16-bit images). By using *MdispControl()* with the `M_VIEW_MODE` control type, you can control the way such buffers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

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

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

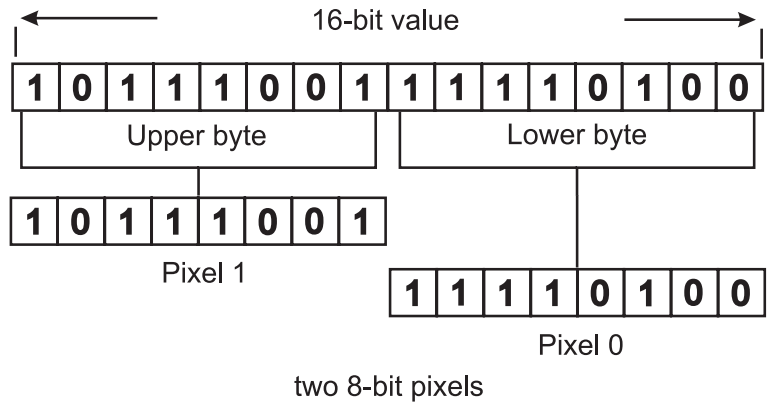
M_BIT_SHIFT

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



M_MULTI_BYTES

The `M_MULTI_BYTES` setting is primarily useful when grabbing from a multi-tap camera. This setting displays each byte of the image in separate display pixels. For instance, each pixel of a 16-bit image will occupy two consecutive display pixels; each pixel of a 32-bit image will occupy four consecutive display pixels.



M_DEFAULT

The default setting is `M_BIT_SHIFT`.

Removing a buffer from the display

After displaying an image buffer (with *MdispSelect()*), you can remove it from the display and close the associated window (for windowed displays), or leave the display blank (for auxiliary displays), using *MdispDeselect()*. To display a different image buffer, you are not required to remove the current buffer from the display; selecting another buffer for display automatically updates the display with the new buffer.

Once you have finished using a display, you should free it, using *MdispFree()*. If a displayed buffer is freed, the buffer is either automatically removed from the display (for windowed displays) or is left blank (for auxiliary displays).

Displaying multiple buffers

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

This is not the case for auxiliary displays, where you can only display one display at a time on a given auxiliary screen. However, you can still view more than one buffer at a time using child buffers. For example, you can display the source and destination buffers of an operation, using the following steps:

1. Allocate a large displayable buffer using *MbufAlloc2d()* or *MbufAllocColor()*. This buffer will be known as the parent buffer.
2. Allocate two non-overlapping child buffers within it, using *MbufChild2d()* or *MbufChildColor()*.
3. Select the parent buffer for display using *MdispSelect()*.
4. Use one of the child buffers as the source image buffer and the other as a destination image buffer of the operation.

The following portion of MIL code shows how to display multiple buffers in a single display. The required portion of the cell image, *cell.mim*, is loaded into a child of a displayable buffer and then text is written into it. The result is stored in another child of the same displayable buffer.

```

/* File name: mmultdis.c
 * Synopsis: This program shows how to display more than one image buffer at
 *           a time. It allocates a displayable image buffer, allocates two
 *           child buffers from it, and then uses these child buffers as the
 *           source and destination of a copy operation. It then writes text
 *           in each of the child buffers.
 *
 *           The display will be zoomed if the system's display supports it.
 */

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

/* MIL image file name. */
#define IMAGE_FILE        "cell.mim"

/* MIL image file specifications. */
#define IMAGE_WIDTH        128L
#define IMAGE_HEIGHT       240L
#define IMAGE_TYPE         8L+M_UNSIGNED
#define ZOOM_VALUE        2L

void main(void)
{
    MIL_ID MilApplication, /* Application identifier. */
    MilSystem,             /* System identifier. */
    MilDisplay,            /* Display identifier. */
    MilParentImage,        /* Image buffer identifier. */
    MilSrcSubImage,        /* Source image buffer identifier. */
    MilDstSubImage;        /* Destination image buffer identifier. */

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

    /* Allocate a display image buffer. */
    MbufAlloc2d(MilSystem, IMAGE_WIDTH*2, IMAGE_HEIGHT, IMAGE_TYPE,
                M_IMAGE+M_DISP+M_PROC, &MilParentImage);
    /* Allocate two child buffers from the displayable parent buffer. */
    MbufChild2d(MilParentImage, 0L, 0L, IMAGE_WIDTH, IMAGE_HEIGHT,
                &MilSrcSubImage);
    MbufChild2d(MilParentImage, IMAGE_WIDTH, 0L, IMAGE_WIDTH, IMAGE_HEIGHT,
                &MilDstSubImage);

```

(cont...)

```

/* Clear the parent buffer. */
MbufClear(MilParentImage, 0L);

/* Display the parent buffer. */
MdispSelect(MilDisplay, MilParentImage);

/* Load the entire source image into the source sub-image buffer. */
MbufLoad(IMAGE_FILE, MilSrcSubImage);

/* Copy the source sub-image into the destination sub-image */
MbufCopy(MilSrcSubImage, MilDstSubImage);

/* Write text in both sub-images */
MgraText(M_DEFAULT, MilSrcSubImage, IMAGE_WIDTH/4, IMAGE_HEIGHT/4,
"Source");
MgraText(M_DEFAULT, MilDstSubImage, IMAGE_WIDTH/8, IMAGE_HEIGHT/4,
"Destination");

/* Report on the Host screen what is being displayed. */
printf("A copy was performed between the sub-image on the\n");
printf("left side of the screen and the sub-image on the right side\n");
printf("of the screen and text was written into each of them.\n");
printf("Press <Enter> to continue.\n\n");
getchar();

/* Report on the Host screen what is being displayed. */
printf("Display zoomed by %ld in X and Y (if supported).\n", ZOOM_VALUE);

/* Zoom both sub-images by zooming the display. */
MdispZoom(MilDisplay, ZOOM_VALUE, ZOOM_VALUE);

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

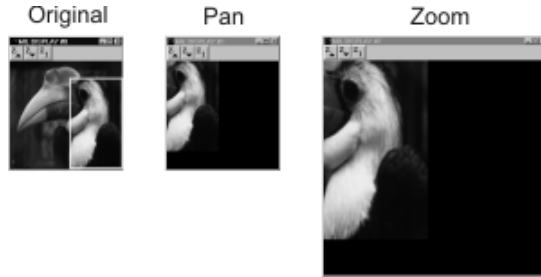
/* Close the display. */
MdispDeselect(MilDisplay, MilParentImage);

/* Free all allocations. */
MbufFree(MilDstSubImage);
MbufFree(MilSrcSubImage);
MbufFree(MilParentImage);
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);
}

```

Panning, scrolling, and zooming

At times, your image buffer might be larger than the display, or have details that are too fine or too small to see. Display effects can be associated with the display to view specific parts of the image. These effects are panning, scrolling, and zooming.



Panning and scrolling

Panning and scrolling displace an image horizontally or vertically, respectively, on the display. You can pan and scroll your image to display the appropriate location at the top-left corner of the window (for windowed displays) or screen (for auxiliary displays), using *MdispPan()*.

Zooming

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

```
MdispZoom(MilDisplay, 2, 2);
```

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

Image placement

For auxiliary displays, to view the image at another location on the display, you must create a large displayable image buffer, display it, and then allocate and use a child buffer at the required location on the display. For windowed displays, this is automatically handled.

Annotating the displayed image non-destructively

For either windowed displays or auxiliary displays, you can annotate the displayed image non-destructively using MIL's overlay-display mechanism. To make use of this functionality, do the following:

1. Enable the overlay-display mechanism using the following function call:

```
MdispControl(DisplayID, M_WINDOW_OVR_WRITE, M_ENABLE)
```

2. Select a buffer to the display:

```
MdispSelect(DisplayID, ImageBufId)
```

Since the overlay-display mechanism is enabled, this will not only display the selected image, but it will also associate a temporary overlay buffer with the display. This buffer is referred to as the **display's overlay buffer**. This overlay buffer can be used to annotate the underlying image with an effect called keying, which replaces pixels of one image that are of the specified keying color with the underlying areas of another image. Therefore, anything that you draw in this overlay buffer in a color other than the keying color, will annotate the image selected to the display.

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

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

4. Draw into the display's overlay buffer with the appropriate graphics function (*Mgra...()*). For example, to write text in the overlay buffer, use *MgraText()*. Note that since this temporary overlay buffer is a real buffer, any function (except grabbing) can be used.

- ❖ You can also annotate the displayed image buffer with Windows GDI annotations, which is discussed later.

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

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

Overlay buffer behavior

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

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

CPU-assisted overlay

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

Keying

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

If you are using an 8-bit display resolution (256 colors), set the keying color to a value between 0 and 255. If you are using a non 8-bit display resolution (15-bit, 16-bit, 24-bit, or 32-bit color resolution), call the macro `M_RGB888` and specify the RGB value. For example:

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

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

Using GDI annotations

If the display has been selected, you can also annotate the displayed image buffer with Windows GDI annotations. Use one of the following methods:

- Allocate a Windows display device context (DC) for drawing in the displayed image buffer. To do so, use *MbufControl()* with `M_WINDOW_DC_ALLOC`. Inquire the identifier of this context using *MbufInquire()* with `M_WINDOW_DC`. Then, use this DC with Windows GDI function calls.

The buffer which you are annotating must be internally stored in `M_DIB` or `M_DDRAW` format, and cannot be a child buffer.

You can create a DC for either the image buffer or the overlay buffer of the display. Note that if you create a DC for the image buffer and then draw using this DC, drawing will be destructive (that is, the data of the image buffer is actually changed).

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

- ❖ If using a `DDRAW` buffer, you must free the DC before signalling MIL.
- Inquire the display's window handle using *MdispInquire()* with `M_WINDOW_HANDLE`. Pass the window handle to the Windows *GetDC()* function to get a Windows display device context (DC). Then, paint the annotations with GDI functions

from a function hooked to the display update event (*MdispHookFunction()*); that is, paint each time the MIL display is modified.

Note that drawing using this method is non-destructive (that is, the actual data of the image buffer is not changed).

The following portion of MIL code shows the creation of the device context of the overlay buffer, the inquiring of the device context, and the drawing and writing in the overlay buffer (see also, *mdispovr.c*).

```
HDC  hCustomDC;
HPEN hpen, hpenOld;
char  chText[80];

/* Create a device context to draw in the overlay buffer with GDI. */
MbufControl(MilOverlayImage, M_WINDOW_DC_ALLOC, M_DEFAULT);

/* Inquire the device context. */
hCustomDC = ((HDC)MbufInquire(MilOverlayImage, M_WINDOW_DC, M_NULL));
if (hCustomDC)
{
    /* Create a blue pen. */
    hpen=CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
    hpenOld = SelectObject(hCustomDC,hpen);

    /* Draw a cross in the overlay buffer. */
    MoveToEx(hCustomDC,0,ImageHeight/2,NULL);
    LineTo(hCustomDC,ImageWidth,ImageHeight/2);
    MoveToEx(hCustomDC,ImageWidth/2,0,NULL);
    LineTo(hCustomDC,ImageWidth/2,ImageHeight);

    /* Write text in the overlay buffer. */
    strcpy(chText, "GDI Overlay Text ");
    SetTextColor(hCustomDC,RGB(0, 0, 255));
    TextOut(hCustomDC,ImageWidth*3/18,ImageHeight*4/6, chText,
            strlen(chText));
    SetTextColor(hCustomDC,RGB(255, 0, 0));
    TextOut(hCustomDC,ImageWidth*12/18,ImageHeight*4/6, chText,
            strlen(chText));

    /* Deselect and destroy the blue pen. */
    SelectObject(hCustomDC,hpenOld);
    DeleteObject(hpen);
}

/* Delete created device context. */
MbufControl(MilOverlayImage, M_WINDOW_DC_FREE, M_DEFAULT);

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

Displaying an image in a user-defined window

For windowed displays, images are automatically displayed in a default window created by MIL, using *MdispSelect()*. This function dynamically creates a window on the Windows desktop for the specified display, if the display is not already selected. The created window respects any window control that has been associated with the display using an *Mdisp...()* function.

Selecting a buffer into a specific display window

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

Using *MdispSelectWindow()*

The *MdispSelectWindow()* function is similar to *MdispSelect()*, except that it allows you to specify the handle of the user-defined window or child window to use for display, rather than displaying into a MIL created window. This user-defined window is automatically refreshed when the display is modified (for example, when the image data is modified). You can use *MdispDeselect()* to deselect the image from the display.

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

The following portion of MIL code from the *mwindisp.c* example shows how to display an image in a user-defined window, grab into such a window, and remove the image from the display.

```

/* File name: mwindisp.c
 *
 * Synopsis: This program displays a welcoming message in a user-
 *           defined window and grabs into it (if supported). It uses
 *           the MIL system and the MdispSelectWindow() function
 *           to display the MIL buffer in a user created client window.
 *
 *           Use MdispDeselect() to remove the selected image buffer
 *           from the display.
 */

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <windows.h>
#include <mil.h>
#include <mwinmil.h>
#include <wingdi.h>

#define BUFFERSIZE_X      640
#define BUFFERSIZE_Y      480
#define BUFFERSIZE_BAND   1
#define MAX_PATH_NAME_LEN 256

/* Prototypes */
void MilApplication(HWND UserWindowHandle);
void MilApplicationPaint(HWND UserWindowHandle);

/*****
 *
 * Name:      MilApplication()
 *
 * synopsis:  This function is the core of the MIL application that
 *            will be executed when the "Start" menu item of this
 *            Windows program will be selected. See WinMain() below
 *            for the program entry point.
 *
 *            It will use MIL to display a welcoming message in the
 *            specified user window and to grab in it if it is supported
 *            by the target system.
 */

```

(cont...)

```

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

    long BufSizeX;
    long BufSizeY;
    long BufSizeBand;

    /* Allocate a MIL application. */
    MappAlloc(M_DEFAULT, &MilApplication);

    /* Allocate a MIL system. */
    MsysAlloc(M_DEF_SYSTEM_TYPE, M_DEVO, M_DEFAULT, &MilSystem);

    /* Allocate a MIL display. */
    MdispAlloc(MilSystem, M_DEVO, M_DEF_DISPLAY_FORMAT, M_DEFAULT
    &MilDisplay);

    /* Allocate a MIL digitizer if supported and sets the target image size.*/
    if (MsysInquire(MilSystem, M_DIGITIZER_NUM, M_NULL) > 0)
    {
        MdigAlloc(MilSystem, M_DEVO, M_DEF_DIGITIZER_FORMAT, M_DEFAULT,
        &MilDigitizer);
        MdigInquire(MilDigitizer, M_SIZE_X, &BufSizeX);
        MdigInquire(MilDigitizer, M_SIZE_Y, &BufSizeY);
        MdigInquire(MilDigitizer, M_SIZE_BAND, &BufSizeBand);
    }
    else
    {
        MilDigitizer = M_NULL;
        BufSizeX = BUFFERSIZE_X;
        BufSizeY = BUFFERSIZE_Y;
        BufSizeBand = BUFFERSIZE_BAND;
    }

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

```

(cont...)

```

/* Allocate a MIL buffer. */
MbufAllocColor(MilSystem, BufSizeBand, BufSizeX, BufSizeY, 8+M_UNSIGNED,
(MilDigitizer? M_IMAGE+M_DISP+M_GRAB : M_IMAGE+M_DISP), &MilImage);

/* Clear the buffer */
MbufClear(MilImage,0);

/* Select the MIL buffer to be displayed in the user specified window */
MdispSelectWindow(MilDisplay, MilImage, UserWindowHandle);

/* Print a string in the image buffer using MIL.
 * Note: After a MIL command writing in a MIL buffer, the display
 * will automatically update the window given to MdispSelectWindow().
 */

MgraText(M_DEFAULT, MilImage, (BufSizeX/8)*3, BufSizeY/2,
" ..... ");
MgraText(M_DEFAULT, MilImage, (BufSizeX/8)*3, BufSizeY/2+25,
" Welcome to MIL !!! ");
MgraText(M_DEFAULT, MilImage, (BufSizeX/8)*3, BufSizeY/2+50,
" ..... ");

/* Windows code to open a message box to wait a key. */
MessageBox(0, ""Welcome to MIL !!!" was printed",
"MIL application example",
MB_APPLMODAL | MB_ICONEXCLAMATION );

/* Grab in the user window if supported by the system. */
if (MilDigitizer)
{
    /* Grab continuously. */
    MdigGrabContinuous(MilDigitizer, MilImage);

    /* Windows code to open a message box to wait a key. */
    MessageBox(0, "Continuous grab in progress",
"MIL application example",
MB_APPLMODAL | MB_ICONEXCLAMATION );

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

/* Deselect the MIL buffer from the display. */
MdispDeselect(MilDisplay, MilImage);

/* Free allocated objects. */
MbufFree(MilImage);

end:

MdispFree(MilDisplay);
if (MilDigitizer)
    MdigFree(MilDigitizer);
MsysFree(MilSystem);
MappFree(MilApplication);
}

```

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

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

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

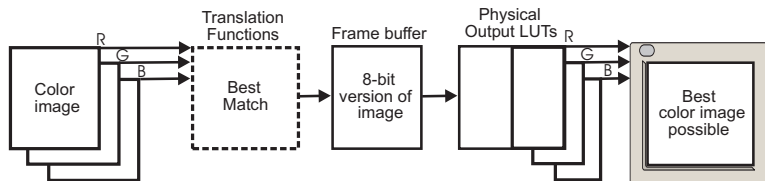
Reference material: Windows palettes and physical output LUTs

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

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

Color images

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

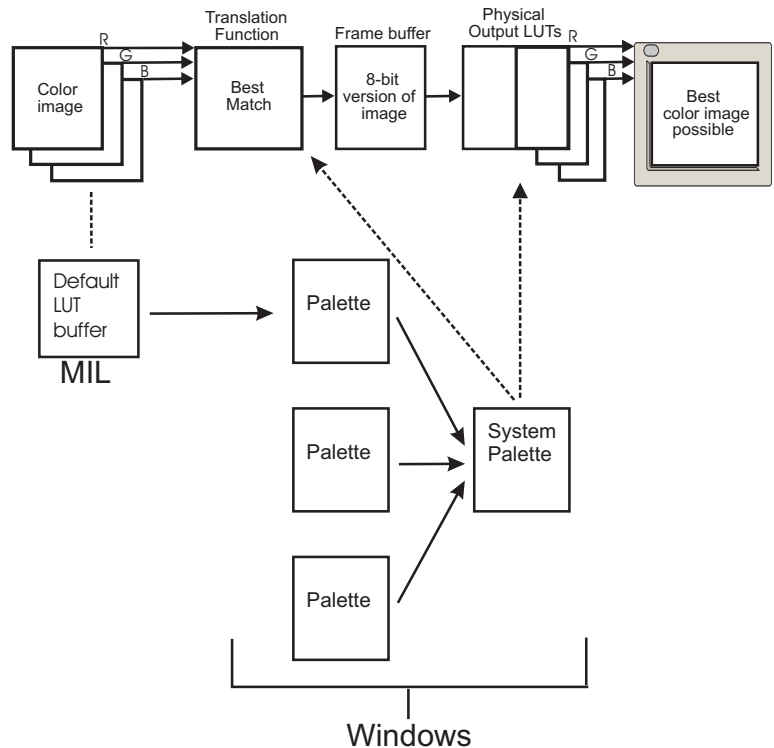


1. The color image goes through Windows translation functions. Windows searches the physical output LUTs for the entry that best matches the color of the image's pixels. Depending on what colors are available, Windows translates the image to an 8-bit index image.

2. The physical output LUTs subsequently translate that index image and display it as the best color image possible.

Palettes and physical output LUTs

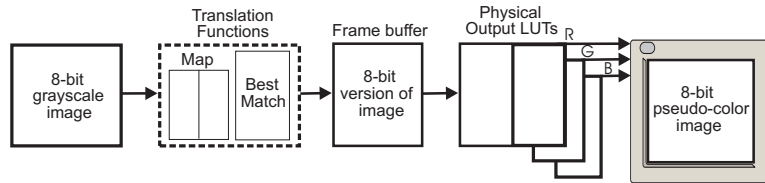
Each of the color images' palette is used to specify the required colors. Windows uses these palettes to realize a system palette. The system palette is ultimately loaded into the physical output LUTs and is searched by the translation functions when generating an 8-bit (index) image.



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

Monochrome images

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



Default palette settings

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

Why change the default LUT values

If the default LUT values are not appropriate for your application, you can change the LUT values to control the displayed colors or gray levels of an image. Some situations that might require special display effects are:

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

Changing the default LUT values

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

Viewing each gray intensity in a different color

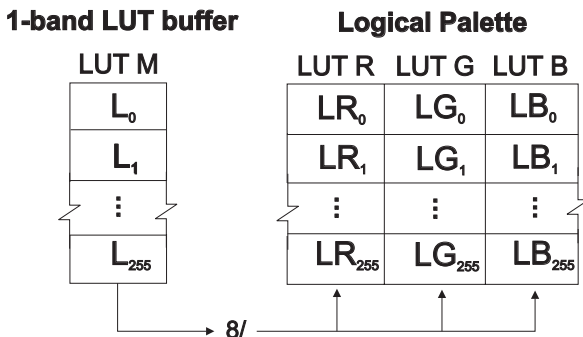
To view an 8-bit image buffer with each gray intensity in a different color, associate the default pseudo-color LUT buffer (`M_PSEUDO`) with the display of the image. In this case, the data is loaded in each component of the logical palette.

A 1-band custom LUT buffer

To invert the values of an 8-bit image on display, you would need physical output LUTs that map each value to the maximum pixel value minus the current pixel value. To do so:

1. Allocate a 1-band 256-entry LUT buffer using *MbufAlloc1d()*.
2. Generate the data into the buffer using *MgenLutRamp()*, or load the data into it, using *MbufPut()*. The depth of the LUT buffer data must be 8 bits.
3. Associate the LUT buffer with the required display using *MdispLut()*, or to a particular image using *MbufControl()* with `M_ASSOCIATED_LUT`.

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



A 3-band custom LUT buffer

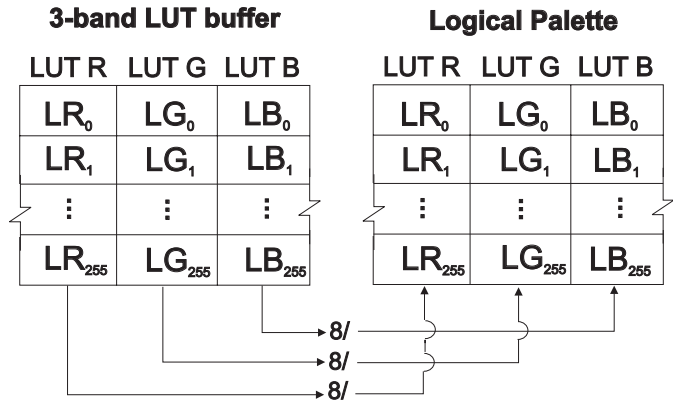
To reduce the loss of color resolution when displaying an image with a specific range of colors, you would need physical output LUTs that contain all the required colors so that, when Windows creates a translation table for the image, most colors are mapped to their exact values. To do so, allocate a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

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

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

3. Associate the LUT buffer with the required display using *MdispLut()*, or to a particular image using *MbufControl()* with `M_ASSOCIATED_LUT`.

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



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

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

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

You can use *MdispInquire()* to obtain information about the physical output LUTs of a display.

CPU-assisted display

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

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

- Overlay flickering.
- Pseudo-live continuous grabbing.

Overlay flickering

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

Grabbing continuously

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

Avoiding CPU-assisted displays

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

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

Chapter 6: Generating graphics

This chapter describes the graphics commands that are available with MIL. These consist of drawing and text-writing commands.

MIL and graphics

The MIL package supports basic drawing and text commands that are useful in typical image processing or machine vision applications. These commands could be used, for example, to create a conditional buffer or to annotate an image.

Preparing for graphics

There are two requirements for graphics operations:

- An image buffer in which to perform the operation.
- A set of graphics parameters, referred to as a graphics context, with which to perform the operation.

Graphics context

Allocate a graphics context, using *MgraAlloc()*. Upon allocation, each of the graphics parameters of the graphics context is set to the default (refer to the *MgraAlloc()* command reference description for the defaults). You can change these parameter settings according to your needs.

Different graphics contexts can coexist. Use their identifier to specify which to use or change.

Once a graphics context is no longer required, it should be freed, using *MgraFree()*.

When a MIL application is created, using *MappAlloc()* or *MappAllocDefault()*, a default graphics context is automatically created. It can be used as a normal graphics context by specifying `M_DEFAULT` as the graphics context identifier. Since `M_DEFAULT` is simply another graphics context, you can change its parameter settings according to your needs.

Graphics parameters

There are two basic parameters that apply to graphic objects:

1. **Background color.** This determines the background color of textual graphic objects. The default background color value is zero (typically corresponds to black). You can change this color, using *MgraBackColor()*.
2. **Foreground color.** This determines the color in which graphic objects are drawn or written. The default foreground color value is the highest positive buffer value (typically corresponds to white). You can change this color, using *MgraColor()*.

Selecting colors

A grayscale value can be any integer or floating-point number. If the given value exceeds the range of the possible values that can be stored in each band of the destination buffer, the least significant bits of the value are used.

Clearing the buffer

Once you are satisfied with the graphics parameters, you should determine whether you need to clear the graphics image buffer prior to drawing or writing to it. You can use *MgraClear()* or *MbufClear()* to clear the buffer to a specific color.

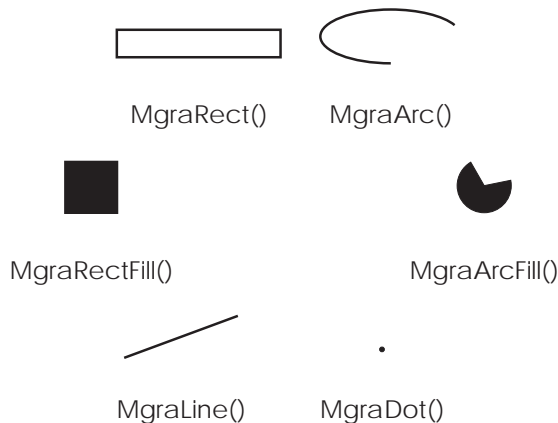
Drawing graphics

With the MIL package, you can draw:

- lines (*MgraLine()*)
- rectangles (*MgraRect()* and *MgraRectFill()*)
- arcs, circles, and ellipses (*MgraArc()* and *MgraArcFill()*)
- dots (*MgraDot()*)

Using *MgraLine()*, *MgraRect()*, *MgraArc()*, or *MgraDot()*, you can draw the outline of most required shapes. The outlines are drawn one pixel wide.

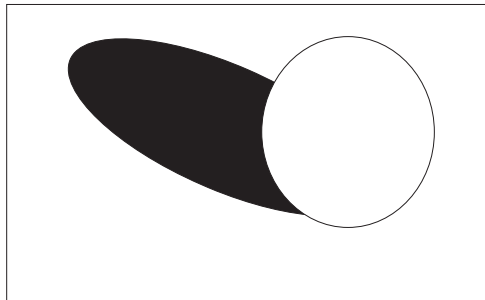
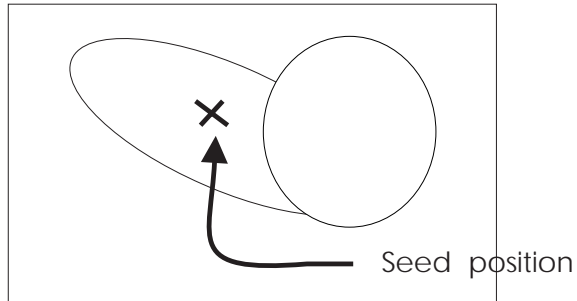
In addition, the MIL package includes *MgraRectFill()* and *MgraArcFill()* so you can draw solid rectangles and arcs.



If you need complex filled-in shapes, draw the outline of the shape and use *MgraFill()* to fill it.

Filling shapes

MgraFill() performs a boundary-type seed fill. It fills an area of the target buffer with the current foreground color, starting from the specified seed position. Filling occurs on adjacent pixels of the same value as the original seed pixel.



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

Writing text

You can also write text in the drawing area, using *MgraText()*. This command writes a null-terminated (`\0`) ASCII string at the specified position in a given buffer, using the foreground and background color and current font of the specified graphics context.

When specifying the location at which to write the string, give the top-left corner coordinates of the first character in the string.

(*x*, *y*)



| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|--|--|--|
| G | o | o | o | o | r | r | r | n | n | n | ! | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|--|--|--|

Although the graphics context specifies a default character font and size, you can change the font and size of this context, using *MgraFont()* and *MgraFontScale()*, respectively. *MgraFont()* provides a set of predefined fonts from which to choose.

Chapter 7: Grabbing with your digitizer

This chapter discusses the cameras supported with MIL; it also discusses the control of your digitizers, including the fine-tuning of the input, and auto-focusing.

Cameras and input devices

The MIL package supports input from any type of input device supported by the digitizer. Data grabbed from an input device through the digitizer, using *MdigGrab()* or *MdigGrabContinuous()*, is stored into an image buffer. Note, since most input devices are cameras, they will hereafter be referred to as such.

For a digitizer to be recognized by MIL, it must be allocated on the target system, using *MdigAlloc()* (or *MappAllocDefault()*). The allocation sets up the digitizer to match your camera's data format and to access the active input channel. Once you have finished using a digitizer, you should free it, using *MdigFree()*.

If you often use the same camera and prefer to use *MappAllocDefault()* to set up and initialize your system, you might want to update the *milsetup.h* file to reflect your camera.

When developing an application, it is recommended that you use a simple camera. Once the application is working, switch to a more sophisticated camera, if necessary. This approach makes debugging much easier.

The data format

MdigAlloc() needs the camera's digitizer configuration format (DCF) to perform the digitizer allocation. The DCF defines such parameters as the input frequency and resolution, and will determine limits when grabbing an image.

MIL provides a number of predefined DCFs for the basic cameras supported by your digitizer. Refer to the *MIL/MIL-Lite Board-specific notes* manual for exact settings. MIL also provides some DCF files that you can load if the predefined DCFs don't suit your needs.

Once a digitizer has been allocated, you can use *MdigInquire()* to inquire about its settings.

If you find a DCF file that is appropriate for your camera (video source), but need to adjust some of the more common settings, you can do so directly, without adjusting the file, using the *Mdig...()* commands. For more specialized adjustments, you can adjust the file itself, using Matrox Intellicam.

If you cannot find an appropriate DCF file because, perhaps, you have a non-standard video source (such as a strobe or trigger device), you can create your own DCF file, using Matrox Intellicam. For more information on Matrox Intellicam, refer to the *Matrox Intellicam User Guide* manual.

If you cannot develop the required DCF using Matrox Intellicam, you should provide the camera specifications to your Matrox Technical Support Engineer. A suitable customized DCF file can then be developed, if your digitizer supports the camera.

The digitizer number

In addition to the data format, *MdigAlloc()* requires that you specify the digitizer number. The digitizer number specifies the required digitizer, and its rank with respect to other digitizers of the same type (color or monochrome) residing in the same system. Note, if there is only one digitizer in the specified system, you must specify the digitizer number as `M_DEV0` or `M_DEFAULT`.

Multiple cameras

MIL also supports applications that require input from different cameras. In general, you cannot simultaneously activate two cameras, whether or not they are connected to the same digitizer.

The input channel

Most digitizers have several multiplexed input channels. This means that they have several channels but can only grab from one of the channels at a time. In this case, if you have a camera that is not connected to the first channel of its digitizer, you must specify the channel, using *MdigChannel()*.

If there are several cameras of the same data format connected to a digitizer, you only need to allocate a digitizer with the DCF of the first camera and use *MdigChannel()* to switch between the others of the same type.

When using different cameras connected to the same digitizer, a different DCF must be used for each camera. In general, to switch between cameras of different formats, you have to allocate the digitizer with one format, grab, free the digitizer, and then allocate the digitizer again with the second format. Some systems permit virtual digitizers (for example, Matrox Corona-II) so that you can allocate several digitizers, specify a channel for each digitizer, and then grab with the appropriate digitizer, without having to free and re-allocate between switches.

Grabbing a single field

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

Line-scan cameras

If your target digitizer supports it, you can grab from a line-scan camera as you would, for example, an RS-170 type camera. However, you should be aware of how data from these cameras is stored.

When acquiring data from a line-scan camera, each line of each destination buffer band is filled from top to bottom. The operation will only end once the entire buffer has been filled.

Grabbing to the display

Live and pseudo-live continuous grabs

With MIL, you can grab to a displayable buffer selected on a display. MIL uses one of two methods to transfer when grabbing:

- **Live grab.** MIL grabs directly to the version of the buffer that is physically allocated in the frame buffers (display memory).
- **Pseudo-live grab.** MIL grabs into the Host memory version of the buffer and then updates the version in the frame buffers (display memory).

See the *Attribute* section in *Chapter 3: Specifying and managing your data buffers* for more information on displayable buffers.

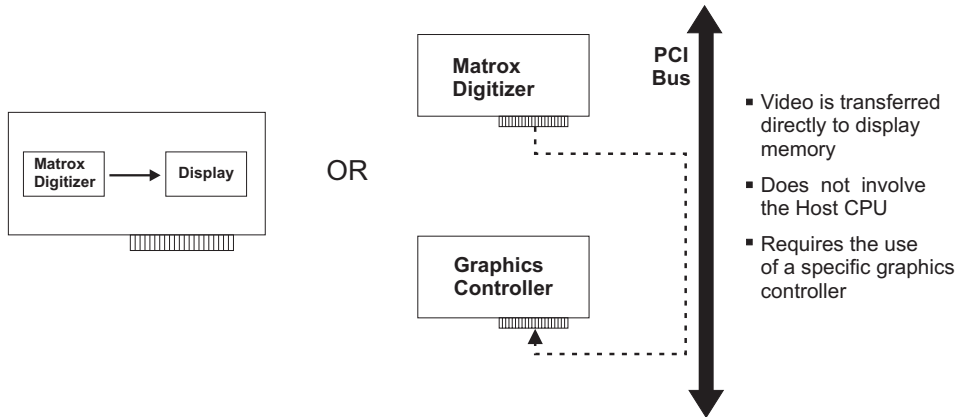
When grabbing, the digitizer (for example, Matrox Meteor-II) always acts as the bus master.

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

By default, at the end of a continuous grab (live or pseudo-live), a copy of the last image grabbed is made in the Host memory version of the buffer (or on-board processing memory). This allows the image to be processed. You can override the copy-to-Host behavior, using *MsysControl()* with the `M_LAST_GRAB_IN_TRUE_BUFFER` control type. Note that in this case, the *MdigGrabContinuous()* call will not modify the Host buffer in any way.

Live transfer to the display

The digitizer can generally transfer all grabbed data directly to display memory, when grabbing to an on-board display or when grabbing to a graphics controller that supports fast linear-memory accesses to its frame buffer.

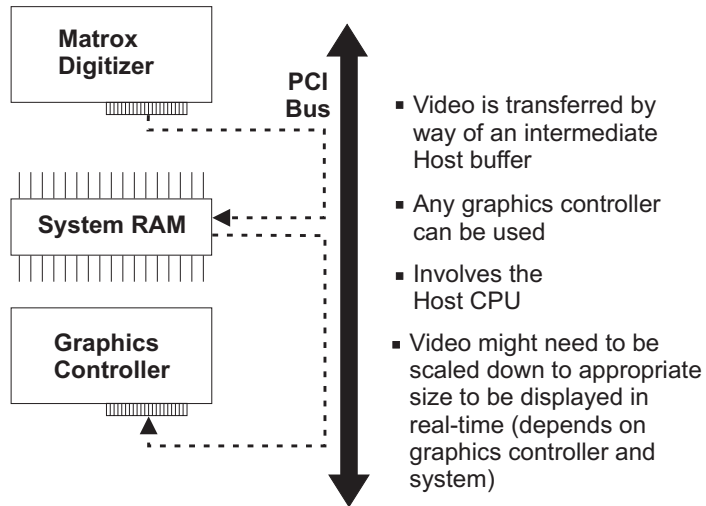


Pseudo-live transfers to the display

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

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

MIL transparently performs pseudo-live grabs:



By default, when a continuous grab switches to pseudo-live, it will transparently double buffer the grab in Host memory. That is, while the digitizer is grabbing one frame into a Host buffer, the display driver performs a blit of the previous frame (stored in the temporary Host buffer) to the frame buffers (graphics controller's display memory). Double-buffering can be disabled using *MsysControl()* with `M_DISPLAY_DOUBLE_BUFFERING`.

Pseudo-live transfers will be real time (that is, full frame rate of 30 for NTSC or 25 fps for PAL) if the CPU transfer from the Host buffer to display memory is fast enough; that is, if the blit is taking at most one frame for its length of time. Blit time is affected by the load of the CPU (for example, the number of process threads and the priorities of other boards).

You can reduce the load of the CPU in the pseudo-live grab operation by disabling the double buffering operation. However, when double buffering is disabled, only half of the full frame rate can be achieved.

Windowed displays

For windowed displays, note that a continuous grab without overlay can be moved from one screen to another and be displayed live when it has the focus. However, when the window displaying the grab intersects two screens, the grab is pseudo-live.

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

Using an MGA graphics controller

Matrox recommends using Matrox MGA boards for real-time display of video data. Selection of an MGA board depends on your application's requirements. To find out more about display mode resolutions on a particular board, see the *MIL/MIL-Lite Board-specific notes* manual.

Using a graphics controller other than MGA

If your graphics controller is not an MGA board, you must reconfigure the [Vga] section in the *mil.ini* file.

The following is an example of a *mil.ini* configuration file, describing the Matrox MGA Millennium-II PCI board (contact your graphics controller vendor for this information). The Matrox vendor identifier is 102B, the MGA Millennium-II device identifier is 051B, the frame buffer is mapped to an address, offset by 0 from its PCI base address of 0:

```
[Vga]
VgaVendorId=102B
VgaDeviceId=051B
VgaBaseAddressIndex=0
VgaBaseAddressOffset=0
```

Instead of specifying all of the above parameters, you can specify the graphics controller's physical address:

```
VgaPhysicalAddress=EF000000
```

If the live grab operation does not have the proper pitch or the proper pixel depth, the following optional entries must be specified:

```
VgaPitch=400
```

```
VgaFormat=M_RGB15+M_PACKED
```

❖ All values are hexadecimal.

The default location of the *mil.ini* file is the Windows directory under Microsoft Windows. A different location can be specified using the environment variable, MILINIDIR.

Screen Tearing

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



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

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

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

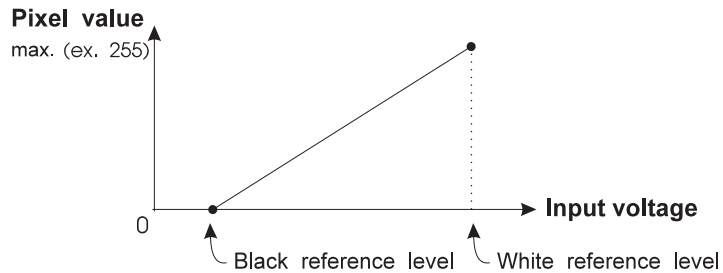
Reference levels, lookup tables, and scaling

MIL provides functions to improve the appearance of a grabbed image on input (if your hardware allows it). You can adjust the brightness and contrast of the images, as well as the hue and saturation for color grabs, by fine-tuning the controls of the analog-to-digital converters in your system. You can also correct and precondition the input data prior to storing it, through scaling, or by mapping it through an input LUT.

Black and white reference levels

When digitizing images, the black and white reference levels determine the zero and full-scale levels, respectively, of the input voltage range. The analog-to-digital converters convert any voltage above the white reference level to the maximum pixel value, and any voltage below the black reference level to a zero pixel value.

Matrox digitizers support fine-tuning of these reference levels. By reducing or increasing either or both the black and white reference levels, you affect the brightness of the image. By reducing one reference level and increasing the other, you affect the contrast of the image.



MIL linearly represents the distance between the minimum and maximum voltages, in which the black reference level can be adjusted (hardware-specific), as units between `M_MIN_LEVEL` and `M_MAX_LEVEL`. The same is done for the white reference level adjustment range. These units are the values by which you can adjust the specified reference level, using *MdigReference()*.

To calculate the value to pass to *MdigReference()*, use the following equation with the appropriate voltages specified in the *MIL/MIL-Lite Board-specific notes* manual for your particular board.

$$\text{Value to pass to } MdigReference() = \left(\frac{\text{Voltage needed} - \text{minimum voltage}}{\text{maximum voltage} - \text{minimum voltage}} \right) (M_MAX_LEVEL - M_MIN_LEVEL)$$

The smallest voltage increment supported by your board can differ such that consecutive reference-level settings might produce the same result.

Note, the new reference level might not take effect until the next grab, at which point, a certain amount of delay might be incurred as the hardware adjusts to the reference-level changes.

Color image reference levels

When grabbing composite color images, *MdigReference()* provides specific control parameters to adjust the levels of contrast, brightness, hue, and saturation. These levels can be set to values from 0 to 255. See the *MIL/MIL-Lite Board-specific notes* manual for your particular board for more details.

Mapping grabbed data through a LUT

You can correct or precondition input data by mapping it through a LUT when grabbing (if the hardware permits). This requires that you copy a LUT buffer to a digitizer's physical input LUT, using *MdigLut()*.

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

In addition, the LUT buffer's number of entries must match the digitizer's input data range.

To revert to the default LUT values, you must copy the default LUT (M_DEFAULT) to the digitizer. For digitizers, the default LUT is one that maps pixels to the same values. This type of LUT is typically referred to as a transparent LUT.

Scaling

The *MdigControl()* function allows you to scale grabbed data horizontally and vertically. If you scale grabbed data, the stored image size is different from the original image by the specified factors in the X and/or Y direction. The scaled image is written in contiguous locations in the image buffer, starting from the

top-left corner. For example, if you set both the X and Y scaling factors to 1/2, only one column and one row out of two are written to the image buffer.

| | | | | | | | | | | |
|---|-----|-----|-----|-----|-----|-----|-----|----|----|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 115 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 215 | 244 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 215 | 243 | 196 | 196 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 215 | 111 | 111 | 87 | 87 | 87 | 87 | 86 | 87 | |
| 0 | 0 | 0 | 111 | 115 | 87 | 87 | 87 | 87 | 0 | |
| 0 | 0 | 0 | 0 | 111 | 111 | 115 | 45 | 0 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 111 | 92 | 92 | 0 | 0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 111 | 111 | 0 | 0 | |



| | | | | |
|---|-----|-----|----|----|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 115 | 0 | 0 | 0 |
| 0 | 243 | 196 | 0 | 0 |
| 0 | 0 | 115 | 87 | 87 |
| 0 | 0 | 0 | 92 | 0 |

Subsampled image

X subsampling factor = $\frac{1}{2}$

Y subsampling factor = $\frac{1}{2}$

Original image

The X and Y scaling factors are independent. Note, depending on the digitizer and camera used, some scaling factors might not be available.

To disable scaling, set scaling factors to 1.

Optimizing application performance when grabbing

Grab mode

When grabbing data with *MdigGrab()*, you can control the synchronization by setting the *MdigControl()* `M_GRAB_MODE` control type to a value of `M_SYNCHRONOUS`, `M_ASYNCHRONOUS`, or `M_ASYNCHRONOUS_QUEUED` (if supported).

- If the grab mode is set to `M_SYNCHRONOUS`, your application will be synchronized with the end of a grab operation. In other words, your application will wait until the grab has finished before executing the next command.
- If the grab mode is set to `M_ASYNCHRONOUS`, your application will not be synchronized with the end of a grab operation. This option allows other commands to execute while still grabbing. This is a useful option when performing double buffering, a technique whereby you can grab data into one buffer while processing the previously grabbed buffer (discussed below). Note, a call to another *MdigGrab()* before the current grab has finished will cause your application to wait until the current grab has finished.
MdigGrabContinuous() is by definition asynchronous since you must use *MdigHalt()* to stop the grab.
- If your imaging board supports queuing, you can set the grab mode to `M_ASYNCHRONOUS_QUEUED`; if another grab is issued before the first one is finished, the grab will be queued on-board, allowing you to perform other processes while waiting for the next *MdigGrab()* to be executed. Note, you can still force your application to wait until the end of a grab before executing an operation, by calling *MdigGrabWait()*.

Double buffering

Double buffering involves grabbing into one image while processing the previously grabbed image. Double buffering allows you to grab and process concurrently. You must switch the destination of the grab between the two image buffers. In addition, you need to synchronize the grabbing and processing so that:

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

Below is an example (*mdbproc.c*) of how to perform double buffering:

```
/* File name: Mdbproc.c
 * This example does double buffered grab with real time processing.
 * Note: This assumes that the processing operation is shorter than a grab
 *       and that the PC has sufficient bandwidth to support the 2
 *       operations simultaneously. Also if the target processing buffer
 *       is not on the display, the processing speed is augmented.
 */

/* Image scale. */
#define IMAGE_SCALE 0.5

/* headers */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <mil.h>

/* Main function. */
void main(void)
{
    MIL_ID  MilApplication;
    MIL_ID  MilSystem      ;
    MIL_ID  MilDigitizer   ;
    MIL_ID  MilDisplay     ;
    MIL_ID  MilImage[2]    ;
    MIL_ID  MilImageDisp   ;

    long    NbProc = 0;
```

(cont...)

```

/* Allocations. */
MappAlloc(M_DEFAULT, &MilApplication);
MsysAlloc(M_DEF_SYSTEM_TYPE, M_DEF_SYSTEM_NUM, M_SETUP, &MilSystem);
MdigAlloc(MilSystem, M_DEFAULT, M_DEF_DIGITIZER_FORMAT, M_DEFAULT, &MilDigitizer);
MdispAlloc(MilSystem, M_DEFAULT, M_DEF_DISPLAY_FORMAT, M_DEFAULT, &MilDisplay);

/* Allocate 2 grab buffers. */
MbufAlloc2d(MilSystem,
            (long)(MdigInquire(MilDigitizer, M_SIZE_X, M_NULL)*IMAGE_SCALE),
            (long)(MdigInquire(MilDigitizer, M_SIZE_Y, M_NULL)*IMAGE_SCALE),
            8L+M_UNSIGNED,
            M_IMAGE+M_GRAB+M_PROC, &MilImage[0]);
MbufAlloc2d(MilSystem,
            (long)(MdigInquire(MilDigitizer, M_SIZE_X, M_NULL)*IMAGE_SCALE),
            (long)(MdigInquire(MilDigitizer, M_SIZE_Y, M_NULL)*IMAGE_SCALE),
            8L+M_UNSIGNED,
            M_IMAGE+M_GRAB+M_PROC, &MilImage[1]);

/* Allocate 1 displayable buffer and clear it. */
MbufAlloc2d(MilSystem,
            (long)(MdigInquire(MilDigitizer, M_SIZE_X, M_NULL)*IMAGE_SCALE),
            (long)(MdigInquire(MilDigitizer, M_SIZE_Y, M_NULL)*IMAGE_SCALE),
            8L+M_UNSIGNED,
            M_IMAGE+M_GRAB+M_PROC+M_DISP, &MilImageDisp);
MbufClear(MilImageDisp, 0x0);
.
.
.

/* Put the digitizer in asynchronous mode. */
MdigControl(MilDigitizer, M_GRAB_MODE, M_ASYNCHRONOUS);

/* Grab into the first buffer. */
MdigGrab(MilDigitizer, MilImage[0]);

/* Process one buffer while grabbing the other. */
while( !kbhit() )
{
    /* Grab second buffer while processing first buffer. */
    MdigGrab(MilDigitizer, MilImage[1]);
    .
    .
    .

```

(cont...)

```

    /* Process the first buffer already grabbed. */
    /* Note: Real time only if PC is fast enough. */
    MimConvolve(MilImage[0], MilImageDisp, M_EDGE_DETECT);
    .
    .
    .
    /* Grab first buffer while processing second buffer. */
    MdigGrab(MilDigitizer, MilImage[0])

    /* Process the second buffer already grabbed. */
    MimConvolve(MilImage[1], MilImageDisp, M_EDGE_DETECT);
}
.
.
.
/* Free allocations. */
MbufFree(MilImageDisp);
MbufFree(MilImage[0]);
MbufFree(MilImage[1]);
MdispFree(MilDisplay);
MdigFree(MilDigitizer);
MsysFree(MilSystem);
MappFree(MilApplication);
}

```

Multiple buffering

When an occasional frame takes longer to process than the time required to grab, you can use a multiple buffering technique to ensure that all processing is completed without losing any frames. To perform multiple buffering, use the *MdigHookFunction()*, when grabbing asynchronously, to hook the grab function to certain grab events, such as the start or end of a frame: the hooked function will interrupt the processing to perform the grab, and return to continue processing after the grab is initiated. You can grab into as many buffers as required to ensure that all processing is finished before overwriting a buffer with a new frame.

Note, processing is generally faster if the buffer is not on the display.

Grabbing a sequence of frames in real-time

To grab a sequence of frames in real-time, simply use successive, asynchronous calls to *MdigGrab()*:

```
/* Put digitizer in asynchronous mode */
MdigControl(MilDigitizer, M_GRAB_MODE, M_ASYNCHRONOUS);

/* Grab the sequence. */
for (n=0; n<NbFrames; n++)
{
    /* Grab one buffer at a time. */
    MdigGrab(MilDigitizer, MilImage[n]);
}
```

You must also allocate a buffer for each frame of the sequence. After you have grabbed a sequence, you can use the *MbufExportSequence()* function to export the sequence of image buffers (compressed or un-compressed 8-bit) to an AVI file. When exporting, you must specify the number of buffers and the frame rate (number of images/second) of the sequence. Note, the MIL identifiers of the image buffers to export must be kept in an array.

Use the *MbufImportSequence()* to import a sequence of images from an AVI file into separate image buffers. You can import compressed (MJPEG) or un-compressed 8-bit images. You can also choose to import the sequence into automatically allocated buffers or previously allocated buffers.

Grabbing with triggers and exposures

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

The camera's digitizer configuration format (DCF) file specifies whether or not to perform a triggered grab and exactly how it should be carried out. For example, if the DCF specifies that an

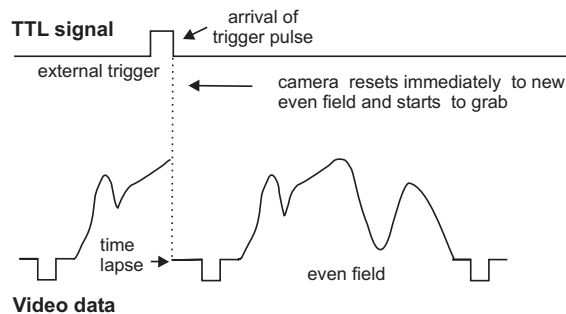
exposure signal should be generated (for the camera) upon the grab trigger event, the actual grab would only be triggered once the active exposure time was over.

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

Asynchronous reset mode

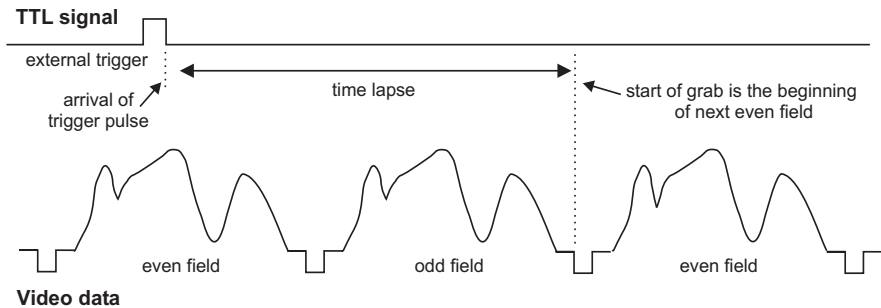
If your digitizer supports asynchronous reset mode, the digitizer resets the camera to begin a new frame when the trigger signal is received.

Asynchronous reset mode



Otherwise, the digitizer waits for the next valid frame (or field) before commencing to grab. The grab activation mode is specified in the DCF file.

Next valid frame (or field) mode



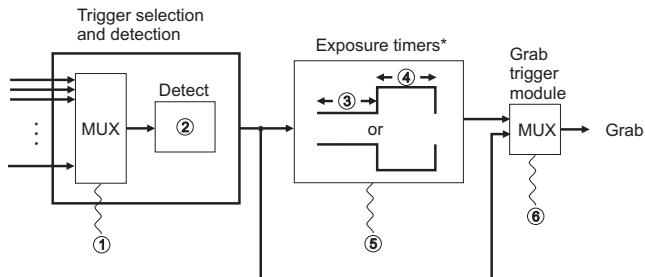
Triggers and exposures

In MIL, there are two methods of grabbing with triggers and exposures: the automatic exposure model and the manual bypass model. They are described in detail in the following diagrams. By default, MIL uses the automatic exposure model. You can change this default using **MdigControl()** with `M_GRAB_EXPOSURE_BYPASS`.

Automatic exposure model

In the automatic exposure model, the digitizer is configured to have the pipeline that is illustrated in the next diagram. (Note that the defines specified in the following illustration are those to be used with the **MdigControl()** function).

(M_GRAB_EXPOSURE_BYPASS set to M_DISABLE or M_DEFAULT)



- ① trigger source (M_GRAB_TRIGGER_SOURCE)
- ② trigger detection method (M_GRAB_TRIGGER_MODE)
- ③ exposure delay (M_GRAB_EXPOSURE_TIME_DELAY)
- ④ exposure time (M_GRAB_EXPOSURE_TIME)
- ⑤ polarity of exposure signal (M_GRAB_EXPOSURE_MODE)
- ⑥ bypass exposure timers if exposure time = 0 (M_GRAB_EXPOSURE_TIME)

* exposure timers will be cascaded automatically (if necessary) to generate one signal that has the required delay and active time

To summarize:

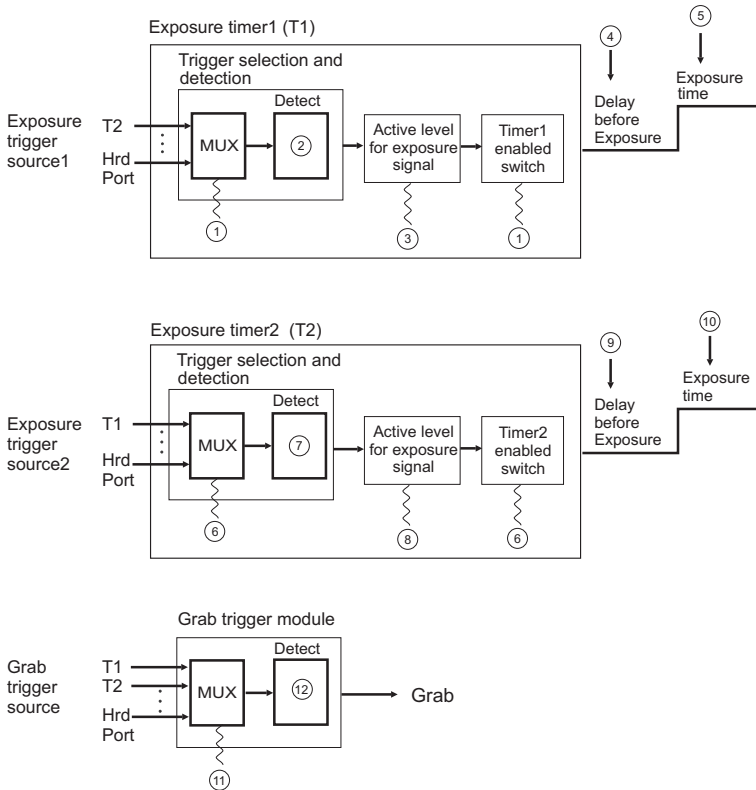
- **MdigControl()** with M_GRAB_TRIGGER_SOURCE selects which signal to use as the source of the trigger (for example, M_HARDWARE_PORT0). **MdigControl()** with M_GRAB_TRIGGER_MODE, selects the trigger detection method (for example, trigger on the rising edge of the signal).
- If the exposure time (**MdigControl()** with M_GRAB_EXPOSURE_TIME) is zero, the trigger sets off the grab trigger module immediately, initiating the actual grab. The exposure timers are bypassed.
- If you set the exposure time to a non-zero value, an exposure signal is generated with an active period equal to the specified exposure time (M_GRAB_EXPOSURE_TIME). The active period occurs after the specified delay (M_GRAB_EXPOSURE_TIME_DELAY). The signal will be generated with the specified polarity (M_GRAB_EXPOSURE_MODE). The end of exposure will trigger the grab trigger module, initiating the actual grab.

Manual exposure bypass model

In the manual bypass model, you are responsible for enabling and setting-up all the exposure timers and grab trigger connections

Manual exposure bypass model

(M_GRAB_EXPOSURE_BYPASS set to M_ENABLE)



① (M_GRAB_EXPOSURE_SOURCE + M_TIMER1)

② (M_GRAB_EXPOSURE_TRIGGER_MODE + M_TIMER1)

③ (M_GRAB_EXPOSURE_MODE + M_TIMER1)

④ (M_GRAB_EXPOSURE_TIME_DELAY + M_TIMER1)

⑤ (M_GRAB_EXPOSURE_TIME + M_TIMER1)

⑥ (M_GRAB_EXPOSURE_SOURCE + M_TIMER2)

⑦ (M_GRAB_EXPOSURE_TRIGGER_MODE + M_TIMER2)

⑧ (M_GRAB_EXPOSURE_MODE + M_TIMER2)

⑨ (M_GRAB_EXPOSURE_TIME_DELAY + M_TIMER2)

⑩ (M_GRAB_EXPOSURE_TIME + M_TIMER2)

⑪ (M_GRAB_TRIGGER_SOURCE)

⑫ (M_GRAB_TRIGGER_MODE)

Software triggers

In general, the digitizer's grab trigger module and exposure timers can also be triggered by software (M_SOFTWARE). In this case, following a grab call, nothing is grabbed until you call a specific function (discussed below). Note that in this case, the grab call must be asynchronous (that is, issue the grab with **MdigGrab()** in asynchronous mode or with **MdigGrabContinuous()**) or the grab call must be called on a separate thread.

In the automatic exposure model

In the automatic exposure model, issue the software trigger by calling **MdigControl()** with M_GRAB_TRIGGER and M_ACTIVATE. This will trigger the grab if the exposure time is 0, otherwise the call will trigger the exposure signal which in turn will trigger the grab.

In the manual bypass model

In the manual bypass model, to issue a software trigger for the grab trigger module, call **MdigControl()** with M_GRAB_TRIGGER and M_ACTIVATE. To issue a software trigger for one of the exposure timers, call **MdigControl()** with M_GRAB_EXPOSURE+M_TIMER n and M_ACTIVATE.

Note, for a digitizer without an exposure timer, the exposure time is considered to be zero.

Chapter 8: Color

This chapter discusses how to handle objects in color with MIL.

Dealing with color

MIL supports grabbing, displaying, and accessing color images.

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

Grabbing

You grab from an input device (typically a camera) into a color image buffer, as you would into a two-dimensional grayscale image buffer, by calling *MdigGrab()* or *MdigGrabContinuous()*.

Before performing a color grab, a digitizer must be allocated, using *MdigAlloc()* (or *MappAllocDefault()*), specifying a color digitization data format. In addition, the digitizer and the image buffer must be allocated on the same system and have compatible dimensions. Once you have finished using the digitizer, you should free it, using *MdigFree()*.

When grabbing from a color digitizer, each color component is transmitted simultaneously. The destination buffer must have the same number of color bands as the digitizer. The data is simultaneously stored in the appropriate component of the image buffer. When grabbing RGB, the red component is stored in the first color band, the green component is stored in the second color band, while the blue component is stored in the third color band.

If the hardware permits, you can control the digitization reference level of each channel, using *MdigReference()*.

❖ Note, upon installation, if you specified a color camera, the default image buffer allocated with *MappAllocDefault()* will be a three-band color image buffer. If you didn't specify a color camera, but would now prefer to use one, you might want to update the *milsetup.h* file to reflect the desired defaults for the allocation of your color camera and a color image buffer.

Note, most examples in this manual assume that the target system has a monochrome digitizer, and that the camera and default image buffer are monochrome. To run the examples using a color digitizer and image buffer, you must modify the code appropriately.

Mapping grabbed data through a LUT

You can also correct or precondition input data by mapping it through a LUT upon acquisition (if the hardware permits). This requires that you associate a LUT buffer with the input device, using *MdigLut()*.

The LUTs that can be associated to a digitizer are either one-dimensional LUT buffers (single rows) or LUT buffers that have the same number of color bands as the digitizer. If you associate a one-dimensional LUT buffer with the digitizer, each of the digitizer's color band input LUTs is loaded with the one-dimensional LUT buffer data. If you associate a multi-band LUT buffer with the digitizer, each of the digitizer's color band LUTs is loaded with its corresponding color band LUT buffer data.

Note, the LUT buffer depth must match the digitizer's pixel depth.

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

Displaying

You display a color-image buffer as you would a two-dimensional grayscale image buffer. You must first allocate the image buffer with a displayable attribute (`M_DISP`), then select it for display, using *MdispSelect()*. To stop displaying the image buffer and leave the display blank, use *MdispDeselect()*.

Before you can display a buffer, the display must be allocated, using *MdispAlloc()* (or *MappAllocDefault()*). The image buffer and the display should be allocated on the same system and have compatible dimensions.

When you display a color-image buffer (usually RGB), the first band is routed to the first output channel (usually red), the second band is routed to the second output channel (usually green), while the third band is routed to the third output channel (usually blue).

When a display is allocated, a default pass-through LUT (transparent LUT) is loaded into the output LUT(s) (if any). You can change the displayed colors of an image by associating a lookup table (LUT) to the display, using *MdispLut()*.

When you associate a one-color-band LUT buffer with a display that has more than one output LUT, the same LUT buffer data is loaded in each of the available output channel LUTs.

When you associate a multi-band LUT buffer to a display that has multiple output LUTs, each output LUT is loaded with the data of the corresponding LUT buffer color band.

To disassociate the LUT buffer from the display, you need to associate the display with the default LUT, using `M_DEFAULT` as a parameter to *MdispLut()*.

Saving and loading color images

MIL supports the saving and loading of color images from disk in different file formats. See the *MbufSave()*, *MbufLoad()*, *MbufRestore()*, *MbufImport()*, and *MbufExport()* command reference descriptions in Part II: The MIL-Lite reference for more details.

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

How to manage your color buffer

The following example demonstrates some ways in which to manage your color buffers:

```
/* File name: mcolor.c
 * Synopsis: This program allocates a displayable color image buffer,
 *           displays it, and loads its contents with a color image.
 *           It then does a copy of this image and writes text into the
 *           color components (RGB) of the copy of the image.
 */

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

/* Source MIL image file specifications. */
#define IMAGE_FILE      "bird.mim"
#define IMAGE_WIDTH     256L
#define IMAGE_HEIGHT    240L
#define IMAGE_BAND      3L
#define IMAGE_DEPTH     8L

void main(void)
{
    MIL_ID MilApplication, /* Application identifier. */
          MilSystem,      /* System identifier. */
          MilDisplay,     /* Display identifier. */
          MilImage,       /* Image buffer identifier. */
          MilSubImage0,   /* Sub-image buffer identifier for source image. */
          MilSubImage1,   /* Sub-image buffer identifier for copied image. */
          MilSubImage1Red, /* Sub-image buffer identifier for red component. */
          MilSubImage1Green, /* Sub-image buffer identifier for green component. */
          MilSubImage1Blue; /* Sub-image buffer identifier for blue component. */
    long ImageSizeX, /* Image width. */
         ImageSizeY; /* Image height. */
```

(cont...)

```

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

/* Find the best size for the display image depending on the display type. */
if (MdispInquire(MilDisplay, M_DISP_MODE, M_NULL) == M_WINDOWED)
{
    ImageSizeX = IMAGE_WIDTH * 2;
    ImageSizeY = IMAGE_HEIGHT;
}
else
{
    /* The size of the entire display to avoid possible display artifacts. */
    ImageSizeX = min(MdispInquire(MilDisplay, M_SIZE_X, M_NULL), M_DEF_IMAGE_SIZE_X_MAX);
    ImageSizeY = min(MdispInquire(MilDisplay, M_SIZE_Y, M_NULL), M_DEF_IMAGE_SIZE_Y_MAX);
}

/* Allocate a color display image buffer to perform processing in it. */
MbufAllocColor(MilSystem, IMAGE_BAND, ImageSizeX, ImageSizeY,
    IMAGE_DEPTH + M_UNSIGNED, M_IMAGE + M_DISP + M_PROC, &MilImage);

/* Clear the image buffer. */
MbufClear(MilImage, 0L);

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

/* Enable keying on display if it is supported. */
if ((M_DEF_DISPLAY_KEY_ENABLE_ON_ALLOC != 0) &&
    MdispInquire(MilDisplay, M_DISP_KEY_SUPPORTED, 0)
)
    MdispOverlayKey(MilDisplay, M_KEY_ON_COLOR, M_EQUAL, 0xFFL,
        M_DEF_DISPLAY_KEY_COLOR);

/* Define 2 sub-image buffers in the display buffer, restricting the
 * work regions to the image size.
 */
MbufChild2d(MilImage, 0L, 0L, IMAGE_WIDTH, IMAGE_HEIGHT,
    &MilSubImage0);
MbufChild2d(MilImage, IMAGE_WIDTH, 0L, IMAGE_WIDTH, IMAGE_HEIGHT,
    &MilSubImage1);

/* Load a color image in image 0. */
MbufLoad(IMAGE_FILE, MilSubImage0);

/* Print a message. */
printf("A color source image was loaded and displayed.\n");
printf("Press <Enter> to continue.\n");
getchar();

```

(cont...)


```

/* Copy the color image. */
MbufCopy(MilSubImage0, MilSubImage1);

/* Create child buffers that map to the red, green and blue components. */
MbufChildColor(MilSubImage1, M_RED, &MilSubImage1Red);
MbufChildColor(MilSubImage1, M_GREEN, &MilSubImage1Green);
MbufChildColor(MilSubImage1, M_BLUE, &MilSubImage1Blue);

/* Write color annotations in each component of the copied image. */
MgraColor(M_DEFAULT, 0xFF);
MgraText(M_DEFAULT, MilSubImage1Red,
         IMAGE_WIDTH/16, IMAGE_HEIGHT/8, "TOUCAN");
MgraColor(M_DEFAULT, 0x80);
MgraText(M_DEFAULT, MilSubImage1Green,
         IMAGE_WIDTH/16, IMAGE_HEIGHT/8, "TOUCAN");
MgraColor(M_DEFAULT, 0x00);
MgraText(M_DEFAULT, MilSubImage1Blue,
         IMAGE_WIDTH/16, IMAGE_HEIGHT/8, "TOUCAN");

/* Print a message. */
printf("The color source image in the top left corner was copied in the\n");
printf("top right corner image and color text annotation was done in it.\n");
printf("Press <Enter> to end.\n");
getchar();

/* Disable keying on the display if it is supported. */
if ((M_DEF_DISPLAY_KEY_DISABLE_ON_FREE != 0) &&
    MdispInquire(MilDisplay, M_DISP_KEY_SUPPORTED, 0))
    MdispOverlayKey(MilDisplay, M_KEY_OFF, M_NULL, M_NULL, M_NULL);

/* Release subimages and color image buffer. */
MbufFree(MilSubImage1Red);
MbufFree(MilSubImage1Green);
MbufFree(MilSubImage1Blue);
MbufFree(MilSubImage1);
MbufFree(MilSubImage0);
MbufFree(MilImage);

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

```

Chapter 9: JPEG compression

This chapter describes how to compress and decompress images.

Introduction

MIL-Lite allows you to compress and decompress images and sequences. Compression allows you to store more images in memory than would normally be possible. In addition, compression allows images to be transferred more quickly, since it reduces the amount of data that must be transferred.

MIL-Lite supports both lossy and lossless JPEG compression algorithms.

JPEG lossless

The JPEG lossless algorithm compresses images without any loss of information. Typically, the algorithm compresses images by a factor of 2:1, although a factor of 4:1 can sometimes be achieved. The JPEG lossless algorithm can compress 8- or 16-bit buffers with 1 or 3 bands.

JPEG lossy

The JPEG lossy algorithm compresses images by a variable factor but introduces some loss of information. The higher the compression factor, the more the compression, but the lower the image quality. The JPEG lossy algorithm can compress 8-bit buffers with 1 or 3 bands. To be compatible with most image-viewing software, MIL-Lite allows you to store compressed color images in YUV format.

Interlaced JPEG

MIL-Lite can perform a JPEG compression such that the image data is stored in separate fields. This is referred to as an *interlaced JPEG compression*. Unless otherwise stated, everything that applies to a JPEG compression also applies to an interlaced JPEG compression.

Control options

MIL-Lite allows you to control certain aspects of a compression. For example, you can use your own compression tables, although the default tables are suitable for most applications.

General steps

Compression

To compress an image:

1. Allocate a buffer in which to hold the compressed image. Use *MbufAlloc...()*, allocating the buffer with an *M_COMPRESS+CompressionType* attribute.
 - ❖ Compressed buffers that are created using the *MbufCreate...()* functions should not be used as the destination buffer of a MIL-Lite function. If a buffer with an *M_COMPRESS* specifier is used as a source buffer for an operation, the data will be decompressed depending on the attributes of the destination buffer.
2. If necessary, change the control settings of the buffer, using *MbufControl()*.

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

3. If the image to compress is stored in a buffer, use *MbufCopy()* to compress it into the buffer allocated in step 1. If it is stored in a file, use *MbufImport()*. Note that, if you want the compressed image stored on file rather than in a buffer, use *MbufExport()* instead of *MbufCopy()*. In this case, there is no need to allocate a destination buffer.

You can also automatically compress your grabbed images. To do so, use *MdigGrab()* with a destination buffer that has an *M_GRAB+M_COMPRESS+CompressionType* attribute.

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

Decompression

To decompress an image, use *MbufCopy()*, *MbufImport()*, or *MbufExport()*, depending on where the source image is stored (in a buffer or on file) and where you want results written (to a buffer or file).

Before the decompression, you should not change any control settings in the source image; the same controls must be used for decompression, otherwise the image data will be lost.

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

Sequences

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

Multi-band buffers, color formats, and control settings - JPEG

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

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

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

Application-specific markers

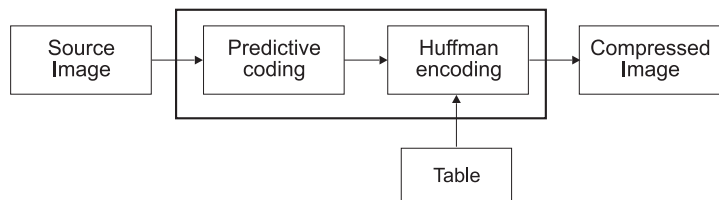
During a compression, MIL-Lite adds some application-specific markers to the resulting image. Most other packages will ignore these markers and therefore be able to decompress the file. MIL-Lite itself ignores unrecognized markers when it decompresses files.

Controlling a JPEG compression

This section provides a brief overview of the JPEG lossless and lossy algorithms and of the controls you have over these algorithms. In general, you should only change these controls if you are familiar with the algorithm you are using. For detailed information about the JPEG lossless and lossy algorithms, see *Information technology -- Digital compression and coding of continuous-tone still images: Requirements and guidelines*, which is available from the *International Standards Organization* (www.iso.ch). The section, *Improving results*, summarizes techniques to use to improve compression operations.

JPEG lossless

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



Predictive coding

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

MIL-Lite supports three types of predictive coding: predictor #0 (no predictor), predictor #1 (the “pixel-to-the-left” predictor), and predictor #2 (the “pixel-above” predictor). By default, MIL-Lite uses the pixel-to-the-left to predict values, which is suitable for most images. In some applications, you might prefer to use the pixel-above predictor. You can also specify no predictor (predictor #0), but note that in this case, the values after predictive coding will be the same as the original values. This predictor can be useful if you have developed your own algorithm to take the place of predictive coding and only need your images Huffman encoded. Note that you must implement your own algorithm to use one of the other “predictors” supported by the JPEG lossless algorithm. You can specify the predictor with the *MbufControl()* M_PREDICTOR control type.

Huffman encoding

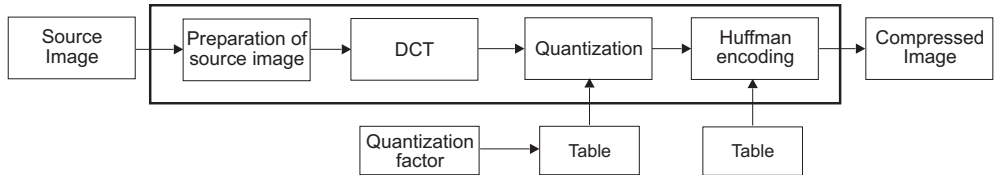
After an image has been predictive coded, Huffman encoding assigns a variable-length “code word” to each value. This code is based on the number of bits by which adjacent values differ. Values are assigned code words according to a DC Huffman table. You can use the default DC Huffman table or you can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

JPEG lossy

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

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

Afterwards, each 8x8 block of the image is represented in its frequency domain through a discrete cosine transform, resulting in 1 DC and 63 AC values. Each block is then quantized and Huffman encoded.



Quantization divides each of the 64 values in a block by a specified value, according to a quantization table. After each block is quantized, Huffman encoding assigns a variable-length “code word” to each value. Each DC value in a block is assigned a code word according to a DC Huffman table. The AC values are assigned a code word according to an AC Huffman table. You can control a JPEG lossy compression by using your own quantization and/or Huffman tables.

Restart markers

When an image is compressed, MIL-Lite adds restart markers to the bit-stream of the compressed image. A restart marker is a special code that signifies that the encoded bit-stream has been padded to the next byte boundary before the encoding process was restarted. Restart markers can be useful if you are transmitting the compressed image over a medium that is susceptible to errors. If an error does occur and there are no restart markers, the error will propagate and affect subsequent data. However, if there are restart markers, the error will be confined to the data between markers.

By default, MIL-Lite places restart markers after a certain number of rows of data have been encoded (for lossless compressions) or after a certain number of 8x8 blocks of data have been encoded (for lossy compressions). If necessary, you can use the *MbufControl()* `M_RESTART_INTERVAL` control type to change the number of rows or blocks between restart markers.

- ❖ For a lossy compression with a high compression ratio, too many restart markers can significantly increase the size of the compressed image. In this case, you might want to increase the number of blocks between restart markers, especially if you are not transmitting the image over a noisy medium. In fact, if you are sure that the transmission medium is not noisy, you might want to set the restart interval to 0, that is, not use restart markers. This will increase the compression ratio, as well as reduce the time required to decompress the image.

Improving results

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

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

For JPEG lossy compression:

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

For JPEG lossless compression:

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

Working with tables

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

❖ For JPEG compression, quantization divides values.

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

| Compression type | Table type | Buffer type, size, and attribute |
|------------------|--------------|---|
| JPEG lossless | DC Huffman | 1-dimensional, 8+M_UNSIGNED, 28 entries, M_ARRAY |
| JPEG lossy | DC Huffman | 1-dimensional, 8+M_UNSIGNED, 28 entries, M_ARRAY |
| | AC Huffman | 1-dimensional, 8+M_UNSIGNED, 178 entries, M_ARRAY |
| | Quantization | 2-dimensional, 8+M_UNSIGNED, 8 x 8 entries, M_ARRAY |

Inquiring values in default tables

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

1. First, inquire the MIL identifier of the default table using *MbufInquire()*. Then, inquire the size of the table using the same function.
2. Allocate a user array of the appropriate size for storing the default table values.
3. Get the values from the inquired table in Step 1 into the user array using *MbufGet()*.

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

Using your own table

To use your own table:

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

For JPEG lossy compressions of YUV images, use the `xx_LUMINANCE` and `xx_CHROMINANCE` control types. The control types without these suffixes control all bands.

- ❖ If you set the `M_Q_FACTOR` control type after specifying a custom table, the custom table will be scaled.

Chapter 10: Data manipulation with multiple systems

Data manipulation with multiple systems

To use multiple Matrox imaging boards, you have to allocate a MIL system for each board.

Processing

To perform a processing operation, your source and destination buffers can be on different systems; MIL will transparently copy buffers to the most efficient of these system, if necessary.

Exchanging data

To exchange data between systems, you can physically copy the data from one system to another. The copy is always performed by the most suitable system. If both systems are of the same type, the copy is always performed by the destination system.

Instead of performing a physical copy using *MbufCopy()*, you can allocate a buffer on one system and use *MbufCreate...()* to access this buffer from another system. *MbufCreate...()* creates a buffer that maps to allocated memory (for example, on the Host or any MIL system); no memory is actually allocated to this newly created buffer.

The second method can be used, for example, to update a buffer (or part of it) with data grabbed from different systems. Note that after writing to the created buffer, you should notify the real buffer that its contents have been changed, by calling *MbufControl()* with *M_MODIFIED*. See *Chapter 3: Specifying and managing your data buffers* for more information about creating data buffers.

Grab and display

To grab, the digitizer and the destination buffer must be allocated on the same MIL system. Similarly, to display a buffer, the display and the buffer must be allocated on the same MIL system.

Systems without an on-board display section use the VGA for display. Therefore, under Windows, such systems will automatically display together on the same screen.

Chapter 11: Using MIL with multi-processing and under multi-thread systems

This chapter describes how MIL handles multi-processing and multi-threading.

Multi-processing

Multi-processing is the ability to execute various processes (applications) simultaneously.

MIL applications are autonomous processes (or executables) designed to execute a complete operation or series of operations. Therefore, they can profit from multi-processing by executing independently, without interference from each other.

In general, when multiple processes are running, no sharing of systems is permitted, except for the Host and VGA. Some particular systems, such as Matrox Genesis, can also be shared.

Systems with multi-processing

Systems that support multiple processes have on-board resources (like processors) that can be shared by different processes. However, if many processes are running at the same time, these processes have to share the available processing time and will not be able to share data.

Systems without multi-processing

Not all systems support multi-processing. For example, a simple frame grabber with only acquisition capability (like the Matrox Meteor-II) cannot ensure either the response time to a command or the independence of a process necessary for multi-processing. Therefore, on such systems MIL will refuse to allocate the system if it is already being used by another process. To use a non-multi-processing system within a multi-processing environment, all processes that need to communicate with the system must do so by sending their requests through a single dedicated process.

Multi-threading

MIL also supports multi-threading. Multi-threading is the ability to perform multiple operations simultaneously in the same process. This is done by creating different threads (execution queues) to ensure sequential execution of operations within the same thread, while allowing simultaneous yet independent execution of other operations in other threads.

Threads within a process share the same data. Therefore, they can communicate and exchange data such as MIL identifiers.

Multi-threading is most appropriate for applications where independent tasks can be done simultaneously but need to share data or to be controlled and synchronized within a main task.

Speed considerations

Multi-threading does not always result in an increase of speed and efficiency. Threads running simultaneously share the same system resources (such as memory) and generally run on the same CPU. This sharing can, in some cases, slow the process. For example, when using a system with multiple CPUs under Windows NT, the threads generally run on separate CPUs and provide more processing power. However, since they share the same memory, operations that are I/O intensive and require only simple processing might not be accelerated.

Alternatives

Most applications do not require the use of multiple threads since there are other ways of multi-tasking. Mechanisms such as asynchronous grab and call-back functions can be used (see *MdigControl()* and *MdigHookFunction()*). Applications resolved by alternative means are often simpler to implement and easier to maintain than multi-threaded applications.

MIL and multi-threading

When your application contains several distinct parts that you want to run in parallel, it is often easier to design it so that each part is controlled by a separate thread (or task). For example, if you have two independent processing tasks that can be performed in parallel, it is often easier to have each controlled by a separate thread.

Thread execution

Under multi-thread operating systems, you can create as many threads as you require. The MIL commands in any thread are executed as follows:

- If the target processor is the Host CPU, processing in each thread is determined by the operating system.
- If the target processor is an on-board processor of a system that supports multi-threading (like the Matrox Genesis), MIL automatically creates, and eventually terminates, an on-board thread for each Host thread that sends commands to the board.

MIL application context

For each new Host thread sending MIL commands, MIL creates a new default MIL application context and initializes it to the state of the main MIL application (the first application allocated with *MappAlloc()*). Its purpose is to handle the context of the new thread, such as error reporting.

You can have the thread's application initialized to its initial state by allocating a new application using *MappAlloc()*; note that this must be the first call to MIL in the thread.

Synchronization

Thread synchronization is generally done by the Host synchronization services (such as Windows NT/2000 and 98 event objects). However, when using a system with an on-board processor, this processor is not synchronized with the Host.

This means that Host threads continue execution without waiting for the execution of the on-board commands to complete. In most cases, this is desirable to make the Host thread available for other tasks. However, for operations that necessitate the completion of a previous command(s) in order

to return valid results (for example, *MbufGet()* after an *MdigGrab()*), MIL automatically synchronizes the threads to force the Host to wait for completion of the earlier command(s).

Explicit synchronization might be necessary if commands sharing a common resource or system might conflict with each other. For example, two threads sharing the same image buffer MIL identifier might each try to clear the buffer to a different value. If the threads are not synchronized, these commands might execute at the same time and the buffer could be cleared to either value or even to a combination of the two values. Use the MIL synchronization command, *MappControlThread()*, to control the flow of such commands.

Thread control

Windows NT/2000 and 98 systems are both multi-process and multi-thread. They provide various thread control services, including events (used to synchronize threads).

The MIL *MappControlThread()* command serves as a link between MIL and the operating system. It controls and coordinates both MIL threads and MIL events. It can create and delete a MIL thread, set a thread as the current active thread, set its processing mode, determine its current state, and synchronize its processing by forcing a "wait" state. It can exert similar controls on MIL events. MIL events can be used in addition to, or instead of, the operating system's events.

Error reporting

Some functions in MIL are asynchronous, that is, they queue their command to the hardware and then immediately return control to the Host. For this reason, errors are only reported when the function is executed.

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

An example of using multiple threads or systems

Multiple threads

The following example illustrates how multiple threads can be used to perform processing. It also illustrates how to synchronize multiple threads, using events.

```

/* File name: mthread.c
* Synopsis: This program shows how to use different threads and synchronize
*           them with MIL. It creates 4 drawing threads that are used
*           to work in 4 different regions of a display buffer.
*
* Thread usage:
*   - The main thread starts a processing thread in each of the 4 different
*     quarters of a display buffer. The main thread then waits for a key to
*     be pressed to stop them.
*   - The top-left and bottom-left threads work in a loop, as follows: the
*     top-left thread draws a rectangle in its buffer (in a size
*     different from the previous rectangle), then sends an event to the
*     bottom-left thread. The bottom-left thread waits for the event from
*     the top-left thread, copies the contents of the top-left buffer into
*     its buffer, draws a circle in the rectangle, then sends an event to
*     the top-left thread. When the top-left thread receives the event, the
*     loop continues.
*   - The top-right and bottom-right threads work exactly the same way as the
*     top-left and bottom-left threads.
*
* Note that the top and bottom threads (of each half) could be set to do
* something else while waiting for each other.
*/

/* headers */
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <windows.h>
#include <mil.h>

/* local defines */
#define IMAGE_FILE           "bird.mim"
#define IMAGE_WIDTH          256
#define IMAGE_HEIGHT         240
#define DRAW_RADIUS_MAX      50
#define DRAW_CENTER_POSX     64
#define DRAW_CENTER_POSY     60
#define STRING_LENHT_MAX     40
#define STRING_POS_X         10
#define STRING_POS_Y         220
#define STRING_TOP            "0"
#define STRING_BOTTOM        "0"

/* Thread function prototypes */
unsigned long MFTYPE TopThread(void *TParam);
unsigned long MFTYPE BotLeftThread(void *TParam);
unsigned long MFTYPE BotRightThread(void *TParam);

```

(cont...)

```

/* Thread parameters structure */
typedef struct
{
    MIL_ID SrcImageId;
    MIL_ID DstImageId;
    MIL_ID EventSendId;
    MIL_ID EventWaitId;
    MIL_ID EventEndId;
    MIL_ID EventEndBotId;
    long *NumberOfIterPtr;
    long *ComVarPtr;
} THREAD_PARAM;

/* Main function: */
void main(void)
{
    MIL_ID MilApplication, /* Application identifier. */
    MilSystem, /* System identifier. */
    MilDisplay, /* Display identifier. */
    MilImage, /* Image buffer identifiers. */
    MilChild, /* Child buffer identifiers. */
    MilTopLeftImage, /* Top left child image. */
    MilBotLeftImage, /* Bottom left child image. */
    MilTopRightImage, /* Top right child image. */
    MilBotRightImage, /* Bottom right child image. */
    EventSendTopLeft, /* Event send by top left thread. */
    EventSendTopRight, /* Event send by top right thread. */
    EventWaitTopLeft, /* Event waited on by top left thread. */
    EventWaitTopRight, /* Event waited on by top right thread. */
    EventEndTopLeft, /* Event used to exit top left thread. */
    EventEndBotLeft, /* Event used to exit bottom left thread. */
    EventEndTopRight, /* Event used to exit top right thread. */
    EventEndBotRight; /* Event used to exit bottom right thread. */

    long NumberOfTopLeft = 0L, /* Number of top left threads iterations */
    NumberOfBotLeft = 0L, /* Number of bottom left threads iterations. */
    NumberOfTopRight = 0L, /* Number of top right threads iterations. */
    NumberOfBotRight = 0L, /* Number of bottom right threads iterations. */
    ComVarLeft = 0L, /* Communication variable for left thread. */
    ComVarRight = 0L; /* Communication variable for right thread. */

    THREAD_PARAM TParTopLeft, /* Parameters passed to top left thread. */
    TParBotLeft, /* Parameters passed to bottom left thread. */
    TParTopRight, /* Parameters passed to top right thread. */
    TParBotRight; /* Parameters passed to bottom right thread. */

    HANDLE ThreadHandle[4]; /* Thread handles. */
    DWORD ThreadId[4]; /* Thread Ids. */

```

(cont...)

```

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

/* Allocate child buffers. */
MbufChild2d(MilImage, 0, 0, IMAGE_WIDTH*2, IMAGE_HEIGHT*2, &MilChild);
MbufChild2d(MilChild, 0, 0, IMAGE_WIDTH, IMAGE_WIDTH, &MilTopLeftImage);
MbufChild2d(MilChild, IMAGE_WIDTH, 0, IMAGE_WIDTH, IMAGE_HEIGHT,
            &MilTopRightImage);
MbufChild2d(MilChild, 0, IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_HEIGHT,
            &MilBotLeftImage);
MbufChild2d(MilChild, IMAGE_WIDTH, IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_HEIGHT,
            &MilBotRightImage);
MdispSelect(MilDisplay, MilChild);

/* Allocate synchronization events. */
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventSendTopLeft);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventSendTopRight);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventWaitTopLeft);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventWaitTopRight);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventEndTopLeft);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventEndTopRight);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventEndBotLeft);
MappControlThread(M_DEFAULT, M_EVENT_ALLOC, M_DEFAULT, &EventEndBotRight);

/* Initialize source buffers. */
MbufLoad(IMAGE_FILE, MilTopLeftImage);
MbufLoad(IMAGE_FILE, MilTopRightImage);

/* Intitalize threads parameter structures. */
TParTopLeft.SrcImageId    = MilTopLeftImage;
TParTopLeft.DstImageId    = MilTopLeftImage;
TParTopLeft.EventSendId   = EventSendTopLeft;
TParTopLeft.EventWaitId   = EventWaitTopLeft;
TParTopLeft.EventEndId    = EventEndTopLeft;
TParTopLeft.EventEndBotId = EventEndBotLeft;
TParTopLeft.NumberOfIterPtr = &NumberOfTopLeft;
TParTopLeft.ComVarPtr     = &ComVarLeft;

TParBotLeft.SrcImageId    = MilTopLeftImage;
TParBotLeft.DstImageId    = MilBotLeftImage;
TParBotLeft.EventSendId   = EventWaitTopLeft;
TParBotLeft.EventWaitId   = EventSendTopLeft;
TParBotLeft.EventEndId    = EventEndBotLeft;
TParBotLeft.EventEndBotId = M_NULL;
TParBotLeft.NumberOfIterPtr = &NumberOfBotLeft;
TParBotLeft.ComVarPtr     = &ComVarLeft;

TParTopRight.SrcImageId   = MilTopRightImage;
TParTopRight.DstImageId   = MilTopRightImage;
TParTopRight.EventSendId  = EventSendTopRight;
TParTopRight.EventWaitId  = EventWaitTopRight;
TParTopRight.EventEndId   = EventEndTopRight;
TParTopRight.EventEndBotId = EventEndBotRight;
TParTopRight.NumberOfIterPtr = &NumberOfTopRight;
TParTopRight.ComVarPtr    = &ComVarRight;

```

(cont...)

```

TParBotRight.SrcImageId      = MilTopRightImage;
TParBotRight.DstImageId      = MilBotRightImage;
TParBotRight.EventSendId     = EventWaitTopRight;
TParBotRight.EventWaitId     = EventSendTopRight;
TParBotRight.EventEndId      = EventEndBotRight;
TParBotRight.EventEndBotId   = M_NULL;
TParBotRight.NumberOfIterPtr = &NumberOfBotRight;
TParBotRight.ComVarPtr       = &ComVarRight;

/*Start rotate and edge detect threads. */

ThreadHandle[0] = (HANDLE) _beginthreadex(NULL, 0L, &TopThread,
                                           &TParTopLeft, 0L, &(ThreadId[0]));
ThreadHandle[1] = (HANDLE) _beginthreadex(NULL, 0L, &BotLeftThread,
                                           &TParBotLeft, 0L, &(ThreadId[1]));
ThreadHandle[2] = (HANDLE) _beginthreadex(NULL, 0L, &TopThread,
                                           &TParTopRight, 0L, &(ThreadId[2]));
ThreadHandle[3] = (HANDLE) _beginthreadex(NULL, 0L, &BotRightThread,
                                           &TParBotRight, 0L, &(ThreadId[3]));

/* Send events to trigger operation of top left and top right threads. */
MappControlThread(EventWaitTopLeft, M_EVENT_SET, M_SIGNALED, M_NULL);
MappControlThread(EventWaitTopRight, M_EVENT_SET, M_SIGNALED, M_NULL);

/* Report what has happened to the Host screen. */
printf("Drawing done in a loop using four threads.\n");
printf("Press <Enter> to continue.\n");
getchar();

/* Make all threads exit. */
MappControlThread(EventEndTopLeft, M_EVENT_SET, M_SIGNALED, M_NULL);
MappControlThread(EventEndTopRight, M_EVENT_SET, M_SIGNALED, M_NULL);

/* Wait before freeing MIL objects that all threads are finished. */
while ((MappControlThread(EventEndTopLeft, M_EVENT_STATE, M_DEFAULT,
                          M_NULL) == M_SIGNALED) ||
       (MappControlThread(EventEndTopRight, M_EVENT_STATE, M_DEFAULT,
                          M_NULL) == M_SIGNALED));

printf("Top left iterations done:   %4ld.\n", NumberOfTopLeft);
printf("Bottom left iterations done: %4ld.\n", NumberOfBotLeft);
printf("Top right iterations done:   %4ld.\n", NumberOfTopRight);
printf("Bottom right iterations done: %4ld.\n", NumberOfBotRight);
printf("Press <Enter> to end.\n");
getchar();

```

(cont...)

```

/* Free buffers. */
MappControlThread(EventSendTopLeft , M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventSendTopRight, M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventWaitTopLeft , M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventWaitTopRight, M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventEndTopLeft , M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventEndTopRight , M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventEndBotLeft , M_EVENT_FREE, M_DEFAULT, M_NULL);
MappControlThread(EventEndBotRight , M_EVENT_FREE, M_DEFAULT, M_NULL);
MbufFree(MilTopLeftImage);
MbufFree(MilTopRightImage);
MbufFree(MilBotLeftImage);
MbufFree(MilBotRightImage);
MbufFree(MilChild);
/* Release defaults. */
MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, MilImage);
}

/* Top left and top right functions: */
/* ..... */
unsigned long MFTYPE TopThread(void *TParam)
{
    MIL_ID SrcImageId   = ((THREAD_PARAM *) TParam)->SrcImageId;
    MIL_ID DstImageId   = ((THREAD_PARAM *) TParam)->DstImageId;
    MIL_ID EventSendId  = ((THREAD_PARAM *) TParam)->EventSendId;
    MIL_ID EventWaitId  = ((THREAD_PARAM *) TParam)->EventWaitId;
    MIL_ID EventEndId   = ((THREAD_PARAM *) TParam)->EventEndId;
    MIL_ID EventEndBotId = ((THREAD_PARAM *) TParam)->EventEndBotId;
    long * RadiusVarPtr = ((THREAD_PARAM *) TParam)->ComVarPtr;
    char  Text[STRING_LENGTH_MAX] = STRING_TOP;
    long  Exit=0;

    while (!Exit)
    {
        /* Wait for event to process. */
        MappControlThread(EventWaitId, M_EVENT_WAIT, M_DEFAULT, M_NULL);

        /* Modify communication variable and reload image if necessary. */
        if (*RadiusVarPtr < DRAW_RADIUS_MAX)
        {
            *RadiusVarPtr += 10;
        }
        else
        {
            *RadiusVarPtr = 0;
            MbufLoad(IMAGE_FILE, DstImageId);
        }

        /* Print number of iterations and draw. */
        (((THREAD_PARAM *) TParam)->NumberOfIterPtr)++;
        ltoa((((THREAD_PARAM *) TParam)->NumberOfIterPtr), Text, 10);
        MgrColor(M_DEFAULT, 0xff);
        MgrText(M_DEFAULT, SrcImageId, STRING_POS_X, STRING_POS_Y, Text);
        MgrRectFill(M_DEFAULT, DstImageId, DRAW_CENTER_POSX-*RadiusVarPtr,
                    DRAW_CENTER_POSY-*RadiusVarPtr, DRAW_CENTER_POSX+*RadiusVarPtr,
                    DRAW_CENTER_POSY+*RadiusVarPtr);
    }
}

```

(cont...)


```

/* Check if processing must be terminated. */
if (MappControlThread(EventEndId, M_EVENT_STATE, M_DEFAULT,
                      M_NULL) == M_SINGALED)
{
    /* Make bottom thread exit. */
    MappControlThread(EventEndBotId, M_EVENT_SET, M_SINGALED, M_NULL);

    /* Set exit loop flag. */
    Exit=1;
}

/* Synchronize main thread with end of drawing. */
MappControlThread(EventSendId, M_EVENT_SET, M_SINGALED, M_NULL);
}

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

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

/* Bottom left functions: */
/* ..... */
unsigned long MFTYPE BotLeftThread(void *TParam)
{
    MIL_ID SrcImageId   = ((THREAD_PARAM *) TParam)->SrcImageId;
    MIL_ID DstImageId   = ((THREAD_PARAM *) TParam)->DstImageId;
    MIL_ID EventSendId  = ((THREAD_PARAM *) TParam)->EventSendId;
    MIL_ID EventWaitId  = ((THREAD_PARAM *) TParam)->EventWaitId;
    MIL_ID EventEndId   = ((THREAD_PARAM *) TParam)->EventEndId;
    long * RadiusVarPtr = ((THREAD_PARAM *) TParam)->ComVarPtr;
    char   Text[STRING_LENGTH_MAX] = STRING_BOTTOM;
    long   Exit=0;

    while (!Exit)
    {
        long i;
        /* Wait for event to process. */
        MappControlThread(EventWaitId, M_EVENT_WAIT, M_DEFAULT, M_NULL);

        /* Increment number of iterations. */
        (((THREAD_PARAM *) TParam)->NumberOfIterPtr) += 01L;
        /* Copy the top image. */
        MbufCopy(SrcImageId, DstImageId);
        /* Print iteration count and draw. */
        ltoa((((THREAD_PARAM *) TParam)->NumberOfIterPtr), Text, 10);
        MgraColor(M_DEFAULT, 0xFF);
        MgraText(M_DEFAULT, DstImageId, STRING_POS_X, STRING_POS_Y, Text);
        MgraColor(M_DEFAULT, 0x80);
        MgraArcFill(M_DEFAULT, DstImageId, DRAW_CENTER_POSX, DRAW_CENTER_POSY,
                    *RadiusVarPtr, *RadiusVarPtr, 0, 360);
    }
}

```

(cont...)

```

/* Check if processing must be terminated. */
if (MappControlThread(EventEndId, M_EVENT_STATE, M_DEFAULT,
                      M_NULL) == M_SINGALED)
    Exit=1;
/* Synchronize main thread with end of drawing. */
MappControlThread(EventSendId, M_EVENT_SET, M_SINGALED, M_NULL);
}

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

/* Bottom right function: */
/* ..... */
unsigned long MFTYPE BotRightThread(void *TParam)
{
    MIL_ID SrcImageId   = ((THREAD_PARAM *) TParam)->SrcImageId;
    MIL_ID DstImageId   = ((THREAD_PARAM *) TParam)->DstImageId;
    MIL_ID EventSendId  = ((THREAD_PARAM *) TParam)->EventSendId;
    MIL_ID EventWaitId  = ((THREAD_PARAM *) TParam)->EventWaitId;
    MIL_ID EventEndId   = ((THREAD_PARAM *) TParam)->EventEndId;
    long * RadiusVarPtr = ((THREAD_PARAM *) TParam)->ComVarPtr;
    char  Text[STRING_LENGTH_MAX] = STRING_BOTTOM;
    long  Exit=0;

    while (!Exit)
    {
        /* Wait for event to process. */
        MappControlThread(EventWaitId, M_EVENT_WAIT, M_DEFAULT, M_NULL);

        /* Increment number of iterations. */
        (*((THREAD_PARAM *) TParam)->NumberOfIterPtr)+= 01L;

        /* Copy the top image. */
        MbufCopy(SrcImageId, DstImageId);

        /* Print iteration count and draw. */
        ltoa(*((THREAD_PARAM *) TParam)->NumberOfIterPtr, Text, 10);
        MgraColor(M_DEFAULT, 0xFF);
        MgraText(M_DEFAULT, DstImageId, STRING_POS_X, STRING_POS_Y, Text);
        MgraColor(M_DEFAULT, 0x40);
        MgraArcFill(M_DEFAULT, DstImageId, DRAW_CENTER_POSX, DRAW_CENTER_POSY,
                   *RadiusVarPtr/2, *RadiusVarPtr/2, 0, 360);

        /* Check if processing must be terminated. */
        if (MappControlThread(EventEndId, M_EVENT_STATE, M_DEFAULT,
                              M_NULL) == M_SINGALED)
            Exit=1;

        /* Synchronize main thread with end of drawing. */
        MappControlThread(EventSendId, M_EVENT_SET, M_SINGALED, M_NULL);
    }

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

```

Chapter 12: Using MIL with Native Mode Functions

This chapter covers the use of Native Mode functions with MIL.

Integrating native functions with MIL code

MIL allows you to mix board-specific code (from the native library function set) with its own code. This is useful when you need to access some board-specific functionality that is not supported directly by the MIL function set or to optimize a time-critical piece of code.

When programming in native mode through MIL, you use the same board driver and programmer's kit that are used by regular native mode programmers. The only difference is the need to use certain rules and commands to ensure proper communication between MIL and the native functions. These rules and commands allow you to enter and leave native mode from MIL and access MIL for information, such as the object native handle, concerning data objects on the target board.

Portability

You should note that applications containing native mode functions are not portable to other present or future Matrox platforms supported by MIL.

Signaling MIL about Native Mode use

MIL must be signaled when entering and leaving native mode and when MIL objects have been modified while in native mode, using *MsysControl()*. For buffer modification, *MbufControl()* can also be used to signal MIL.

On entering native mode, MIL does not affect the current state of either the board or the environment.

The *M...Inquire()* functions can be used to determine the buffer, digitizer, or display native identifier (handle) required to use the system's native library.

On leaving native mode, MIL assumes that the board is in the same state as when entering. Therefore, you must ensure that you return the board to the proper state before returning control to MIL. Inquiries about the board state must be made using the board's native library inquiry functions.

A native mode example

In this example, we use MIL mixed with Genesis native library code to grab and warp an image.

Code

```

/* File name: mnatgen.c
 * Synopsis: This program shows how to use GENESIS native library
 *           function calls mixed with MIL function calls.
 */

/* general includes */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mil.h>
#include <imapi.h>

/* Operation control defines */
#define ALLOCATE 1
#define PROCESS 2
#define FREE 3

/* Native functions to grab and warp an image. */
void GrabAndWarp(MIL_ID MilSystem, MIL_ID MilDisplay, MIL_ID MilCamera,
                MIL_ID MilImage, long Operation);

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

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

    /* Allocate and initialize work buffers. */
    GrabAndWarp(MilSystem, MilDisplay, MilCamera, MilImage, ALLOCATE);

    /* Print a message on the host screen. */
    printf("Native function called in a loop...\n");
    printf("Press <Enter> to end...");
}

```

(cont...)

```

/* Grab and warp grabbed image in a loop */
while (!kbhit())
{
    GrabAndWarp(MilSystem, MilDisplay, MilCamera, MilImage, PROCESS);
}

/* Free work buffers */
GrabAndWarp(MilSystem, MilDisplay, MilCamera, MilImage, FREE);

MappFreeDefault(MilApplication, MilSystem, MilDisplay, MilCamera,
                MilImage);
}

/* Native function: */
/* ..... */
void GrabAndWarp(MIL_ID MilSystem, MIL_ID MilDisplay, MIL_ID MilCamera,
                MIL_ID MilImage, long Operation)
{
    /* Warp coefficient and LUT ID variables.
     * (kept in static to avoid warp coefficient calculation at each call.)
     */
    static long NativeWarpBufId      = M_NULL;
    static long NativeWarpLutXBufId  = M_NULL;
    static long NativeWarpLutYBufId  = M_NULL;
    static long NativeGrabBufId      = M_NULL;
    static long NativeWarpResultBufId = M_NULL;

    /* Inquire useful MIL information. */
    long SizeX  = MdigInquire(MilCamera, M_SIZE_X, M_NULL);
    long SizeY  = MdigInquire(MilCamera, M_SIZE_Y, M_NULL);
    long SizeBand = MdigInquire(MilCamera, M_SIZE_BAND, M_NULL);

    /* Miscellaneous local variables */
    double CornerX1 = 0.0;
    double CornerY1 = 0.0;
    double CornerX2 = SizeX - 1.0;
    double CornerY2 = 0.0;
    double CornerX3 = 2.0 * SizeX;
    double CornerY3 = SizeY - 1.0;
    double CornerX4 = -1.0 * SizeX;
    double CornerY4 = SizeY - 1.0;
    long SrcXStart = 0L;
    long SrcYStart = 0L;
    long SrcXEnd   = SizeX - 1L;
    long SrcYEnd   = SizeY - 1L;

```

(cont...)

```

/* Inquire Genesis native Id's */
long NativeSysThreadId = MsysInquire(MilSystem, M_NATIVE_THREAD_ID, M_NULL);
long NativeDigCameraId = MdigInquire(MilCamera, M_NATIVE_CAMERA_ID, M_NULL);
long NativeDigControlId = MdigInquire(MilCamera, M_NATIVE_CONTROL_ID,
                                     M_NULL);
long NativeDigId = MdigInquire(MilCamera, M_NATIVE_ID, M_NULL);
long NativeBufId = MbufInquire(MilImage, M_NATIVE_ID, M_NULL);

/* Notify MIL that we are entering native mode. */
MsysControl(MilSystem, M_NATIVE_MODE_ENTER, M_NULL);

/* Do the selected operation.*/
switch (Operation)
{
/* Preallocate grab and warp buffers (done once for speed). */
case ALLOCATE:
{
    imBufAlloc(NativeSysThreadId, SizeX, SizeY, SizeBand, IM_UBYTE,
               IM_PROC, &NativeGrabBufId);
    imBufAlloc(NativeSysThreadId, SizeX, SizeY, SizeBand, IM_UBYTE,
               IM_PROC, &NativeWarpResultBufId);
    imBufAlloc(NativeSysThreadId, 3L, 3L, 1L, IM_FLOAT, IM_PROC,
               &NativeWarpBufId);
    imBufAlloc(NativeSysThreadId, SizeX, SizeY, 1L, IM_SHORT, IM_PROC,
               &NativeWarpLutXBufId);
    imBufAlloc(NativeSysThreadId, SizeX, SizeY, 1L, IM_SHORT, IM_PROC,
               &NativeWarpLutYBufId);
    if (NativeGrabBufId && NativeWarpResultBufId && NativeWarpBufId &&
        NativeWarpLutXBufId && NativeWarpLutYBufId)
    {
        /* Calculate warp coefficients */
        imGenWarp4Corner(NativeSysThreadId, NativeWarpBufId, CornerX1,
                        CornerY1, CornerX2, CornerY2, CornerX3, CornerY3,
                        CornerX4, CornerY4, SrcXStart, SrcYStart,
                        SrcXEnd, SrcYEnd, IM_DEFAULT, 0L);
        imGenWarpLutMatrix(NativeSysThreadId, NativeWarpLutXBufId,
                           NativeWarpLutYBufId,
                           NativeWarpBufId, 0L, 0L);
    }
    else
    {
        printf("Error allocating resources...\n");
    }
    break;
}
}

```

(cont...)

```

/* Grab and Warp buffer. */
case PROCESS:
{
    /* Process if allocations were successful */
    if (NativeGrabBufId && NativeWarpResultBufId && NativeWarpBufId &&
        NativeWarpLutXBufId &&NativeWarpLutYBufId)
    {
        /* Grab the image */
        imDigGrab(NativeSysThreadId, NativeDigId, NativeDigCameraId,
            NativeGrabBufId, 1L, NativeDigControlId, 0L);

        /* Warp the grabbed image. */
        imIntWarpLut(NativeSysThreadId, NativeGrabBufId, NativeWarpResultBufId,
            NativeWarpLutXBufId, NativeWarpLutYBufId, 0L, 0L);

        /* Copy the result into the display buffer */
        imBufCopy(NativeSysThreadId, NativeWarpResultBufId, NativeBufId, 0L,
            0L);
    }
    break;
}

/* Free grab and warp buffers. */
case FREE:
{
    if (NativeGrabBufId)
        imBufFree(NativeSysThreadId, NativeGrabBufId);
    if (NativeWarpResultBufId)
        imBufFree(NativeSysThreadId,NativeWarpResultBufId);
    if (NativeWarpBufId)
        imBufFree(NativeSysThreadId, NativeWarpBufId);
    if (NativeWarpLutXBufId)
        imBufFree(NativeSysThreadId, NativeWarpLutXBufId);
    if (NativeWarpLutYBufId)
        imBufFree(NativeSysThreadId, NativeWarpLutYBufId);
    break;
}

/* Notify MIL that we leave native mode. */
MsysControl(MilSystem, M_NATIVE_MODE_LEAVE, M_NULL);

/* Notify MIL that the buffer was modified. */
MbufControl(MilImage, M_MODIFIED, M_DEFAULT);
}

```

Chapter 13: Distribution

This chapter presents how to distribute MIL-Lite applications.

Distribution of MIL-Lite-based applications

There are some details that you must consider before you can distribute a MIL-based application, either for your customers' use or for your own. When doing this distribution, you must redistribute MIL-Lite run-time DLLs and device drivers. This chapter deals with the different ways to distribute your application.

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

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

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

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

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

Redistributing directly from the MIL-Lite CD

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

Redistributing using your own setup program

Alternatively, to redistribute your MIL-Lite run-time DLLs and device drivers, you can have your application's setup program call MIL-Lite's redistribution setup program in one of two ways:

- **Normal redistribution.** Prompts your customer for setup information.
- **Silent redistribution.** Uses a custom response file instead of prompting your customer for information.

These are described in detail in the next sections.

Normal redistribution using your custom CD

To distribute your MIL-Lite run-time DLLs and device drivers, you can have your application's setup program call MIL-Lite's redistribution setup program. If you use the normal redistribution mechanism, your customer will be prompted for information during the setup. To use your setup program, do as follows:

1. Copy the `\REDIST` directory from the MIL-Lite CD to your installation directory.
2. If you want to save space on your target CD, you can remove certain components of the `\REDIST` directory if they are not required for your particular application. For example, the `\REDIST\GENESIS` and `REDIST\MATROX\DRIVERS\GENESIS` directories can be removed if you are not using a Matrox Genesis board. Refer to the *redist.txt* file for more examples.
3. Note that the driver-specific **.inf* files must be modified. These files can be found in the appropriate driver directory located in the `\REDIST` directory. If you do not modify these files there will be a reference to MIL-Lite and the MIL-Lite CD each time a driver is being installed. You would likely want to replace these references with references to your product and CD. As well, under Windows 98/Me/2000, it is only possible to install the Matrox frame grabber drivers if the boards are physically present in the computer at the time of installation.

4. Have your redistribution program call the *setup.exe* file in the `\REDIST` directory. The *setup.exe* program installs the required run-time MIL-Lite DLL files and device drivers on your client's computer. An example call would be:

```
C:\Redist\Matrox\redist.exe REDISTRIBUTION
```

Silent redistribution

A silent redistribution does not prompt your user for any MIL-Lite information; instead, it uses a response file to provide the necessary setup parameters for the intended computer. You would use a silent redistribution when you are including MIL-Lite within your application and you do not want to have any Matrox Imaging setup dialog boxes appear. You could also use a silent redistribution if you wanted to control the setup parameters for your client.

It is not possible to create a response file for only a few of the setup parameters and ask the customer for the rest of the setup information. If you are going to use a response file, you must answer all of the setup questions in the response file.

To redistribute the MIL-Lite run-time DLLs and device drivers using a silent redistribution:

1. Follow the steps for a normal MIL-Lite redistribution.
2. Create a response file that provides the setup questions with the answers you want. Refer to the next subsection for details.
3. Have your redistribution program call the *redist.exe* program with the additional `'REDISTRIBUTION RESPONSEFILE = "<filename>" -s'` parameter to specify the name and the location of your response file. For example:

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

- ❖ In a silent redistribution, if a copy of MIL-Lite is found on the target computer, the installation will not occur. MIL-Lite will not be overwritten or uninstalled. In this instance, there will be an error code in the registry key *HKEY_CURRENT_USER\ SOFTWARE\ MATROX ENTRY: STATUS*, signifying that a previous version of MIL or MIL-Lite has been found.

Response file parameters

The response file's format parameters follow; the error codes can be found in the ***Redist.txt*** file. The order in which you place the parameters in the response file is not important. What is important is that each parameter, and its settings, is written as one unbroken line of code, with a carriage return at the end of the line. Any errors in the response file will cause the silent distribution to stop. Here is an example of a typical response file for an installation under Windows 98/Me.

```
SILENTMODE = 1
INSTALLATION_DIRECTORY = C:\Program Files\Matrox Imaging
DRIVER1 = METEOR_II
DRIVER2 = ORION
DMA_METEOR_II = 4
DMA_ORION = 4
DMA_VGA = 0
MGA_DRIVER = INSTALL
SUPERPRO = IGNORE
```

These are the potential parameters for a response file.

■ SILENTMODE.

This parameter designates silent installation and is a required parameter. SILENTMODE must be set to 1.

■ INSTALLATION_DIRECTORY.

This parameter specifies the target installation directory and is a required parameter.

■ DRIVER x .

This parameter designates the drivers that are to be installed. Replace x with the number to assign to the driver. More than one driver can be installed, but each driver must be assigned a different number; these numbers do not have

to be consecutive. The available settings for this parameter are: CORONA_II, METEOR_II, METEOR_II_1394, ORION, and VGA. Use the METEOR_II setting for both the Matrox Meteor-II /Standard and Meteor-II /Multi-Channel boards.

GENESIS, METEOR_II_DIG, and METEOR_II_CL are also possible settings for a response file, but in this case the redistribution will not be completely silent. Completely silent redistribution is not possible with the Matrox Genesis, Meteor-II /Digital, and Meteor-II /Camera Link boards because they call on another library which does not support silent redistribution.

Also, note that for backwards compatibility, response file settings METEOR2 and METEOR2D can also be used for **DRIVERx**. However, we recommend using the new settings, METEOR_II and METEOR_II_DIG.

■ **DMA_MEMORY_SIZE.**

This parameter is used only for Windows NT/2000 and designates the non-paged memory size in Mbytes. The memory set aside is not board-specific. This must be a minimum of one Mbyte; if you only have a graphics controller, or a graphics controller and a Matrox Meteor-II /1394 board, you can set this parameter to a minimum of 0 Mbytes.

■ **DMA_CORONA_II, DMA_METEOR_II, DMA_ORION, DMA_VGA.**

These parameters are used with Windows 98/Me and they designate the non-paged memory sizes in Mbytes for each board. This must be done for each board installed under Windows 98/Me except for Matrox Meteor-II /1394.

The **DMA_VGA** parameter designates the non-paged memory sizes in Mbytes for the graphics controller.

DMA_VGA must have a value if you are creating a response file under Windows 98/Me; in general, this should be set to zero. This memory can be used to store data across processes (using MbufCreate ()), although this should be used with caution; it is generally better to leave memory control to MIL.

■ MGA_DRIVER.

This parameter has two settings: INSTALL and IGNORE. The INSTALL setting will install the MGA driver in silent mode, while the IGNORE setting will not install the MGA driver.

▲ *Warning!*

If the MGA driver is installed and a previous version of the MGA driver is on the computer, the installation program will overwrite the previous version rather than uninstall it.

Note, in the *RESPONSE.txt* provided, you can comment out or disable a setting by preceding the line with two slashes: // (putting "/" anywhere on the line will also disable the setting).

Debugging the response file

The response file needs to be error-free. Some common errors to avoid are:

- **Missing underscores.** The underscores must be present in the parameter name and the parameter settings.
- **Incorrect case.** Every character in a parameter or setting must be uppercase, unless it is part of a file path.
- **Incorrect parameter setting.** If a setting is defined in units, those units must not be written in the response.txt file (for example, DMA_CORONA_II = 4, not DMA_CORONA_II = 4Mbytes)
- **Broken line of code.** Each parameter and its setting must be written as one unbroken line of code finished by a carriage return.
- **Unknown location for response file.** The response file is not in the location specified by your redistrib.exe command line parameter **Responsefile**.
- **Unknown location for \Redist\.** The \REDIST\ directory is not in the correct location on the installation CD. Make sure when writing the redistribution program that the path is correct when calling the *redist.exe*.

- **Missing INF and DLL files (for Windows 98/Me).** The *MtxImage.inf* and *Mtximgci.dll* files are not in the root of the installation CD. Make sure you copy them with the *\REDIST* directory.

To debug your response file, you will have to run the redistribution executable file, and fix any errors that occur. Should an error occur, there will be an error code in the registry key, *HKEY_CURRENT_USER\ SOFTWARE\ MATROX ENTRY: STATUS*. Refer to the *redist.txt*, where the entire error code list is stored.

Important notes for Windows 98/Me users

- Ensure that the O/S has a standard VGA display driver installed prior to the installation of MGA drivers.
- The MGA drivers are only installed for boards that have an on-board graphics controller (Matrox Corona-II and Matrox Orion only).
- To avoid an MGA diagnostic message when installing the MGA driver, set the *RUN_DIAG* parameter to NO in the *MGA.ini* file prior to the installation. This file is located in the *\REDIST\MGADRV\WIN98* directory.

Important notes for Windows NT/2000 users

- Ensure that the O/S has a standard VGA display driver installed prior to the installation of the MGA drivers.
- The MGA drivers are only installed for boards that have an on-board graphics controller (Matrox Corona-II, Matrox Genesis and Matrox Orion only).

Uninstallation

To uninstall previous versions of MIL-Lite, or other Matrox Imaging Products, call the Matrox Imaging Products Uninstallation program and specify the products that you want to uninstall. To make the uninstallation silent, make the call with the silent mode parameter. The following command-line parameters are available:

| Parameter | Description |
|-----------|---|
| /s | Silent mode (no dialog box). |
| /n | No reboot at the end of uninstallation. |
| /m | UnInstall MIL or MIL-Lite. |
| /a | UnInstall ActiveMIL or ActiveMIL-Lite. |
| /i | UnInstall Intellicam. |
| /h | Display parameters. |

For example, if you want to uninstall MIL and ActiveMIL in silent mode without rebooting your system, call:

```
C:\Winnt\UnInstallMIP.exe/s/n/m/a
```

❖ The Matrox Imaging Products Uninstallation program is copied to the Windows directory at the time of installation.

You will find the uninstallation status in the following registry key: *HKEY_CURRENT_USER\ SOFTWARE\ MATROX ENTRY: UNINSTALL_STATUS*

To see the error codes, refer to the *redist.txt*.

MIL and MIL-Lite licenses

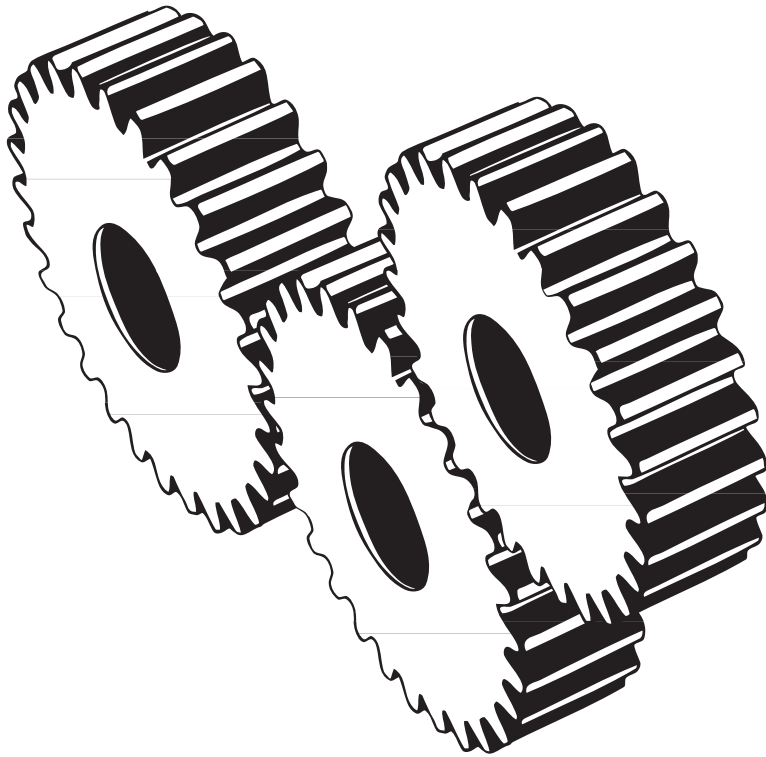
MIL and MIL-Lite have different licensing terms and mechanisms.

When you purchase MIL-Lite, you receive a development license and a registration number. MIL-Lite also comes with a royalty-free run-time license, which allows you to redistribute applications based on MIL-Lite without paying additional royalty fees.

With MIL, you can purchase two types of permanent licenses: a development license and/or a run-time license. Licenses are always verified when a MIL application is allocated, as well as when the application is running; this verification has negligible overhead. You can use a temporary license while waiting for a permanent license.

Please refer to the Matrox Software License Agreement for the legal provisions of using/redistributing MIL or MIL-Lite.

Part II:
The MIL-Lite
reference

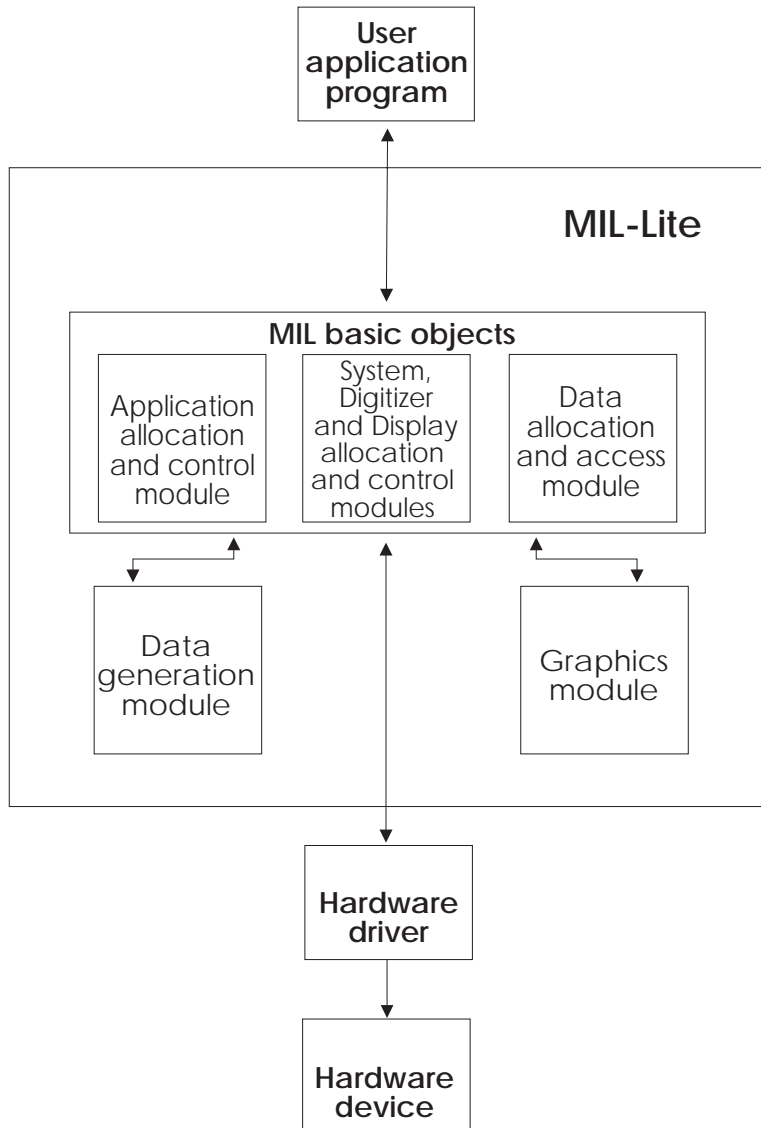


Putting thoughts into motion....

Chapter 14 : Programming with MIL

A MIL overview

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



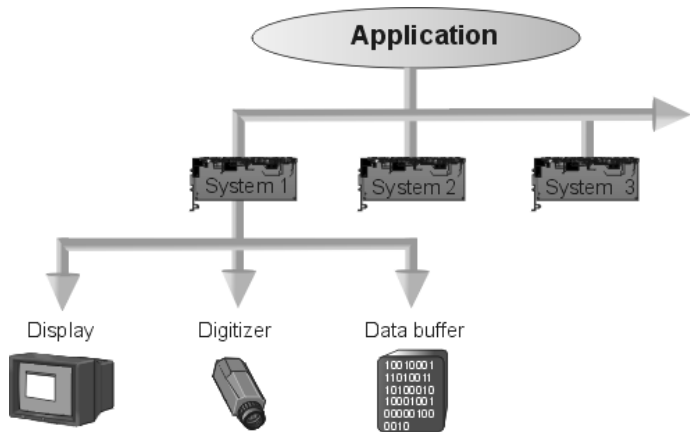
Part I of this manual, *Using MIL-Lite*, describes how to solve typical applications using the library. Code examples are also provided.

Starting your MIL application

Application and system initialization

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 *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()**. Refer to *Chapter 7: Grabbing with your digitizer* for more information on digitizers and *Chapter 5: 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. Refer to *Chapter 3: 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 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. |
| MbufCopy() | SrcBufId, DestBufId | Copy data from one buffer to another. |
| 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. |

| Data allocation and access commands | Command parameters | Description |
|--|---|---|
| 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. |
| 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. |

| Data allocation and access commands | Command parameters | Description |
|--|---|---|
| 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 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. |
| MdigFree() | DigId | Free a digitizer. |
| MdigGrab() | DigId, DestImageBufId | Grab data from an input device into a buffer. |

| Digitizer allocation and control commands | Command parameters | Description |
|--|--|--|
| 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. |
| MdispPan() | DisplayId, XOffset, YOffset | Pan and scroll a display. |

| Display allocation and control commands | Command parameters | Description |
|---|---|---|
| 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. |

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 |
|-------------------------|---|--|
| 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. |

| Basic graphics commands | Command parameters | Description |
|-------------------------|---|---------------------------------------|
| 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 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 15: 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

Part I of this manual 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()**

MappAllocDefault

Synopsis Allocate MIL application defaults.

Format `void MappAllocDefault(InitFlag, ApplicationIdPtr, SystemIdPtr, DisplayIdPtr, DigIdPtr, ImageBufIdPtr)`

| | |
|---------------------------|--|
| long InitFlag; | Initialization flag |
| MIL_ID *ApplicationIdPtr; | Storage location for application identifier |
| MIL_ID *SystemIdPtr; | Storage location for system identifier |
| MIL_ID *DisplayIdPtr; | Storage location for display identifier |
| MIL_ID *DigIdPtr; | Storage location for digitizer identifier |
| MIL_ID *ImageBufIdPtr; | Storage location for image buffer identifier |

Description This macro sets up the requested MIL and processing environments using the defaults specified in the *milsetup.h* file. It can allocate and initialize a MIL application, allocate the system to receive the MIL commands, allocate the digitizer and display, and allocate and clear a displayable image buffer on this target system, depending on what is requested.

The **InitFlag** parameter specifies the type of initialization setup to perform and is used principally to initialize the default system. This parameter can be set to one of the following:

| | |
|------------|---|
| M_COMPLETE | Perform a complete initialization of the MIL environment: initialize MIL to its default state and download any system's required resident software. At least one complete initialization is necessary after you power-up your system. |
| M_PARTIAL | Initialize MIL to its default state, but do not download any system's resident software. |
| M_SETUP | Set InitFlag to one of the above, based on the default state requested when the installation utility was run (refer to the <i>milsetup.h</i> file to determine what these setup defaults are). |

M_PARTIAL should only be selected if the required resident software has already been downloaded. This option is particularly useful when debugging since resident software generally needs to be downloaded once after power-up (or rebooting the system) and the downloading process can take a substantial amount of initialization time on certain systems.

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 |
|-------------|-----------------|---|
| M_ERROR | M_PRINT_ENABLE | Enable printing of error messages (default). |
| M_ERROR | M_PRINT_DISABLE | Disable printing of error messages. If error printing is disabled, you can still check for error, using MappGetError() . |
| M_TRACE | M_PRINT_ENABLE | 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)**

| | |
|--------------------------------|--------------------------|
| long HookType; | Type of event to hook |
| MAPPHOOKFCTPTR 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 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()**

MapplInquire

Synopsis Inquire about the application parameter setting.

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

| 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 **MappInquire()** 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 **MappControl()**

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 |

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

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. |

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_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 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. |

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.

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

Status Current limitation:

- For M_LUT data buffer, the data type must be 8, 16, or 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. |

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_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_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 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. |

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_DDRA | 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 **BuflIdPtr** 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.

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

Status Current limitation:

- For M_LUT data buffer, the data type must be 8, 16, or 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 **MbufAlloc1d0** and **MbufAlloc2d0** 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 **MappGetError0**, or by verifying that the buffer identifier returned is not M_NULL.

When a buffer is no longer required, release it, using **MbufFree0**.

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_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_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 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, and 3-band data in: M_RGB24 or M_RGB48. |
| 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, and 3-band 8-bit data in: M_RGB24, M_YUV24, M_YUV12, M_YUV9, M_YUV16 + M_PLANAR, or M_YUV16 + M_PACKED. |
| M_JPEG_LOSSY_INTERLACED | 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 M_YUV16 + M_PACKED. |

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 **BuflDPtr** 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.

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 **BufIdPtr** 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-Lite 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-Lite will be unable to decompress the data and the data is therefore irrelevant. See *Chapter 9: JPEG compression* for more information.

| ControlType | ControlValue | Description |
|--------------|-------------------------------------|--|
| 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. |

| ControlType | ControlValue | Description |
|--------------------------|-------------------------------------|---|
| 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. |
| 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. |

| ControlType | ControlValue | Description |
|----------------------------|--|--|
| 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. |
| M_Q_FACTOR_LUMINANCE | integer value between 1 and 99; default value is 50 | Quantization factor for JPEG 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 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. |
| M_QUANTIZATION_LUMINANCE | ID of buffer with M_ARRAY attribute | Associate a quantization table to the buffer. Only used for JPEG 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 lossy compressions of YUV buffers. The table is applied to the U and V bands. |
| 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). |

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()**

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. |

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_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_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: | |
|---|-----------------------------------|
| 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. |

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 | DataPtr is the Host address of the data buffer. The pitch is in pixels. |
| M_HOST_ADDRESS +M_PITCH_BYTE | DataPtr is the Host address. The pitch is in bytes. |
| M_PHYSICAL_ADDRESS +M_PITCH | DataPtr is the physical address of the data buffer in memory. The pitch is in pixels. |
| M_PHYSICAL_ADDRESS +M_PITCH_BYTE | 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 **BuflIdPtr** 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.

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.

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)

| | |
|------------------------|--|
| 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 |
| long ControlFlag; | Creation control flag |
| long Pitch; | Value of pitch, if necessary |
| void **ArrayOfDataPtr; | Array of data buffer pointers |
| MIL_ID *BufIdPtr; | 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_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_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.

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. |
| M_OFFSET_CENTER_Y | Offset center Y coordinate. |

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 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. |
| M_BMP | Save the buffer contents in BMP file format. The BMP file format that is used is the standard Windows format. |

| FileFormat | Description |
|----------------------------|---|
| 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 or M_JPEG_LOSSLESS_INTERLACED 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.

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. |

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; 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) 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 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:

| | |
|------------------------------|---|
| <code>M_MIL</code> | Import data that is in MIL file format. |
| <code>M_TIFF</code> | Import data that is in TIFF file format (only available for image buffers). The TIFF 6.0 specification is used. |
| <code>M_BMP</code> | Import data that is in BMP file format (only available for image buffers). The standard Windows BMP format is used. |
| <code>M_RAW</code> | Import data that is in RAW file format. |
| <code>M_JPEG_LOSSLESS</code> | Import a JPEG lossless image. |
| <code>M_JPEG_LOSSY</code> | 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 that 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. |

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. |
| 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. |
| 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. | |

| InquireType | Description |
|------------------------|--|
| 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_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 or M_JPEG_LOSSY_INTERLACED: | |
| M_Q_FACTOR | Quantization factor for JPEG lossy compressions. 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 JPEG lossy compressions of a YUV image buffer. |
| M_Q_FACTOR_CHROMINANCE | Quantization factor of the U and V bands for JPEG lossy compressions of a YUV image buffer. |
| M_QUANTIZATION | Identifier of the array buffer containing the quantization table (for a JPEG 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 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 JPEG lossy compressions. |

| InquireType | Description |
|---|---|
| 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. |
| 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()**.

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) 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 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()**

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 MdigGrab0 . 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) | 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. 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. Note, an error is returned if the specified exposure time cannot be generated. | |
| 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) | Set the delay (in nsec) between the trigger and the start of exposure. If M_DEFAULT, same value as DCF. Note, an error is returned if the specified delay cannot be generated. | |
| 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()**

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, mwindisp.c**

See also **MdigGrabContinuous(), MdigControl()**

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_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_BIT_SHIFT (8-bit shift), 32-bit M_BIT_SHIFT (24-bit shift), 32-bit float M_BIT_SHIFT (0 shift) |
| 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 | <p>Activate the keys associated with the display window: M_ENABLE (default) or M_DISABLE.</p> <p><i>The default key usage is:</i></p> <table> <tr> <td>+</td><td>Increase the X and Y zoom factors.</td></tr> <tr> <td>-</td><td>Decrease the X and Y zoom factors.</td></tr> <tr> <td>Pg-up</td><td>Scroll the buffer up to the previous display section.</td></tr> <tr> <td>Pg-dn</td><td>Scroll the buffer down to the next display section.</td></tr> <tr> <td>Up arrow</td><td>Scroll the buffer up to the previous line.</td></tr> <tr> <td>Dn arrow</td><td>Scroll the buffer down to the next line.</td></tr> <tr> <td>Left arrow</td><td>Pan the buffer left by one pixel.</td></tr> <tr> <td>Right arrow</td><td>Pan the buffer right by one pixel.</td></tr> <tr> <td>Ctrl Up arrow</td><td>Scroll the buffer up to the previous display section.</td></tr> </table> | + | 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. |
| + | 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_ERASEBKGD 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, 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 4:Lookup tables* and *Chapter 5: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 4: Look-up tables (LUTS)*, as well as *Chapter 5: 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()**.

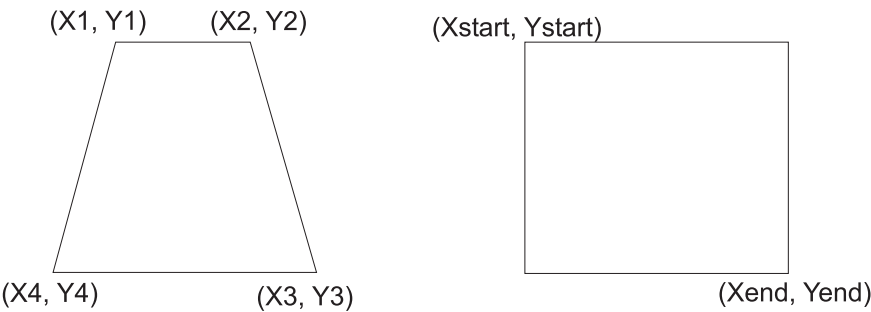
The **StartIndex** and **EndIndex** parameters specify the first and last LUT index entry for which to generate values. **StartIndex** must be less than or equal to **EndIndex**.

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()**

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



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, mwarp.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)**

| | |
|---------------------|-------------------------------------|
| MIL_ID GraphContId; | Graphics context identifier |
| long InquireType; | Graphic parameter to inquire |
| void *UserVarPtr; | 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 **M_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 |
|-------------------|--|
| M_COLOR | Foreground color. |
| M_BACKCOLOR | Background color. |
| M_BACKGROUND_MODE | Background mode. |
| M_FONT | Character font. |
| M_FONT_X_SCALE | Font scaling factor in X. |
| M_FONT_Y_SCALE | Font scaling factor in Y. |
| M_OWNER_SYSTEM | MIL identifier (MIL_ID) 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: |
|-------------------|---------------|
| M_COLOR | double |
| M_BACKCOLOR | double |
| M_BACKGROUND_MODE | long |
| M_FONT | 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()**

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_DEVO
#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_DISPLAY_SETUP           M_DEF_DISPLAY_FORMAT
#define M_DEF_DISPLAY_KEY_COLOR   OL
#define M_DEF_DISPLAY_KEY_ENABLE_ON_ALLOC OL
#define M_DEF_DISPLAY_KEY_DISABLE_ON_FREE OL

```

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*).

```
/* 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 shows the use of the MIL native mode programmer's kit.
*           This example shows how the user can mix MIL code and
*           user code to create a pseudo-MIL function.
*           This example creates a function that ADDs a constant
*           to a LUT buffer before to use that LUT on the display.
*           Note: The Lut must have 256 entry and be 8 bit unsigned.
*/
/* headers. */
#include <stdio.h>
#include <mil.h>
#define MAX_LUT_SIZE    256
#define MAX_LUT_DEPTH  8
#define DIMENSION_ERROR 1

```



```

/* Function definition. */
void AddConstToLut(MIL_ID LutId, unsigned char ConstantToAdd)
{
    MIL_ID      Func;
    short       n, TmpValue;
    unsigned char LutContent[MAX_LUT_SIZE];

    /* Prepare the start of the function and register the parameters. */
    Func = MfuncAlloc("AddConstToLut",2);
    MfuncParamId(Func,1,LutId,M_LUT,M_IN+M_OUT);
    MfuncParamChar(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(LutId,M_SIZE_X,M_NULL) == MAX_LUT_SIZE) &&
              (MbufInquire(LutId,M_SIZE_BIT,M_NULL) == MAX_LUT_DEPTH)) )
        {
            /* Read the Lut content. */
            MbufGet(LutId,LutContent);

            /* Add the constant. */
            for (n = 0; n < MAX_LUT_SIZE; n++)
            {
                /* Calculate the value to write */
                TmpValue = (short)LutContent[n] + (short)ConstantToAdd;
                /* Write the value if no overflow else saturate */
                if (TmpValue <= 0xff)
                    LutContent[n] = (unsigned char)TmpValue;
                else
                    LutContent[n] = 0xff;
            }

            /* Write the result in the Lut. */
            MbufPut(LutId,LutContent);
        }
        else
        {
            /* Report a MIL error. */
            MfuncErrorReport(Func,M_FUNC_ERROR+DIMENSION_ERROR,
                             "Lut dimensions not supported",
                             "Size is not 256 entries or",
                             "Depth is not 8 bit.",
                             M_NULL );
        }
    }
    /* Mark the end of the function. */
    MfuncFreeAndEnd(Func);
}

```

(cont...)

```

/* Main to test the function. */
void main(void)
{
    MIL_ID MilApplication,    /* Application Identifier. */
    MilSystem,               /* System Identifier.      */
    MilDisplay,              /* Display Identifier.     */
    MilImage,                /* Image buffer Identifier. */
    MilLut;                  /* Lut buffer Identifier.  */

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

    /* Load a reference image */
    MbufLoad("Board.mim", MilImage);

    /* Pause */
    printf("Custom pseudo-MIL function creation and usage: \n\n");
    printf("Reference image was loaded, press a key to continue.\n");
    getchar();

    /* Allocate a LUT buffer */
    MbufAllocId(M_DEFAULT, MAX_LUT_SIZE, MAX_LUT_DEPTH, M_LUT, &MilLut);

    /* Set the LUT to a ramp (transparent). */
    MgenLutRamp(MilLut, 0, 0, MAX_LUT_SIZE-1, MAX_LUT_SIZE-1);
    /* Call the Pseudo-MIL function to add an offset (0x40) to the LUT. */
    AddConstToLut(MilLut, 0x40);

    /* Skip target system not supporting display lut */
    if (MsysInquire(MilSystem, M_SYS_TYPE, M_NULL) == M_SYSTEM_PULSAR_TYPE)
    {
        printf("Display LUT not supported on Pulsar. Image not modified.\n");
    }
    else
    {
        /* Use the new LUT for the display. */
        MdispLut(MilDisplay, MilLut);
        /* Pause */
        printf("The white level of the image was augmented using some\n");
        printf("regular MIL functions and a custom pseudo-MIL function.\n");
    }

    printf("Press a key to terminate.\n");
    getchar();

    /* Free the LUT buffer */
    MbufFree(MilLut);

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

```

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.

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.

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.

Note that the arguments tagged as M_OUT will have their internal modification count incremented to signal that they have been modified.

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.

See also **MfuncAlloc()**, **MfuncFreeAndEnd()**, **MfuncParamId()**, **MappControl()**

Index

A

- AC Huffman table 144
- acquisition
 - attribute 39, 224, 228, 232, 263, 267
 - continuous 33, 333
 - image 32, 108, 332
 - input LUT 133
 - precondition 119
- address
 - Host 55
 - logical 55
- allocate
 - application 21, 199
 - buffers 29, 185
 - child buffer 42, 240, 242, 244–245
 - data buffer 36
 - defaults 22, 184, 201
 - digitizer 32, 108, 185, 319
 - display 74, 185, 346
 - graphics context 102, 375
 - image buffer 27, 39
 - LUT buffer 69–70
 - multi-band buffer 132, 232
 - one-dimensional buffer 224
 - pseudo-MIL function 421
 - pseudo-mil identifier 423
 - system 183, 395
 - thread 206
 - two-dimensional buffer 228
- allocation error 41
- analog reference levels 344
- annotation
 - display 86
 - image 102
- application
 - allocate 199
 - building 21
 - control environment 204
 - control module 187
 - free 209
 - inquire environment 220

- pseudo-MIL, parameter checking flag 435
 - pseudo-MIL, parameter registering flag 441
 - simultaneous processing 152
 - starting 183
- application context 154
- arcs, draw 104, 377
- arcs, draw filled 378
- attributes, data buffer usage 224, 228, 232, 263, 267
- auxiliary display 76
 - Matrox Millennium G400, G450, G550 78
 - video output format 77
- auxiliary screen 76
- avi files 273, 277, 297

B

- background color
 - associate to graphics context 103, 379
 - inquire 389
- Bayer cameras
 - formats 62
 - grabbing with 59–60
 - using 59–60
- Bayer images
 - converting to color images 59–60, 62
 - white balancing 64
- Binary buffers 49
- binary buffers, packed 38
- blanking, display 81, 355
- brightness, adjust on input 117
- buffer
 - accessing a 54
 - RGB 47
 - storage format 46–49, 51–52
 - user-allocated 55
- buffers
 - address 55
 - binary 49
 - displayable 39
 - grab 39
 - pitch of 55
 - YUV 49

C

camera

- acquisition from 32, 108
- adjusting/focusing 33
- sophisticated 108
- specification 109, 319

characters, text 106, 193, 394

child buffers 42

- allocate 42, 240, 242, 244–245
- ancestor buffer 301
- attributes 240, 242, 244–245
- data buffer attributes 42
- definition 36
- dimensions 42
- display 43, 85
- display multiple 82
- inheriting parent features 42
- LUT 69
- multiple dimensions 240, 242, 245
- offset 240, 242, 244–245
- offset from parent 42
- parent buffer 301
- physical space 240, 242, 244–245
- purpose 185
- returned coordinates 42, 240, 242, 244–245
- size 42
- type 240, 242, 244–245

circles, draw 104, 377

clear

- buffer 247, 380
- display 81, 355
- graphics image buffer 103

clipping

- borders 43
- data 254
- graphics 105

color

- handling techniques 131
- input LUT 133

color band 37, 232, 263, 267

color images

- allocate buffer 132, 232
- allocate child buffer 240, 242
- copy 255, 257
- copy single band 43
- create buffer 267

dealing with 132

displaying 134, 368–369

grabbing 132, 332–333

loading 135, 306, 318

put data in band 44

reference levels 119

saving 135, 306, 318

command reference

- order 196
- quick reference 187
- status section 196

commands

Function DevelopersToolkit' 419

functions 22

MIL, command summary 187

predefined constants 196

pseudo-MIL 164, 416

communication channels 21, 24, 183

compensation

- memory 205
- processing 205

compiling 23

compressing images 140

conditional buffer, creating 102

continuous grab 33, 333

contrast 118

image, adjusting 117

control

application environment settings 204

areas processed 42

buffer features 248

digitizer 322

display 350

graphic context 382

messages, error 204, 206

parameter checking 205

processing compensation 205

system processing 397

thread 206

timer 223

trace mechanism 204

conversion

data format 45, 274, 293

coordinates

child buffer 42

of a pixel 58

text writing 106

- copy
 - bit truncation/extension 44
 - clip, and 43, 254
 - color band 43, 70, 255, 257
 - conditional 43, 260
 - data 43, 253, 255, 257, 260, 280–281, 283, 287, 307–308, 310, 314–315
 - data line 312
 - data to LUT 70
 - mask 43, 262
 - specific buffer areas 43
- Corona-II
 - exposure 125, 128–130
 - automatic model 127
 - bypass model 129
 - triggers 125, 128–130
- CPU
 - CPU-assisted display 87
- custom
 - window, VGA 90

D

- data allocation and access module 35, 188
- data buffer
 - allocation 224, 228, 232, 263, 267
 - ancestor 301
 - attributes 38, 41, 224, 228, 232, 263, 267
 - automatically allocated 45
 - child 36, 42
 - clear 103, 247, 380
 - clip border 43, 254
 - color band 37, 132, 232, 263, 267
 - control 248
 - copy 253
 - copy color 255, 257
 - copy theoretical line 287
 - defined 36, 185
 - depth 38, 224, 228, 232, 263, 267
 - dimensions 37
 - display 73
 - export data 45, 274
 - free 37, 279
 - get data, put in array 44
 - handling 35
 - import 45
 - import data 293
 - incorrect usage 41

- inquire 272, 300
- integer 38
- intended usage 39
- load 274
- location 38
- LUT, see LUT buffer 69
- management 44
- multiple dimensions 232, 267
- multiple, display 43
- multiple, handling 43
- packed binary 38
- parent 301
- pseudo-MIL, modification 434
- put data 44, 307–308, 310, 314–315
- range 38, 224, 228, 232, 263, 267
- restore 45, 316
- retrieve data 280–281, 283, 289–290
- save 45, 274, 318
- sign 224, 228, 232, 263, 267
- two-dimensional 263
- type 38, 232, 263, 267
- write data 312
- data format, input device 109, 319
- data generation
 - LUT 69, 373
 - module 192
- data objects, manipulation concepts 184
- data type 29
- data, overwriting 29
- DC Huffman table 144
- dcf files 109
- debugging 186
- decompressing images 140
- default graphics context 102
- defaults
 - application 201
 - display 411
 - free 210
 - image buffer 18, 22, 29, 31, 412
 - initialization flag 410
 - initializing 18
 - input device 108, 412
 - input LUT 119
 - setup 410
 - system 411
- depth
 - data buffer 38
- destination buffer 29

- device
 - control module 107
- digitizer
 - allocate 32, 108, 319
 - color data format 132
 - configuration format 109
 - control 322
 - data format 319
 - event hook 336
 - free 108, 331
 - input channel 110, 321
 - inquire 109, 339
 - LUT 72, 119, 133, 343
 - number 319
 - reference levels 117, 344
- digitizer configuration files 109
- display
 - allocation 21, 346
 - annotation 86
 - Windows GDI 86, 88
 - border handling 74
 - buffer 29, 43
 - clear 81
 - color 96
 - color image 134
 - control behavior 350
 - control module 73, 191
 - CPU-assisted 87, 100
 - device number 79
 - example 91
 - format 346
 - free 91, 356
 - image buffer 368–369
 - image location 74
 - inquire 359
 - keying 87
 - LUT 71, 96, 134, 363
 - memory 74
 - monochromatic effect 71
 - monochrome buffer 28
 - multiple buffers 82
 - non 8-bit buffers 80
 - number 346
 - pan 85, 367
 - pseudo-color effect 71
 - pseudo-color LUT 97
 - scroll 85, 367
 - size and depth 79

- true color effect 72
- user-defined window 90
- zoom 85, 370
- display type 75
 - auxiliary 76
 - Matrox Millennium G400, G450, G550 78
 - video output format 77
- windowed 75
 - extended desktop 75
- Displayable buffers 39
- dots, draw 104, 384
- double buffering, definition 121
- DrawDIBDraw()
 - VGA 347–348
- drawing 102, 104
- dynamic range 118

E

- edge
 - rising/falling 326–327
- error reporting
 - appendix 185
 - automatic 31
 - code 211
 - hook 217, 357
 - memory, insufficient 41
 - message control 22, 204, 206
 - messages 186, 211
 - pseudo-MIL function 424, 428
 - screen 185
 - suberror code 211
 - thread 154–155
 - use 185
- examples
 - color, run with 133
 - display in user-defined window 91
 - display multiple buffers 82
 - Function DeveloppersToolkit' 416
 - general information 18, 24
 - grab 32
 - image allocation/display 30
 - installing 18
 - MIL sample program 24
 - mmultdis.c 82
 - modify for color 28
 - mstart.c 24

- mwindisp.c 91
- Native Mode ProgrammersToolkit' 165
- pseudo-function development 416
- standard defaults 28
- export data buffer 45, 274
- exposure 130
 - automatic model
 - Corona-II 127
 - bypass model
 - Corona-II 129
 - Corona-II 125

F

- field
 - grabbing 111
- field grabbing 326
- file format 45, 274
- files
 - avi 273, 277, 297
- filled-in shapes 105, 378
 - boundary-type seed fill 385
- font
 - associate to graphics context 106, 386
 - inquire 389
 - predefined 106
 - scale 106, 387
 - size 106, 387
- foreground color
 - associate to graphics context 103, 381
 - fill with 105
 - inquire 389
- frame
 - grabbing 33, 111
- frame buffer 74
- free
 - application 209
 - application defaults 210
 - buffer, data 37, 279
 - defaults 184, 210
 - digitizer 331
 - display 356
 - graphics context 102, 192, 388
 - pseudo-MIL function 426
 - pseudo-MIL identifier 427
 - system 402

- function
 - development 165, 416
 - execution success 22
 - hook 186, 217, 336, 357
 - pseudo-MIL, allocate 421
 - pseudo-MIL, example 416
 - pseudo-MIL, free 426
 - pseudo-MIL, start 443
 - user-created 15
- Function Developers Toolkit 415
- Function DevelopersToolkit
 - command summary 419
 - example 416
- functions
 - commands 22
- functions See also, commands 196

G

- gamma correction 72
- Genesis
 - system 396
- global library state 410
- Grab 39
- grab 119
 - color images 132
 - continuous 33, 100, 333
 - data buffer 39, 132
 - example 32
 - frames 33, 111
 - halt 33, 335
 - image 32, 108, 332
 - mode 121
 - monochrome 32
 - multi-dimensional buffers 132
 - scale 323
 - synchronization 121
 - wait 334
- Grab buffers 39
- GrabAndWarp()
 - example 165
- grabbing
 - single frame 111
- graphics 102
 - arcs, draw 104, 377
 - boundary type seed fill 105, 385
 - buffer, clear 103, 380
 - capabilities 14

- circles, draw 104, 377
- clipping 105
- dots, draw 104, 384
- filled elliptic arcs, draw 104, 378
- filled rectangles, draw 104, 193, 393
- filled-in shapes 104
- lines, draw 104, 193, 391
- module 101
- outline, draw 104
- parameters 103
- rectangles, draw 104, 392
- text, write 106, 193, 394
- graphics context
 - allocate 102, 375
 - background color, associate 103, 379
 - control 382
 - default 102, 376
 - definition 102
 - font scale, associate 106, 387
 - foreground color, associate 103, 381
 - free 102, 192, 388
 - inquire 389
 - object parameters 103
 - text font, associate 106, 386
- graphics controller
 - memory 74

H

- halt grabbing 33, 335
- header file 23
- hook
 - digitizer event 336
 - error 217, 357
 - get information 214
 - to an event 217
 - trace 217, 357
 - user-defined function 186
- Host
 - communication 183
 - screen 185
- host
 - communication 21
 - CPU 14
 - default system 37, 199, 202, 224, 228, 232, 295, 375, 395
 - screen 22
 - system 22, 70, 373, 395

- hue 243, 255, 258, 282, 284, 309, 311
 - HLS 240
- Huffman encoding 143–144

I

- identifier, MIL objects 196
- Image
 - placement 85
- image
 - grabbing 32
- image buffer
 - acquisition 39
 - allocation 27, 29–30, 38–39
 - clear 247, 380
 - color 28, 33
 - conditional 102
 - default 22, 29, 133
 - defined 29
 - destination 29
 - display 30, 39
 - display border 74
 - display multiple 82
 - display position 74
 - free 37
 - map through LUT 68
 - removing from display 81, 355
 - select for display 74, 368
 - select window for display 369
 - size 29
 - source 29
 - two-dimensional 28–29, 33
 - uses 29
- import data 293
- include file 23
- initialization
 - default 184, 201
 - input device 108
 - system 18, 108, 183
- input device
 - brightness 117
 - contrast 117
 - control module 190
 - defaults 108
 - frequency 109
 - line-scan 111
 - LUT 119
 - reference level 341

- resolution 109
- subsampling 120
- using 32
- input signal 324
- inquire
 - application environment 220
 - data buffer 272, 300
 - digitizer 109, 339
 - display 359
 - graphics context 389
 - system 291, 403
- installation
 - MIL 18
 - test program 23
- integer buffers 38
- Intellicam 109
- intensity
 - correction 70
- interlaced JPEG compression 140

J

- JPEG
 - discrete cosine transform 145
- JPEG compression 140

K

- kernels
 - buffer allocation 39
- keying 87
- inquire 359

L

- lines
 - draw 104, 193, 391
- line-scan device 111
- link program with library 23
- load
 - color image 135
 - data 44–45, 306
 - LUT data 70
- look-up table
 - 1-band custom 97
 - 3-band custom 98
 - changing default 97

- control loading into physical output LUTs 94
- pseudo-color 97
- lossless compression, JPEG 140
- lossy compression, JPEG 140
- luminance
 - HLS 240, 243, 255, 258, 282, 284, 309, 311
- LUT
 - 1-band custom 97
 - 3-band custom 98
 - changing default 97
 - control loading into physical output LUTs 94
 - pseudo-color 97
- LUT buffer
 - allocation 38, 69
 - child buffer 69
 - color bands 69, 133
 - data generation 69–70
 - dimensions 69
 - load 70
 - management 69
 - one-dimensional 69
 - restore 70
- LUTs
 - custom 97
 - data generation 371, 373
 - definition 68
 - display 71, 134, 363
 - display color, change 96
 - general information 67
 - index 69
 - input 117, 119, 133, 343
 - input mapping 72
 - intensity correction 70
 - monochromatic effect 71
 - multiple-color-band 99
 - one-color-band 98
 - pseudo-color effect 71
 - ramp 69
 - true-color effect 72
 - usage 71

M

M_DISP 39
M_GRAB 39
macros 197
Mapp...() 187, 199
MappAlloc() 22, 102, 154, 183, 199
 example 91
MappAllocDefault() 18, 21, 29, 31–32, 74,
 102, 108, 132–134, 184, 201, 410
 example 24, 30, 32–33, 82, 165, 416
MappControl() 22, 185–186, 204
MappControlThread() 155, 206
MappFree() 22, 183, 209
 example 91
MappFreeDefault() 21, 184, 210
 example 24, 30, 32–33, 82, 165, 416
MappGetError() 22, 155, 186, 211
 example 30
MappGetHookInfo() 214
MappHookFunction() 22, 186, 217
MappInquire() 220
MappModfiy() 222
MappTimer() 223
mask, copy 43, 262
Mbuf...() 188
MbufAlloc() 54
MbufAlloc1d() 36, 69, 97, 185, 224
 example 416
MbufAlloc2d() 29, 36, 82, 185, 228
 example 30, 91
MbufAllocColor() 22, 36, 46, 69, 132, 185, 232
MbufBayer() 59–60, 62, 64, 237
MbufChild1d() 42, 244
MbufChild2d() 42, 82, 245
 example 82
MbufChildColor() 42, 240
MbufChildColor2d 242
MbufClear() 103, 247
 example 82, 91
MbufControl 88
MbufControl() 88, 97–98, 141, 148, 248
 example 165
MbufCopy() 70, 141, 253
MbufCopyClip() 43, 254
MbufCopyColor() 43, 70, 255
MbufCopyColor2d() 257
MbufCopyCond() 43, 260
MbufCopyMask() 43, 262
MbufCreate2d() 55, 263
MbufCreateColor() 55, 267
MbufDiskInquire() 272
MbufExport() 45, 135, 141, 274
MbufExportSequence 142
MbufExportSequence() 277
MbufFree() 22, 37, 42, 69, 279
 example 82, 416
MbufGet() 44, 54, 280
 example 416
MbufGet1d() 44, 289
MbufGet2d() 290
MbufGetColor() 44, 281
MbufGetColor2d() 283
MbufGetHookInfo() 285
MbufGetLine() 287
MbufHookFunction 291
MbufHookFunction() 291
MbufImport() 45, 135, 141, 293
MbufImportSequence() 142, 297
MbufInquire() 55–56, 88, 300
 example 165, 416
MbufLoad() 45, 70, 135, 306
 example 82, 416
MbufPut() 44, 54, 70, 97, 307
 example 416
MbufPut1d() 44, 70, 314
MbufPut2d() 315
MbufPutColor() 44, 70, 308
MbufPutColor2d() 310
MbufPutLine() 312
MbufRestore() 45, 70, 135, 316
MbufSave() 45, 135, 318
Mdig...() 190
MdigAlloc() 22, 32, 108–110, 132, 319
 example 91
MdigChannel() 110, 321
MdigControl() 33, 121, 153, 322
MdigFree() 22, 108, 132, 185, 331
 example 91
MdigGrab() 33, 121, 132, 141, 332
 example 32
MdigGrabContinuous() 33, 132, 333
 example 33, 91
MdigGrabWait() 121, 334
MdigHalt() 33, 121, 335
 example 33, 91

- MdigHookFunction() 153, 336
- MdigInquire() 109, 339
 - example 165
- MdigLut() 72, 119, 133, 343
- MdigReference() 119, 133, 344
- Mdisp...() 191
- MdispAlloc() 22, 29, 74–75, 87, 134, 346
 - device number 79
 - example 91
 - M_AUXILIARY 77
 - M_WINDOWED 75
- MdispControl() 80, 88, 350
- MdispDeselect() 29–30, 81, 134, 355
 - example 82, 91
- MdispFree() 22, 81, 185, 356
 - example 91
- MdispHookFunction() 89, 357
- MdispInquire() 88, 99, 359
- MdispLut() 71, 97–98, 134, 363
- MdispOverlayKey() 87, 365
- MdispPan() 85, 367
- MdispSelect() 29, 39, 43, 74, 82, 134, 368
 - example 82
 - VGA 90
- MdispSelectWindow() 90, 369
 - example 91
 - VGA 90
- MdispZoom() 85, 370
 - example 82
- memory
 - compensation 205
 - insufficient 41
 - resources 21–22
- messages, error 22, 31
- Meteor
 - system 395
- MfuncAlloc() 421, 436–437, 439–440
 - example 416
- MfuncAllocId() 423, 430, 432–433
- MfuncErrorReport() 424
 - example 416
- MfuncFreeAndEnd() 424, 426
 - example 416
- MfuncFreeId() 427
- MfuncGetError() 428
- MfuncIdGetObjectType() 430
- MfuncIdGetUserPtr() 431
- MfuncIdSetObjectType() 432
- MfuncIdSetUserPtr() 433
- MfuncModified() 434
- MfuncParamChar()
 - example 416
- MfuncParamCheck() 435
 - example 416
- MfuncParamDouble() 436
- MfuncParamId() 437, 443
 - example 416
- MfuncParamLong() 439
- MfuncParamPointer() 440
- MfuncParamRegister() 441
- MfuncParamString() 442
- MfuncStart() 421, 426, 436–437, 439–440, 442–443
 - example 416
- Mgen...() 192
- MgenLutFunction() 69–70, 371
- MgenLutRamp() 69, 97, 373
 - example 416
- Mgra...() 192
- MgraAlloc() 102, 375
- MgraArc() 104, 377
- MgraArcFill() 104, 378
- mgrab.c 413
- MgraBackColor() 103, 379
- MgraClear() 103, 380
- MgraColor() 103, 381
- MgraControl() 382
- MgraDot() 104, 384
- MgraFill() 104–105, 385
- MgraFont() 106, 386
- MgraFontScale() 106, 387
- MgraFree() 102, 388
- MgraInquire() 389
- MgraLine() 104, 391
- MgraRect() 104, 392
- MgraRectFill() 104, 393
- MgraText() 106, 394
 - example 24, 91
- MIL
 - file format 45, 274
 - header file 23
 - include file 23
 - objects 15, 196, 222
 - running application 23, 183
 - structure 182

MIL modules

- application 187
- data allocation and access 188
- data generation 192
- digitizer control 190
- display allocation 191
- display control 73, 191
- graphics 101, 192
- I/O device control 107
- system device 193

mil.h 23, 184, 197

mil.ini

- Meteor-II 115

milsetup.h 18, 21–22, 28, 108, 133, 184, 201, 210, 410

MimBinarize()

- example 82

MimHistogramEqualize() 70

MimLutMap()

- example 416

mmultdis.c 82

MMX Technology, Intel 16

mnatfct.c 416

mnatgen.c 165

monochromatic effect 71

monochrome image buffer 29

mstart.c 23

Msys...() 193

MsysAlloc() 22, 183, 395

- example 91

MsysControl() 397

- example 165

MsysFree() 22, 183, 402

- example 91

MsysInquire() 403

- example 91, 165

multi-dimensional buffers 132

multiple buffers

- displaying 82

multi-processing 151–152

- definition 152

multi-threading 151, 153

- definition 153

mwindisp.c 91

N

native mode 163, 415

- example code 165
- flag 410
- integrating with MIL 164
- interface 164
- portability 416

non 8-bit buffers

- displaying 80

O

object identifier 196

object type

- pseudo-MIL function 430
- pseudo-MIL, assign 432

open communication 21, 24, 183

overlay

- buffer 86
- behavior 87
- flickering 100
- simulated 100

overwriting data 29

P

packed binary buffers 38

palette

- image 69

panning, display 85, 367

parameter

- double, pseudo-MIL 436
- long, pseudo-MIL 439
- MIL_ID, pseudo-MIL 437
- null-terminated string, pseudo-MIL 442
- pointer, pseudo-MIL 440

parameter checking control 205

parent buffer 36, 42, 185

- display 82, 85

physical memory 29

- buffer allocation 38

pitch 55

pixel

- coordinates 58
- depth 15
- value, minimum/maximum 117

- pointer
 - pseudo-MIL object 431
 - pseudo-MIL object, assign 433
- portability
 - native mode 164
- portability, native mode 416
- predictive coding 143
- preprocess
 - input data 119
- processing
 - attribute 224, 228, 232, 263, 267
 - compensation 397
 - control 205
 - limiting 42
 - system, force 397
- program examples 18
- pseudo-color
 - effect 71
- pseudo-MIL commands 416
- pseudo-MIL functions 416, 421
- put data
 - 1D data buffer 314
 - 2D data buffer 315
 - array, from 44
 - data buffer 307–308, 310

Q

- quantization
 - JPEG 145

R

- ramp, LUT 69
- read.me 18, 20, 23–24
- rectangles, draw 104, 392
- rectangles, draw filled 193, 393
- reference level
 - analog 117
 - black/white 117, 341, 344
 - controls 117, 344
 - digitizer 341
 - input channel 117, 341
- reporting errors 185
- resident software, required 410
- restart markers 145

- restore
 - data buffer 316
 - LUT buffer 70
- retrieve data
 - 1D data buffer 289
 - 2D data buffer 290
 - color bands 281, 283
 - data buffer 280–281, 283
- RGB
 - buffers 47

S

- sample program 23
- saturation
 - HLS 240, 243, 255, 258, 282, 284, 309, 311
- save
 - color image 135
 - data 44–45, 274, 318
- scale, input 119, 323
- scaling 119
- scrolling, display 85, 367
- seed fill, boundary-type 385
- select
 - digitizer input channel 321
 - image to display 368
- setup flag 410
- size
 - child buffers 42
 - data buffer 37
 - image buffer 29
 - LUT buffer 69
 - text character 106
- software triggers
 - Corona-II 130
- source buffer 29
- speed
 - multi-threading 153
- stop grabbing 33, 335
- storage area 29
- strobe device 109
- structure, MIL 182
- structuring elements
 - buffer allocation 39
- subsampling input 119

- synchronization
 - of grab 121, 325
 - thread 153–154
 - with grab end 325
- system
 - allocation 21, 395
 - buffers 29
 - configuration 18
 - control behavior 397
 - default 15, 18
 - default setup configuration 410
 - definition 14
 - device 71, 132, 134, 183–184
 - display criteria 29
 - free 402
 - Genesis 396
 - grab criteria 33
 - Host 395
 - initialization 18, 108
 - inquire 291, 389, 403
 - Meteor 395
 - module 193
 - multiple 22
 - multi-processing capabilities 152
 - number 395
 - type 395
 - VGA 395

T

- target system
 - system 15
- test installation program 24
- text
 - character font 106, 386
 - character size 387
 - graphics 106
 - support 102
 - write 193, 394
- theoretical data line 287, 312
- thread
 - allocate or control 206
 - application context 154
 - data sharing 153
 - error reporting 154–155
 - multi-threading 151, 156
 - synchronization 154
- TIFF file format 274

- timer control 223
- toolkit
 - Function Developers' 163, 415
 - Native Mode Programmers' 415
- trace
 - application 186
 - hook 217, 357
 - mechanism control 204
- transforming data 45, 274, 293
- trigger device 109
- triggers 128, 130
 - Corona-II 125, 129
- true color effect 72

U

- user-allocated buffer 55

V

- VCF (Video Configuration Format) 346
- VGA
 - system 395

W

- wait, grab 334
- White balance
 - and Bayer images 64
 - determining coefficients
 - monochrome 65
 - RGB 65
 - YUV 65
- Window occlusion
 - Meteor-II 117
- windowed display 75
 - extended desktop 75
- Windows
 - custom window, VGA 90
- Windows desktop screen(s) 75
- Windows GDI annotations 86, 88
- Windows NT
 - display
 - size and depth 78
 - extended desktop restriction 75

X

xfontscale, inquire 389

Y

yfontscale, inquire 389

YUV buffers 49

Z

zoom

display 85, 370

Product Support

Product Assistance Request Form

[illegible]

