

MIL-Lite

version 6.1

User Guide and Command Reference

Manual no. 10514-801-0610

March 1, 2000

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

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

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

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

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

Texas Instruments is a trademark of Texas Instruments Incorporated.

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

© Copyright Matrox Electronic Systems Ltd., 2000. All rights reserved.

Disclaimer: Matrox Electronic Systems Ltd. reserves the right to make changes in specifications at any time and without notice. The information provided in this document is believed to be accurate and reliable. However, no responsibility is assumed by Matrox Electronic Systems Ltd. for its use; nor for any infringements of patents or other rights of third parties resulting from its use. No license is granted under any patents or patent rights of Matrox Electronic Systems Ltd.

PRINTED IN CANADA

Contents

Chapter 1: Getting started 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.	20
Distributing your MIL application	24

Chapter 2: Allocating an image buffer and grabbing images. 27

Getting started	28
Allocating and displaying an image buffer	29
Grabbing images.	32

Chapter 3: Specifying and managing your data buffers. 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

Chapter 4: Lookup tables 59

Lookup tables	60
LUTs and data buffers	61
Loading and generating data into LUTs	61
Generating data directly into the LUT buffer.	61
Loading LUTs with precalculated data	62
Using LUTs	63
Displaying using LUTs.	63
LUTs and digitizers	64

Chapter 5: Displaying an image65

Displaying an image	66
Display configuration	67
Single-screen configuration.	67
Dual-screen configuration.	67
Multi-head display configuration.	68
Display modes and the display window	69
Displaying in windowed-mode.	69
Displaying in non-windowed mode	69
Display size and depth	70
Displaying buffers of different data depths	70
Removing a buffer from the display	71
Displaying multiple buffers	72
Panning, scrolling, and zooming	75
Annotating the displayed image non-destructively	76
Using GDI annotations	78
Displaying an image in a user-defined window	81
Using the MdispSelectWindow() function. . .	81
LUTs and changing the displayed colors or gray levels.	85
Changing the default LUT values	86

Different display architectures in windowed mode	88
Underlay display architecture	89
Overlay/regular display architecture	90
DirectDraw underlay-surface display architecture.	91
Advanced controls for windowed mode	92
Display types in windowed mode	92
Zoom types in windowed mode.	93
Controlling how the LUT buffer is loaded into the Windows palette	94
Controlling how the logical palette is loaded into the physical output LUTs	94
<hr/>	
<i>Chapter 6: Generating graphics</i>	<i>97</i>
MIL and graphics	98
Preparing for graphics	98
Drawing graphics	100
Writing text	102
<hr/>	
<i>Chapter 7: Grabbing with your digitizer</i>	<i>103</i>
Cameras and input devices	104
The data format	105
The digitizer number	106
Multiple cameras	106
Number of frames or fields	107
Grabbing to the display	108
Live and pseudo-live continuous grabs. . .	108
Live transfer to the display.	109
Pseudo-live transfers to the display	109

Window occlusion	111
Reference levels, lookup tables, and scaling. .113	
Black and white reference levels	113
Color image reference levels	114
Mapping grabbed data through a LUT . . .114	
Scaling	115
Optimizing application performance when grabbing	116
Grab mode	116
Double buffering.	117
Multiple buffering.	119
Grabbing a sequence of frames in real-time.	120
Grabbing with triggers and exposures	120
Asynchronous reset mode.	121
Triggers and exposures.	122
Software triggers.	125
<hr/>	
<i>Chapter 8: Color</i>	127
Dealing with color	128
Grabbing.	128
Displaying.	130
Saving and loading color images	131
How to manage your color buffer.	131

Chapter 9: JPEG compression 135

Introduction	136
General steps	137
Controlling a JPEG compression	138
JPEG lossless	138
JPEG lossy	139
Using your own table	140
Restart markers	141

Chapter 10: Data manipulation with multiple systems 143

Data manipulation with multiple systems. . .	144
--	-----

Chapter 11: Using MIL with multi-processing and under multi-thread systems 145

Multi-processing	146
Multi-threading	147
MIL and multi-threading	148

Chapter 12: Using MIL with Native Mode Functions . 157

Integrating native functions with MIL code . .	158
Portability	158
Signaling MIL about Native Mode use. . .	158
A native mode example.	159

Chapter 13: Programming with MIL165

A MIL overview	166
Starting your MIL application	167
Header file and libraries	168
MIL object manipulation concepts.	168
Error handling	169
Tracing an application	170
A quick command reference	171
The application allocation and control module	171
The buffer allocation and access module	172
The digitizer allocation and control module	174
The display allocation and control module	175
The basic data generation module.	176
The basic graphics module	176
The system allocation and inquiry module	177

***Chapter 14: The command reference
descriptions.179***

The reference description notes	180
---	-----

Appendix A: The default setup configuration file . . .385

The default setup configuration file	386
When you do not want to use defaults.	390

<i>Appendix B: The MIL Function Developer's Toolkit</i>	<i>393</i>
The MIL Function Developer's Toolkit	394
An example using the Function Developer's Toolkit	394
MIL Function Developer's Toolkit Command Reference	397

<i>Appendix C: Troubleshooting</i>	<i>423</i>
Error reporting	424
Error messages explained	425
Driver error messages explained	437

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/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.

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.

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

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

Supported compilers

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

Getting started

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

After installation, read the *readme.txt* file in the *\MIL* 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

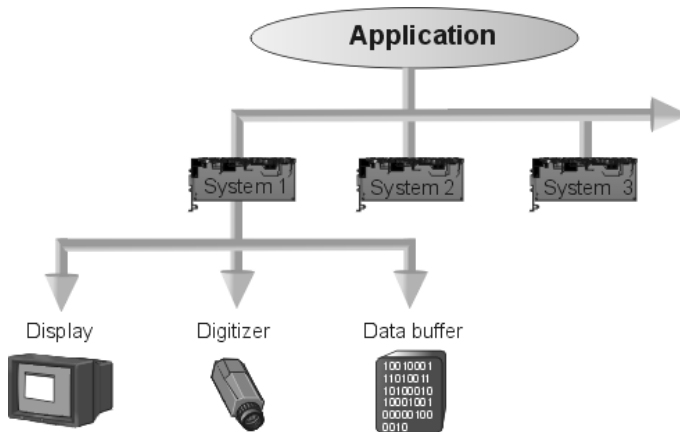
The MIL Configuration utility, located in your Matrox Imaging\MILConfig directory, provides licensing, DMA configuration, and system information tools. For example, if you need to change the amount of reserved memory or if you change the amount of physical memory in your computer, you can change the 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.

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		Board Libraries	
Library	Description	Library	Description
mil.lib	Core library	mil1394.lib	Matrox Meteor-II/1394 library.
milvga.lib	VGA library.	milgen.lib	Matrox Genesis library.
		milmet2.lib	Matrox Meteor-II/Standard/MultiChannel library.
		milmet2D.lib	Matrox Meteor-II/Digitizer
		milorion.lib	Matrox Orion library.
		milpul.lib	Matrox Pulsar library.

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

We have provided a sample program, *mstart.c*, that allows you to test the installation process and become familiar with running a MIL application. This test program allocates the application, opens communication with the default target system, displays a welcoming message, pauses, and frees the system resources.

```

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

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

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

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

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

    /* Print a message on the Host screen. */
    printf("\n");
    printf("\\"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 Intellicam.

Examples in general

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

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

Distributing your MIL application

To distribute your MIL application, you will have to distribute MIL run-time DLL files with your application.

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

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

- Normal distribution
- Silent distribution

Normal distribution

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

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

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

Silent distribution

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

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

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

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

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 must be allocated on the same system.

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

```

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

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

#define IMAGE_DEPTH 8L

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

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

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

    (cont. ...)

```

```

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

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

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

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

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

```

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

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

Displaying multiple buffers

With MIL, you can also display multiple buffers. This is discussed later in the manual, in *Chapter 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: Input devices and digitizers*.

*Continuous grabbing
and adjusting your
camera*

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

This is discussed in greater detail in *Chapter 7: Input devices and digitizers*. The following example is of adjusting a camera using a continuous grab.

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

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

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

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

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

(cont. ...)

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

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

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

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

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.

Data buffers

Data buffers

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

Allocating data buffers

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

When allocating a data buffer, you must specify its:

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

Controlling specific parts

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

Target system

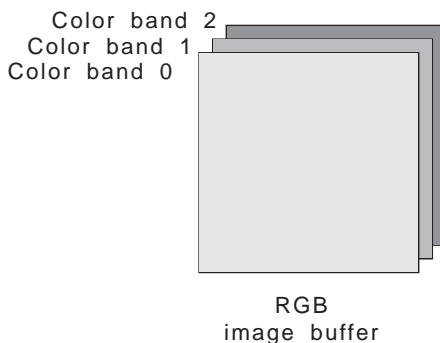
A data buffer is allocated on the specified system. If the `M_DEFAULT_HOST` system is specified, the default Host system of the current MIL application will be used. If `M_DEFAULT` is specified, MIL will select the most appropriate system on which to allocate the data buffer (it can be the default Host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

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

Specifying the dimensions of a data buffer

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

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



Certain MIL functions support manipulating multi-band image buffers. See *Chapter 21: Color 8* 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).

Allocating an image buffer

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

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 surface (maintained directly or through a DIB) for display purposes (not necessarily the same size). When the Host buffer is modified, its associated buffer in the frame buffer surface is automatically updated. When displaying a buffer, both the buffer and the display must have been allocated on the same system.

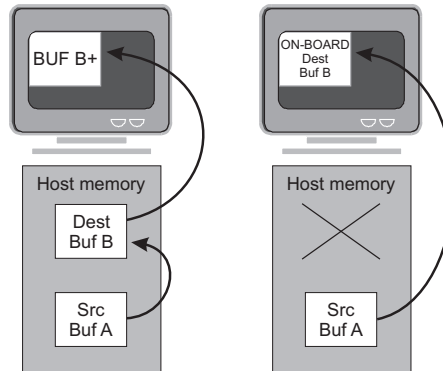
When grabbing a single frame into a displayable buffer, MIL grabs into the Host memory version of the buffer and then updates the display of the buffer. When grabbing continuously, the grab is made directly to the frame buffer surface and then at the end of the grab, the Host buffer is updated.

Overriding the default allocation sequence

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

Overriding the default allocation sequence is useful when allocating a displayable buffer under any non-windowed mode. If you are not using the displayable buffer for processing or are

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



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

Internal format of the buffer

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

Insufficient memory

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

Inappropriate data buffer usage

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

Manipulating and controlling certain data buffer areas

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

Child buffers

Child buffers are subsets of parent buffers

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

Allocating child buffers

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

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

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

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

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

Copying specific buffer areas

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

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

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

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

Managing data buffers

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

Putting and retrieving data

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

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

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

Loading a data buffer

You can load data, using one of two methods:

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

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

Saving a data buffer

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

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

Loading and saving a sequence of data buffers

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

Controlling how color image buffers are stored

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

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

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

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

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

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

RGB buffers

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

RGB data formats

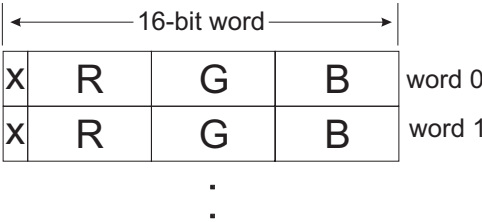
BGR24 packed (M_BGR24+M_PACKED) is a format whereby each pixel is internally stored as three consecutive bytes in little-endian order, that is:

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

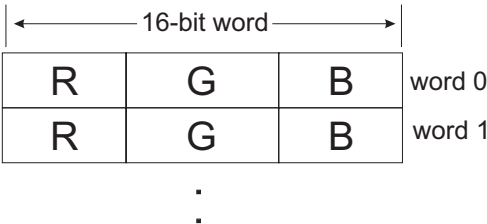
BGR32 packed (M_BGR32+M_PACKED) is a format whereby each pixel is internally stored as four consecutive bytes, in little-endian order. The most-significant byte is a "don't care" byte, as shown below:

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

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



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



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

Binary buffers

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

YUV buffers

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

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

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

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

The supported YUV formats are:

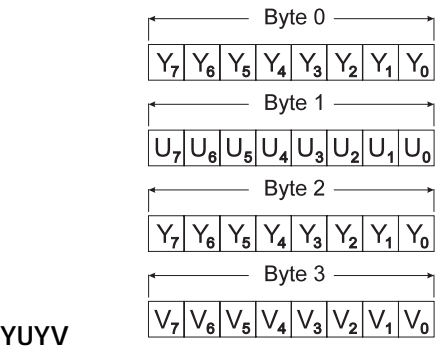
- YUV16 Packed
- YUV9 Planar
- YUV12 Planar
- YUV16 Planar

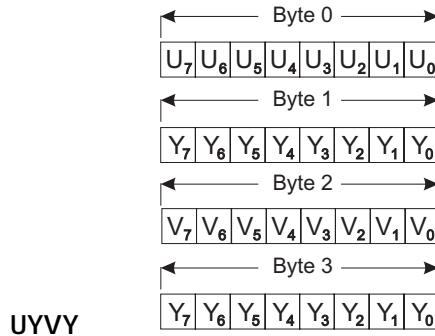
YUV16 Packed

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

The YUV16 packed data format has two available formats: YUYV and UYVY. The only difference between these two YUV formats is the ordering of data in the buffer. Certain digitizer boards grab data in exclusively YUYV or UYVY packed data format. In addition, certain display adapters are optimized to handle YUYV format (or can only handle the YUYV format in their underlay).

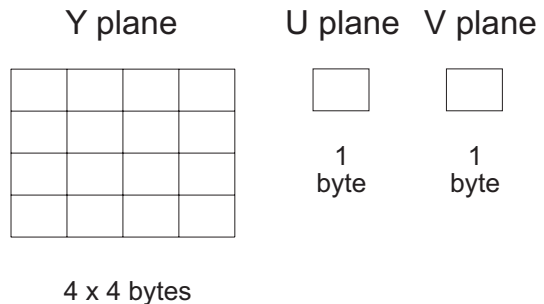
When you allocate an M_YUV16+M_PACKED buffer, MIL allocates the buffer in the format that is most suitable for the selected platform and the specified buffer attributes. You can, however, force a format using the M_YUV16_YUYV or M_YUV16_UYVY control types. When the buffer has an M_GRAB attribute, forcing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





YUV9 Planar

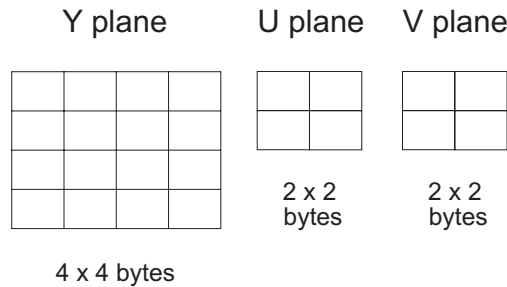
YUV9 Planar (M_YUV9+M_PLANAR) is a planar format whose components have a depth of one byte but are not of the same size. Although each pixel has a corresponding 1 byte Y (luminance) component, each block of 16 pixels share the same one byte of U (chrominance U) and the same one byte of V (chrominance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



YUV12 Planar

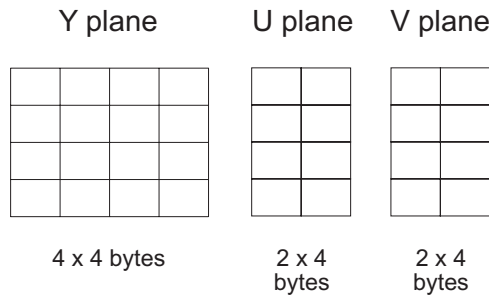
YUV12 Planar (M_YUV12+M_PLANAR) is a planar format whose components have a depth of one byte but are not of the same size. Although each pixel has a corresponding 1 byte Y (luminance) component, each block of 4 pixels share the same one byte of U (chrominance U) and the same one byte of V

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



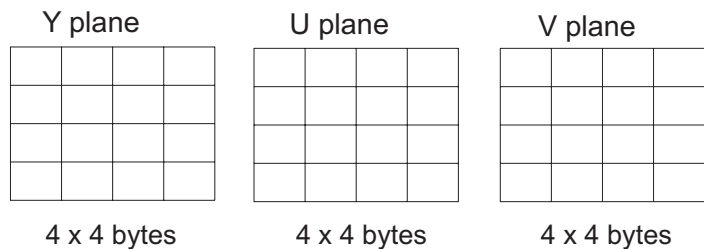
YUV16 Planar

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



YUV24 Planar

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



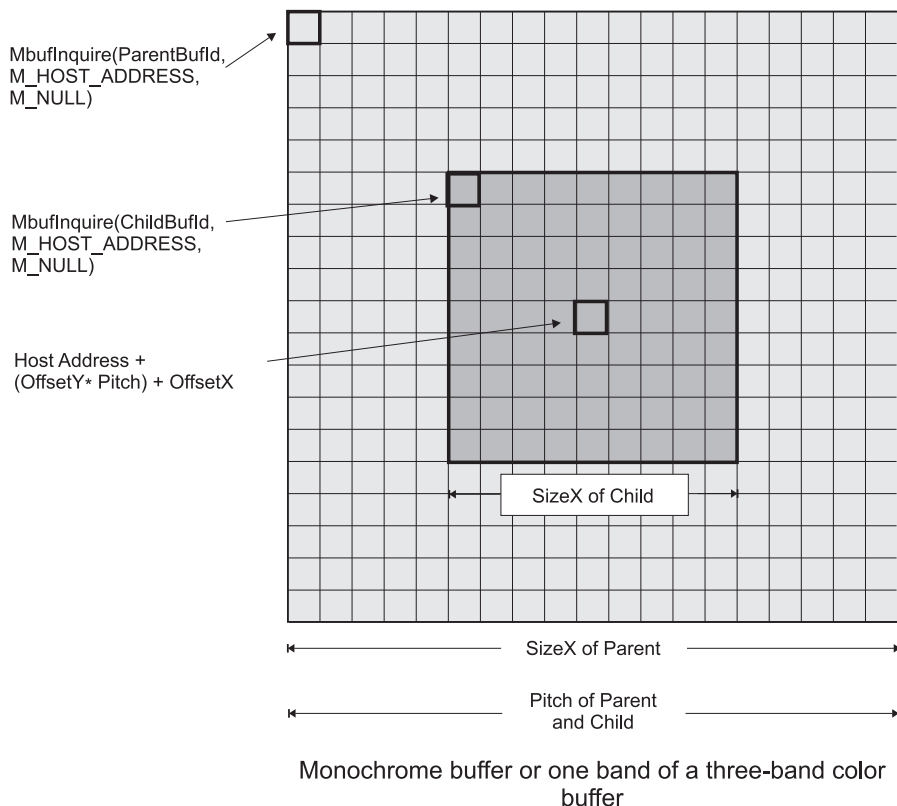
Child YUV buffers

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

Accessing a MIL buffer directly

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

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



Address

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

Pitch

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

Mapping a data buffer to user-allocated memory

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

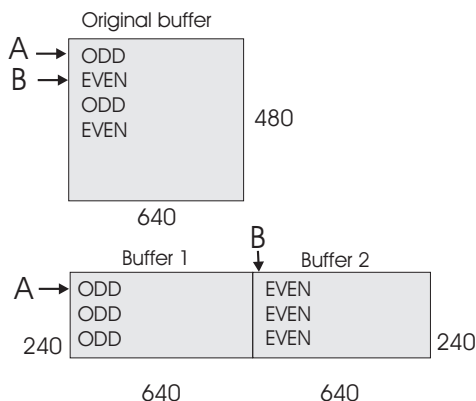
When creating a buffer with *MbufCreateColor()*, you must pass an array of pointers to the addresses of the data. For packed color buffers, you must pass an array of one pointer; for planar buffers, you must pass an array with the same number of pointers as the number of bands in the buffer. When creating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

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

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

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

Create Buffers 1 and 2 as follows:



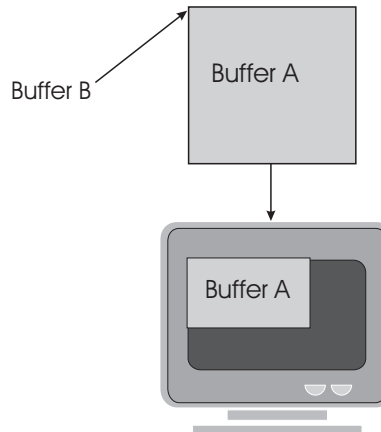
■ Buffer 1: (Odd field)

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

■ Buffer 2: (Even field)

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

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



See *with multiple systems* for another instance where creating buffers is useful.

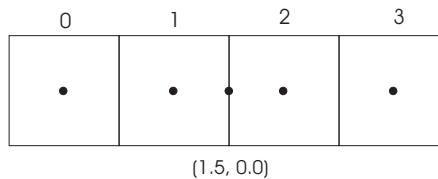
Pixel conventions

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

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

$$\left(\frac{\text{Width} - 1}{2}, \frac{\text{Height} - 1}{2} \right)$$

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

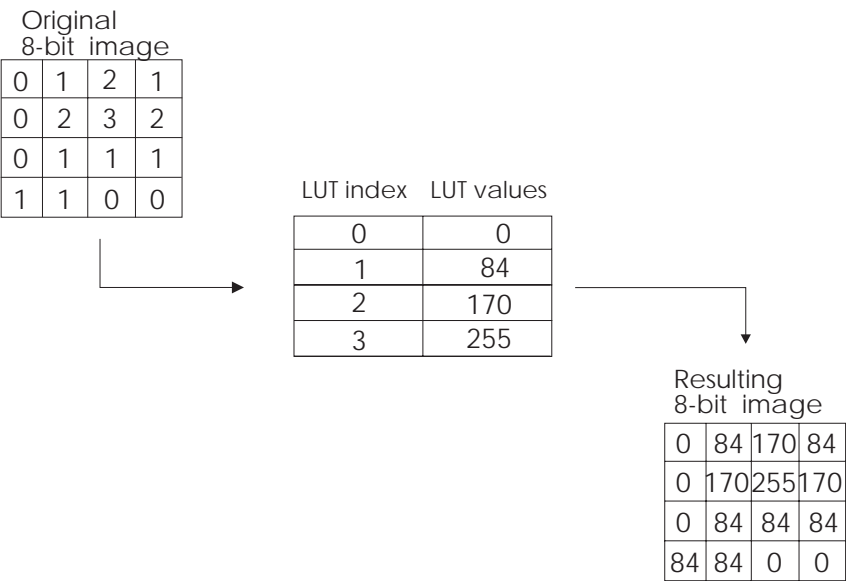


Chapter 4: Lookup tables

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

Direct LUT data generation

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

The *MgenLutRamp()* command generates a value for each LUT index within the specified index range. The index range together with the start and end values determine the increment.

The increment

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

The *MgenLutFunction()* command generates data within the specified LUT buffer area according to a specified function. The functions available are: M_LOG, M_EXP, M_SIN, M_COS, M_TAN, and M_QUAD. The LUT index and the start x value are used as the x value in the equation.

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

Color LUTs

When generating data in a color LUT buffer, each row of each color band is loaded with the same data. For example, for an RGB LUT, the red, green, and blue bands of the LUT are loaded with the same data.

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

Loading LUTs with precalculated data

More complex LUTs

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

- Calculate data, using your Host system, and then load it into the LUT, using *MbufPut()*, *MbufPut1d()*, or *MbufPutColor()*.
- Generate data into another data buffer, using MIL commands other than *MgenLutRamp()*, 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

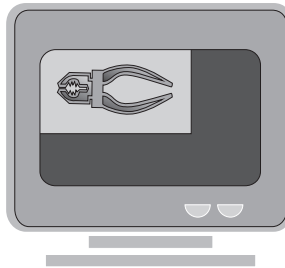
MIL is platform independent. If your system is not using an imaging board with a display section, MIL will use your VGA for display purposes.

Displayable image buffers

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

Selecting a buffer for display

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



If you want to display only one band of a three-band color buffer, you must first allocate a two-dimensional displayable image buffer and copy the required band into it using `MbufCopyColor()`. You can then display this buffer.

Frame buffers

This manual uses the term frame buffer to refer to display memory. The number of available frame buffer surfaces depends on the system you are using. Matrox imaging boards that have a display section typically have two frame buffer surfaces: a dedicated or dynamically allocated main (underlay)

surface and an overlay (VGA) surface. Separate VGA boards typically have only one frame buffer surface, a VGA frame buffer.

Display configuration

MIL supports various display configurations which use combinations of imaging boards with display sections, separate VGAs, and multiple screens. Some of these configurations might not be supported on your system, therefore it is important that you are aware of your system's hardware restrictions when allocating a display in MIL.

Single-screen configuration

The single-screen configuration is a display configuration in which a single board is used both as a VGA to display the user interface (for example, the Windows desktop) and for the display of images. Both the user interface and images are viewed on a single screen. When using an imaging board with a display section in this configuration, the VGA controls the overlay (VGA) frame buffer.

This configuration is supported on systems using an imaging board with a display section and those which use a separate VGA. In other words, this configuration is supported on all systems.

Dual-screen configuration

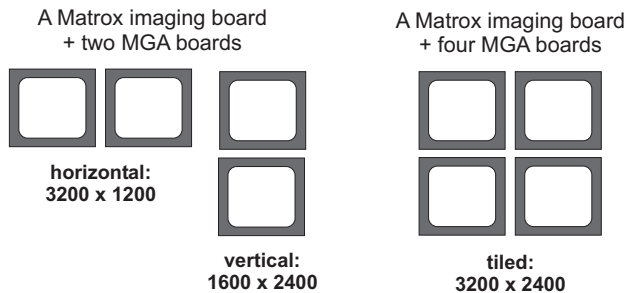
The dual-screen configuration is a display configuration that consists of a separate VGA board for main user-interface display (for example, the Windows desktop) and a Matrox imaging board for image display. This configuration is only supported on systems using an imaging board with a display section.

In this configuration, you can override the default and have images displayed on the same screen as your Windows desktop, so that the display is essentially running in a single-screen configuration. To do so, set the initialization flag for *MdispAlloc()* to *M_WINDOWED*; this operation is discussed later.

- ❖ To configure your Matrox imaging board with a display section in dual-screen mode, see the installation manual of your board to install the board appropriately.

Multi-head display configuration

If you are running Windows NT, you can run in a multi-head display configuration. This configuration is a multi-board configuration that uses a combination of Matrox imaging boards and/or Matrox MGA boards (up to 4 boards). A multi-head display configuration creates one large Windows desktop across multiple screens, in a horizontal, vertical, or tiled fashion.



To run a multi-head display, click on **List All Modes...** in your Windows Display utility (Control Panel) and choose a dual-sized desktop area (for example, 3200x1200) from the list of available resolutions. Note that in a multi-head display configuration, your monitor settings should be compatible with your least-capable monitor.

Display modes and the display window

There are two display modes available, depending on your system's configuration:

- **Windowed mode** (M_WINDOWED).
- **Non-windowed mode** (M_NON_WINDOWED).

You must select one of these modes when allocating a display with *MdispAlloc()*. These modes are described below.

Displaying in windowed-mode

A windowed-mode display (M_WINDOWED) is displayed in its own window. The window is tracked and updated with the image buffer selected on the display; that is, if the window moves or is occluded, the window is updated with the image buffer accordingly.

In windowed mode, multiple windowed-mode displays can be allocated and selected for display; therefore, the display device number should always be set to M_DEFAULT.

This mode is the default allocation mode in a single-screen configuration (M_DEFAULT). If your board has a display section and you are using it in a dual-screen configuration, you can still choose not to use it, and display an image, even a live grabbed image, in windowed mode. In this case, the display is on your Windows desktop.

In windowed-mode, MIL does not communicate directly with the VGA, but uses the normal Windows mechanisms (Windows API functions and extensions) to display images. In other words, it allocates image buffers in a Windows Device Independent BITMAP (DIB or DirectDraw surface) and loads LUT buffers into Windows logical palettes (refer to the Microsoft SDK Programming Guide for information on Windows DIBs, DirectDraw surface, and logical palettes).

Displaying in non-windowed mode

A non-windowed mode (M_NON_WINDOWED) display has no window associated with it. You are responsible for moving and tracking this type of display, if required.

The buffer, selected on the display, is displayed at the top-left corner of the screen. On boards with two dedicated frame buffers, this buffer is actually displayed from the main (underlay) frame buffer surface, and is only visible wherever the overlay (VGA) is set to the keying color (by default, 0).

In this mode, only one MIL display can be allocated and selected for display. This is the default configuration in dual-screen mode.

Display size and depth

In a single-screen configuration, you determine the display format (size and depth) of the overlay frame buffers using the Windows Display utility (Control Panel); in this case, the display format of the MIL display has no effect and should be set to M_DEFAULT. In dual-screen configuration, the display format or video configuration format (VCF) of the selected display determines the display format of the frame buffers.

If the display section has two dedicated frame buffers, a main (underlay) frame buffer and an overlay (VGA) frame buffer, both surfaces are configured to the same size.

In windowed mode, when you select a buffer to a display, Windows will create a display of the same size as the buffer, unless such a display cannot fit in the Windows desktop.

In non-windowed mode, MIL will create a display of the same size as the display format of the frame buffers.

Displaying buffers of different data depths

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

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

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

- The `M_AUTO_SCALE` setting remaps the pixel values to the display such that the minimum and maximum values in the image (not the full range of the buffer) are set to 0 and 255, respectively. If the image buffer contains a single value, its corresponding displayed value is determined by linearly re-mapping the full range of the buffer (for example, 0 to 64K) to 0 through 255.
- The `M_MULTI_BYTES` setting is primarily useful when grabbing from a multi-tap camera. This setting displays each byte of the image in separate display pixels. For instance, each pixel of a 16-bit image will occupy two consecutive display pixels; each pixel of a 32-bit image will occupy four consecutive display pixels.

The default display mode (`M_DEFAULT`) will automatically select the appropriate mode, depending on the image depth.

Removing a buffer from the display

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

You can only remove the entire image buffer from the display. That is, when displaying a parent buffer, you cannot remove one of its child buffers from the display.

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

Displaying multiple buffers

MdispSelect() only allows you to view one buffer at a time in a display. However, in windowed mode, you can use many displays to view more than one buffer at a time. In non-windowed mode, you can view more than one buffer at a time using child buffers. For example, you can display the source and destination buffers of an operation, using the following steps:

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

An example...

The following portion of MIL code shows how to display multiple buffers in a single display. The required portion of the cell image, *cell.mim*, is loaded into a child of a displayable buffer and then 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. Note that these are only display effects; they do not affect the content of the image buffer.

Panning and scrolling

Panning and scrolling displace an image horizontally or vertically, respectively, on the display. You can pan and scroll your image to display the appropriate location at the top-left corner of the window (in windowed mode) or screen (in non-windowed mode), using *MdispPan()*. Note, in non-windowed mode, to display the image at another location on the display, you must create a large displayable image buffer, display it, and then allocate and use a child buffer at the required location on the display.

Zooming

Zooming is the horizontal and/or vertical replication of each pixel by the given integer factor. You can zoom the display by an integer factor using *MdispZoom()*. Note that zooming by a large factor might cause a "blocky" effect. In windowed mode, you can also reduce the size of an image on the display. To do so, pass a negative zoom factor to *MdispZoom()*; this functionality is not supported in non-windowed mode.

In the *mmultdis.c* example, the source and destination image buffer dimensions are rather small, so the parent buffer is zoomed by a factor of 2. This is achieved with the line following the binarizing operation:

```
MdispZoom(MilDisplay, ZOOM_VALUE, ZOOM_VALUE);
```

Annotating the displayed image non-destructively

In windowed mode

In windowed mode, you can annotate the displayed image non-destructively using MIL's overlay-display mechanism.

To make use of this functionality in windowed mode, do the following:

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

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

2. Select a buffer to the display:

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

Since the overlay-display mechanism is enabled, this will not only display the selected image, but it will associate a temporary overlay buffer with the display. This overlay buffer will annotate the underlying image with an effect called keying, which makes portions of the overlay buffer transparent so that underlying areas of the displayed image show through. Therefore, anything that you draw in this buffer will annotate the image selected to the display. Note that when you select another image to the display, another temporary overlay buffer is created.

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

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

This overlay buffer will have the same number of bands as the buffer selected to the display.

4. Draw into the display's overlay buffer with the appropriate graphics function (*Mgra...()*). For example, to write text in the overlay buffer, use *MgraText()*.

In non-windowed mode

To make use of this functionality in non-windowed mode, follow the same steps. However, in this mode, when the overlay-display mechanism is enabled, the display is associated with a temporary overlay buffer immediately. This overlay

buffer is also the same size as the display. When selecting another buffer to the display, the overlay buffer remains the same.

Using the overlay

If available, whenever it is most efficient, both the underlay and overlay frame buffer surfaces are used to annotate the displayed image.

If your board does not have two frame buffer surfaces, a simulated version of the overlay effect is produced. This means that the display update will be slower and a continuous grab operation will appear only in pseudo-live, due to the additional operation needed to combine the grabbed image with the overlay buffer. However combined, the actual buffer selected on the display is not overwritten by the content of the overlay buffer.

Keying

When allocating a display (*MdispAlloc()*), keying is automatically enabled, if required, and the keying color is automatically set to a default color (generally appropriate).

If required, select another keying color with *MdispOverlayKey()*. If you are using an 8-bit display resolution (256 colors), you can set the color to a value between 0 and 255. If you are using a non 8-bit display resolution (15-bit, 16-bit, 24-bit, or 32-bit), call the macro *M_RGB888* and specify the RGB value, for example, as follows:

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

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

The following portion of MIL code shows the enabling of the overlay on the display, the inquiring of the overlay buffer identifier, and the display of text in the overlay (see also, *mdispovr.c*). It assumes that an image has been selected to the display.

```

/* Enable overlay effects on top of the display buffer */
MdispControl(MilDisplay, M_WINDOW_OVR_WRITE, M_ENABLE);

/* Inquire the identifier of the overlay buffer associated with the
 * displayed buffer.
 */
MdispInquire(MilDisplay, M_OVR_BUF_ID, &MilOverlayImage);

/* Print a string in the overlay buffer to appear over the displayed
 * image buffer.
 */
MgraText(M_DEFAULT, MilOverlayImage, 0, 0, " · MIL Overlay Text · ");

```

*Forcing the display in
the overlay*

Although not commonly done, if using an imaging board that has a display section, you can allocate a MIL display that is only in the overlay (VGA) frame buffer surface, if the display is allocated with the M_OVR attribute. When in non-windowed mode and using an imaging board with a display section, your buffer must be allocated with an M_IMAGE+M_OVR+... attribute before it can be selected to an M_OVR display.

Using GDI annotations

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

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

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

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

When either buffer is changed, signal MIL by calling *MbufControl*(..., M_MODIFIED,...).

This method avoids a flickering display when drawing.

- Inquire the display's window handle using *MdispInquire()* with M_WINDOW_HANDLE. Pass the window handle to the Windows *GetDC()* function to get a Windows display device context (DC). Then, paint the annotations with GDI functions from a function hooked to the display update event (*MdispHookFunction()*), that is, paint each time the MIL display is modified.

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

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

```

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

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

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

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

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

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

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

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

```

Displaying an image in a user-defined window

Selecting a buffer into a specific display window

Under Windows, you can display a specific image buffer in a user-defined window, using *MdispSelectWindow()*. For best results, the display must have the same resolution as the image buffer depth. The window must be created with the Windows API functions. If the defined window is of different dimension than the image buffer, any excess window area will be left untouched or any excess image area will be cropped.

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

Using the *MdispSelectWindow()* function

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

An example...

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

```

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

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

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

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

```

(cont...)

```

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

    long BufSizeX;
    long BufSizeY;
    long BufSizeBand;

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

    /* Allocate a MIL system. */
    MsysAlloc(M_DEF_SYSTEM_TYPE, M_DEV0, M_DEFAULT, &MilSystem);

    /* Allocate a MIL display. */
    MdispAlloc(MilSystem, M_DEV0, M_DEF_DISPLAY_FORMAT, M_DEFAULT
    &MilDisplay);

    /* Allocate a MIL digitizer if supported and sets the target image size.*/
    if (MsysInquire(MilSystem, M_DIGITIZER_NUM, M_NULL) > 0)
    {
        MdigAlloc(MilSystem, M_DEV0, M_DEF_DIGITIZER_FORMAT, M_DEFAULT,
        &MilDigitizer);
        MdigInquire(MilDigitizer, M_SIZE_X, &BufSizeX);
        MdigInquire(MilDigitizer, M_SIZE_Y, &BufSizeY);
        MdigInquire(MilDigitizer, M_SIZE_BAND, &BufSizeBand);
    }
    else
    {
        MilDigitizer = M_NULL;
        BufSizeX = BUFFERSIZE_X;
        BufSizeY = BUFFERSIZE_Y;
        BufSizeBand = BUFFERSIZE_BAND;
    }

    /* Do not allow example to run in dual-screen mode */
    if (MdispInquire(MilDisplay, M_DISPLAY_MODE, M_NULL) != M_WINDOWED)
    {
        MessageBox(0, ""This example does not run in dual-screen mode.",
        "MIL application example",
        MB_APPLMODAL | MB_ICONEXCLAMATION );
        goto end;
    }
}

```

(cont...)

```

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

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

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

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

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

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

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

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

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

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

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

end:

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

```

LUTs and changing the displayed colors or gray levels

In general, when displaying in a 256-color display resolution, images are mapped through physical output LUTs on display. These LUTs are available and programmable so that you can achieve the best display effect for your images since not all colors are available in this resolution. Note that when displaying in a non-256-color display resolution, MIL can simulate a display LUT in software for 8-bit images.

How images are mapped through the physical output LUTs

By default in windowed mode, when displaying 8-bit images, MIL tries to use the image's pixel values to address the physical output LUTs. When displaying color images, MIL will search the physical output LUTs for the entry that best matches the color of the image's pixels being displayed. It then creates a translation table for the image. This table is used to convert each pixel value upon display to a value that will address the appropriate physical output LUT entry.

In non-windowed mode, MIL uses the image's pixel values to address the physical output LUTs.

Palette versus physical output LUTs

The actual programming of the physical output LUTs is handled by MIL in one of two ways. In windowed mode, MIL indirectly programs the physical output LUTs through the use of a Windows palette. In non-windowed mode, MIL programs the physical output LUTs directly.

Default palette settings in windowed mode

By default in windowed mode, MIL provides a good default logical palette for the realization of the physical output LUTs (*MdispLut(..., M_DEFAULT, ...)*). MIL takes into consideration the displayed image, the Windows display driver used, and the VGA physical output LUT capabilities, and produces the best "portability versus visual quality" compromise possible.

Default physical output LUT settings in non-windowed mode

By default in non-windowed mode, MIL generates a ramp in the physical output LUTs, which uses the full range of available intensities (*MdispLut(..., M_DEFAULT, ...)*). This type of mapping is also referred to as a pass-through LUT mapping (or transparent LUT mapping).

Changing the default LUT values

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

- When displaying monochrome images, you might want to view the images with each gray intensity in a different color. For example, you can associate specific colors to ranges of temperatures obtained by an infra-red camera.
- When displaying monochrome images, you might want to invert the image values. For example, when grabbing a film negative, you can display the film as it will be printed.
- In windowed mode, when displaying color images under a 256-color display driver resolution, you might want to reduce the loss of color resolution. For example, when displaying a color image with many shades of red, you might want to select a LUT so that all shades of the image are represented.

To change the LUT values, associate a pseudo-color or custom LUT buffer to the display with *MdispLut()* or to the displayed image buffer with *MbufControl()*. Please note that LUT buffers used for display have the following restrictions:

- If the LUT buffer values are changed while the image is selected on the display, the changes will not take effect.
- The LUT buffer will not be used when displaying a 3-band 8-bit image under a non-8-bit display resolution.
- A LUT buffer cannot be associated to a display that belongs to a system using an imaging board with an on-board display section, unless that display has been allocated with the *M_OVR* initialization parameter.
- The LUT buffer must have one or three bands. Note that the number of LUT buffer entries must be the same as the maximum number of intensities that can be represented in the displayed buffer. In other words, if you want to invert an 8-bit grayscale image (that is, an image that can have 256 intensities), your LUT must also have 256 entries.

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

Associating a pseudo-color LUT

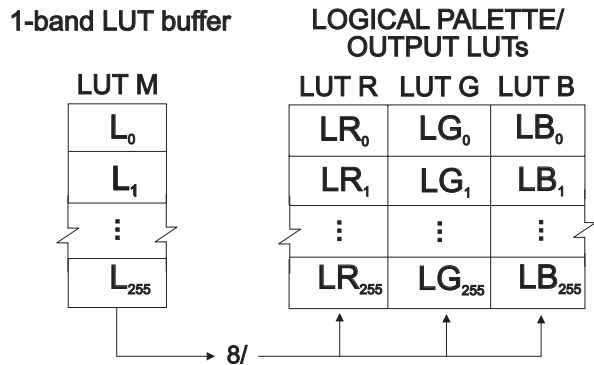
To view an 8-bit image buffer with each gray intensity in a different color, associate the default pseudo-color LUT buffer (M_PSEUDO) with the display of the image. In windowed mode, the data is loaded in each component of the logical palette. In non-windowed mode, the data is loaded into the physical output LUTs of the display.

A 1-band custom LUT buffer

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

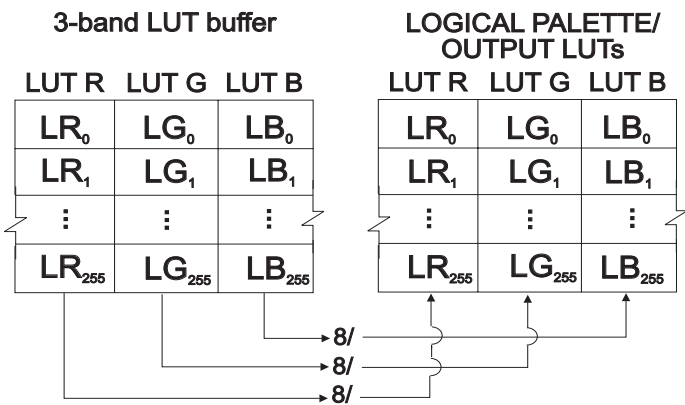
1. Allocate a one-band LUT buffer (*MbufAlloc1d()*).
2. Generate the data into the buffer, using *MgenLutRamp()* or load the data into it, using *MbufPut()*. The depth of the LUT buffer data must be 8-bits.
3. Associate the LUT buffer with the required display using *MdispLut()*, or to a particular image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

If you associate a one-band LUT buffer with the display or buffer in windowed mode, the same data is loaded into each component of the logical palette. In non-windowed mode, the same data is loaded in each available display physical output LUT.



A 3-band custom LUT buffer

To reduce the loss of color resolution when displaying an image with a specific range of colors, you would need physical output LUTs that contain all the required colors so that when Windows creates a translation table for the image, most colors are mapped to their exact values. Follow the steps for a 1-band LUT buffer, except allocate and load a 3-band LUT buffer instead. When in windowed mode, each band of the LUT buffer is loaded into its corresponding component of the logical palette. In non-windowed mode, each band is loaded in a different display physical output LUT (if a different LUT is available for each display channel).



Different display architectures in windowed mode

Although all MIL display windows have a similar appearance, MIL uses one of three different architectures to make an image buffer visible through a display window. The particular architecture determines the behavior of the display window under specific circumstances. For example, the display architecture will determine how a continuous grab operation will behave when overlapped by another window. Similarly, the display architecture will determine whether or not the Host CPU is used to overlay graphical annotations or images on top of the buffer selected for display.

The three display architectures are listed and discussed below.

- Underlay display
- Overlay/ regular display
- DirectDraw underlay-surface display
- ❖ The type of display architecture that can be used depends on the available hardware. Keep in mind that MIL automatically selects the most appropriate display architecture when a display is allocated (*MdispAlloc()*) with the default (M_DEFAULT) initialization flag.

Underlay display architecture

Under the underlay display architecture, the Windows desktop sits in a dedicated overlay frame buffer surface, whereas the selected MIL buffer(s) sits in a dedicated underlay frame buffer surface. The data of the buffer, selected to a windowed display, is visible through a special hardware keying mechanism.

An underlay display architecture is used only when the video frame buffer is physically split into a main (underlay) and overlay frame buffer. The display resolution sets the size of the overlay and the underlay frame buffer surfaces that are used for on-screen purposes. Accordingly, the amount of video memory required is twice that of the current display resolution. This display architecture is used only when it is more efficient to do so.

The underlay display architecture is used only on Matrox Imaging frame grabbers such as Matrox Corona, Matrox Genesis, and Matrox Pulsar frame grabbers.

Under this display architecture, there are a number of important features:

- *MdigGrabContinuous()* is always performed live with no Host CPU intervention, irrespective of overlapping of the display windows.
- Graphics and video overlay on top of the selected buffer is done with no Host CPU intervention.
- *MdispLut()* is usually not supported.

- *MdispZoom()* is not accelerated by the hardware, which means that it is emulated by the software.
- The underlay surface data-format usually follows the Windows display resolution. This means that the underlay surface data-format is typically in RGB format.

MIL will choose the most appropriate display architecture at the time of display allocation (*MdispAlloc()*). Still, you can use the `M_WINDOWED + M_UND` initialization flag to force this dedicated underlay display architecture, provided that you have the appropriate hardware.

Overlay/regular display architecture

Under a overlay/regular display architecture, image buffers are allocated in a Windows Device Independent Bitmap (DIB or Direct Draw surface) and then Windows handles their display. In addition, LUT buffers are loaded into logical Windows palettes.

An overlay/regular display architecture uses the regular Windows mechanisms (Windows API function and extensions) to display images. In this case, the video frame buffer does not have to meet any special conditions.

Under this display architecture, there are a number of important features:

- *MdigGrabContinuous()* is performed live with no Host CPU intervention when (1) the display format is supported by the frame grabber, (2) the display window is not overlapped by another window, and (3) when there is no overlay. Otherwise, when these conditions are not met, MIL automatically switches to a pseudo-live grab, which uses Host CPU to emulate the grab operation to the display.
- Graphics and video overlay on top of the selected buffer is emulated by the software. Consequently, the graphics overlay makes use of the Host CPU.
- *MdispLut()* is supported. Therefore, it is possible to perform a continuous grab operation in pseudo-color.
- *MdispZoom()* is not accelerated by the hardware, which means that it is emulated by the software.

Although MIL will choose the most appropriate display architecture, at the time of display allocation (*MdispAlloc()*), you can use the `M_WINDOWED + M_OVR` initialization flag to force this overlay display architecture.

DirectDraw underlay-surface display architecture

The DirectDraw underlay display architecture is similar to the underlay display architecture described above. The display resolution sets the size of the overlay frame buffer surface. However, under the DirectDraw underlay-surface display architecture, the VGA display adapter dynamically allocates an underlay surface that is of the same size as the buffer selected to the display. Accordingly, the amount of video memory used for on-screen purposes depends on the size and depth of the image buffer selected for display.

A DirectDraw underlay-surface display architecture can only be used when the VGA display adapter can dynamically allocate an underlay surface, and is only used when it is more efficient to do so.

The DirectDraw underlay-surface display architecture is currently available when using a Matrox frame grabber with a Matrox G200 or Matrox G400 graphics controller such as the Matrox Orion frame grabber. The Cyrix processor's companion chip on the Matrox 4Sight imaging platform also supports the DirectDraw underlay-surface display architecture.

Under this DirectDraw underlay surface display architecture, there are a number of important features:

- *MdigGrabContinuous()* is always performed live with no Host CPU intervention, irrespective of overlapping of the display windows.
- Graphics and video overlay on top of the selected buffer is done with no Host CPU intervention.
- *MdispLut()* is not supported.
- *MdispZoom()* is accelerated by the hardware, which means that there is no Host CPU intervention.

- The underlay surface data format is usually YUV. Typically, this is an advantage because it allows true color images to be displayed even in 256 colors display resolution (if there are no hardware restrictions which apply).

MIL will choose the most appropriate display architecture at the time of display allocation (*MdispAlloc()*). Still, you can use the `M_WINDOWED + M_DDRAW_UND` initialization flag to force this DirectDraw underlay-surface display architecture, provided that you have the appropriate hardware.

Advanced controls for windowed mode

Display types in windowed mode

In windowed mode, when allocating a display (*MdispAlloc()*), you can specify how image buffers are displayed in a 256-color display resolution. There are three types of display initialization:

- **Enhanced** (`M_DISPLAY_ENHANCED`, `M_DISPLAY_8_ENHANCED`, `M_DISPLAY_24_ENHANCED`)
- **Basic with optimization** (`M_DISPLAY_BASIC`, `M_DISPLAY_8_BASIC`, `M_DISPLAY_24_BASIC`)
- **Basic without optimization** (`M_DISPLAY_WINDOWS`, `M_DISPLAY_24_WINDOWS`)

Select both an `M_DISPLAY_8_XXX` and `M_DISPLAY_24_XXX` display initialization to independently control the display of 8-bit and 3-band 8-bit images.

Enhanced

When using an enhanced initialization, the MIL display calls the Microsoft Video for Windows **DrawDIBDraw()** function to display image buffers. This function's use of dithering particularly improves the display of 3-band 8-bit images under 256-color display resolution.

Note, with enhanced initializations, the actual display color values are selected, on a best-match basis, from the logical palette's available display colors. Therefore, effects such as those of an inverse LUT are not possible. This is the default display initialization for an 8-bit 3-band image.

Basic with optimization

When using a basic with optimization initialization, the MIL display calls the Windows API **StretchDIBits()**, **StretchBlt()**, or **DirectDrawBlt()** function to display image buffers. When 8-bit images are displayed, the pixel values are used, as much as possible, to index the physical output LUTs. When 3-band 8-bit images are displayed in an 256-color display resolution, the display uses an algorithm optimized for speed. This algorithm converts 24 bits to 8 bits by taking the most-significant bits of each component: 3 bits each are taken from the red and green components, and 2 bits from the blue. This produces an 8-bit DIB with 3:3:2 RGB values for display; it is these values that are used to address the physical output LUTs. This is the best possible combination when you are not aware of the color content of the image buffer.

Basic without optimization

When using a basic without optimization initialization, the MIL display calls the Windows API **StretchDIBits()**, **StretchBlt()** or **DirectDrawBlt()** function to display image buffers; however no optimization for speed is done when displaying a 3-band 8-bit image in a 256-color display resolution. The display will display such images (color images) on a best-match basis and display 8-bit images using their pixel values to address the physical output LUTs.

This display initialization can result in slow display performance.

Zoom types in windowed mode

In windowed mode, when allocating a display (*MdispAlloc()*), you can specify how image buffers are zoomed in a 256-color display resolution. There are two types of zoom initialization:

- **Enhanced** (M_ZOOM_ENHANCED)
- **Basic** (M_ZOOM_BASIC)

Enhanced

When using an enhanced zoom initialization, the **DrawDIBDraw()** function is called to perform a zoom. Although zooming might be a little slower than using the basic initialization option, it does not alter the dithering quality,

providing a better quality zoom. This option is the default and is only available when an `M_DISPLAY_XXX_ENHANCED` display initialization is used.

Basic

When using a basic zoom initialization, Windows (Windows API functions) is called to perform the zoom. Note, if an `M_DISPLAY_XXX_ENHANCED` display initialization is used, this zoom might alter the quality of the **DrawDIBDraw()** dithering.

Controlling how the LUT buffer is loaded into the Windows palette

When calling *MdispLut()*, MIL will copy the data of each band of the LUT buffer to the corresponding component of the logical palette, without modification. To obtain good results, the specified color values must be carefully selected to provide the best color match upon image display. If the specified values closely match the RGB values that occur frequently in the image to be displayed, very good results can be obtained.

Controlling how the logical palette is loaded into the physical output LUTs

When in windowed mode, you can control how Windows loads the logical palette into the physical output LUTs. If you want the Windows palette manager to use palette optimization when realizing the physical output LUT values from the logical palette, use *MdispControl()* with the `M_WINDOW_PALETTE_NOCOLLAPSE` control type.

The `M_WINDOW_PALETTE_NOCOLLAPSE` control type can be used to control palette optimization in one of two ways:

- The Windows palette manager realizes the physical output LUTs with the best color usage of the logical palette (`M_DISABLE`); this is the default setting.
- The Windows palette manager realizes the physical output LUTs by loading the logical palette "as is" (`M_ENABLE`).

Optimizing the physical output LUTs

When setting the `M_WINDOW_PALETTE_NOCOLLAPSE` control type to `M_DISABLE`, the Windows palette manager attempts the best color usage of the logical palette when realizing the

physical output LUTs. The palette manager tries to map colors from the logical palette into the currently realized physical output LUTs to reduce the number of requested new entries. This reduces the chance of a color occurring more than once in the physical output LUTs.

*Realizing the physical
output LUTs without
modification*

When setting the `M_WINDOW_PALETTE_NOCOLLAPSE` control type to `M_ENABLE`, the Windows palette manager loads each component of the logical palette "as is" into the corresponding physical output LUT. This can result in a color occurring more than once in the physical output LUT.

Chapter 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.

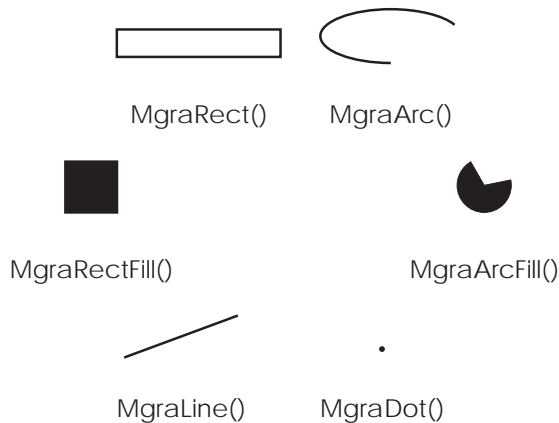
Drawing graphics

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.

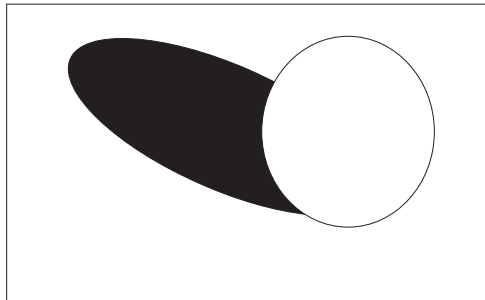
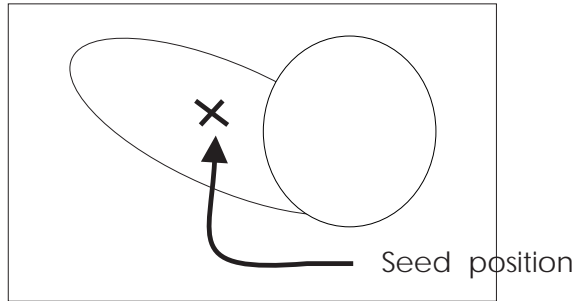
In addition, the MIL package includes *MgraRectFill()* and *MgraArcFill()* so you can draw solid rectangles and arcs.



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 and the control of your digitizers, including the fine-tuning of the input and auto-focusing.

Cameras and input devices

The MIL package supports input from any type of input device supported by the digitizer. Data grabbed from an input device with the digitizer using *MdigGrab()* or *MdigGrabContinuous()*, is stored into an image buffer. For color cameras, you must use color image buffers, with the same number of bands as the incoming data. Note, since most input devices are cameras, they will hereafter be referred to as such.

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

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

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

The data format

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

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

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

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

If you cannot find an appropriate DCF file or have a non-standard input device that does not appear in our list (such as a strobe or trigger device), you can create your own, also using Matrox Intellicam. For more information on Matrox Intellicam, refer to the *Matrox Intellicam User Guide* manual.

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

The digitizer number

The device number

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

Multiple cameras

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

The input channel

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

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

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

Number of frames or fields

To grab a single frame or field, use *MdigGrab()*. The type of scanning used by your camera determines whether you grab fields or frames. With progressive scanning cameras, frames are grabbed. If your camera uses interlaced scanning, fields are grabbed. By default, if your camera uses interlaced scanning, one call to *MdigGrab()* will grab both the odd and even fields.

To grab a series of continuous frames, use *MdigGrabContinuous()*; this function uses the specified digitizer to continuously acquire frames of data until *MdigHalt()* is called.

Note, when grabbing data with *MdigGrab()*, you can specify how many fields or frames to grab using *MdigControl()*, with `M_GRAB_FIELD_NUM` or `M_GRAB_FRAME_NUM`.

Line-scan cameras

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

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

Grabbing to the display

Live and pseudo-live continuous grabs

With MIL, you can grab to a displayable buffer selected on a display. If your system is not using an imaging board with a display section, MIL will use your VGA for display purposes. MIL uses one of two methods to transfer when grabbing:

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

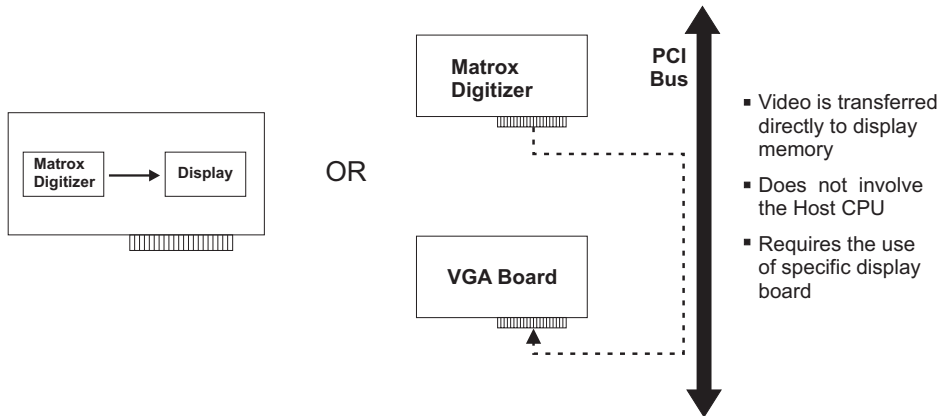
When grabbing, the digitizer (for example, Matrox Meteor-II) always acts as the bus master.

A monoshot grab is always pseudo-live. Grabbing a specific number of frames is also performed pseudo-live (note that under a non-windowed display it is possible to perform a live monoshot grab by allocating your buffer directly on the VGA board with `M_ON_BOARD`).

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

Live transfer to the display

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



Pseudo-live transfers to the display

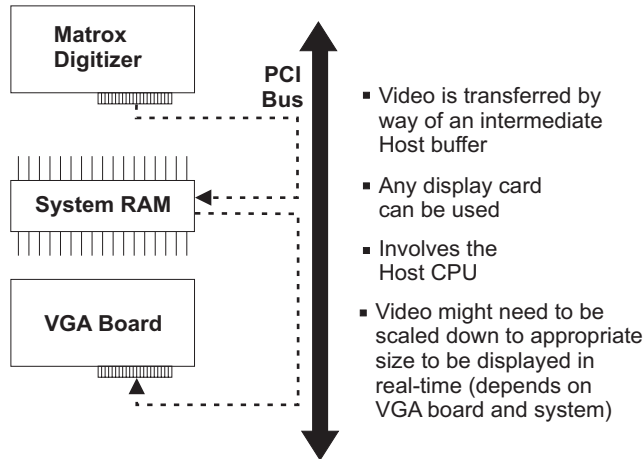
A continuous grab will automatically switch to pseudo-live if:

- Your VGA board does not support fast linear-memory accesses (discussed later in this section).
- The format of the grabbed data is not compatible with your VGA display mode. For example, performing a color grab in 256 color display resolution.
- Your board does not have both an underlay and overlay frame buffer surface and there is a non-rectangular overlap between the display windows on the display device.
- Your board does not have both an underlay and overlay frame buffer surface, DDraw is disabled or you are in multi-head mode, and the grab display window does not have the focus, that is, is not active.
- Your board does not have both an underlay and an overlay frame buffer surface and you are using the display's overlay buffer, that is, have enabled `M_WINDOW_OVR_WRITE` with

MdispControl(). In this case, the grab will be pseudo-live because an additional operation is required to combine a grabbed image with a simulated version of the overlay.

- You are in multi-head mode and the display window occupies more than one screen.

MIL transparently performs pseudo-live grabs:



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

Pseudo-live transfers will be real time (that is, full frame rate of 30 for NTSC or 25 fps for PAL) if the CPU transfer from the Host buffer to display memory is fast enough. That is, if the blit is taking at most one frame time length. Blit time is affected by the load of the CPU (for example, the number of process threads and the priorities of other boards). You can reduce the load of the CPU in the pseudo-live grab operation by disabling the double buffering operation. However, when double buffering is disabled, only half of the full frame rate can be achieved.

Multi-head mode

In multi-head mode, note that a continuous grab without overlay can be moved from one screen to another and be displayed live when it has the focus. However, a continuous grab with overlay will only be live on the screen attached to the on-board display section; it will switch to pseudo-live on the other screen(s). In both cases, when the window displaying the grab intersects two screens, the grab is pseudo-live.

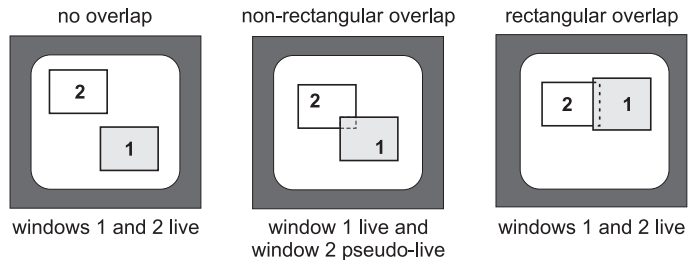
The table below indicates the type of configurations which are supported on particular boards.

Display configuration	Corona	Genesis	Meteor-II	Orion	Pulsar
single-screen or multi-head (windowed mode)	x	x	x	x	x
single- or dual-screen (non-windowed mode)	x	x			x

For more information about your board's transfer capabilities, consult the *MIL/MIL-Lite Board-specific notes* manual.

Window occlusion

When there is no overlap or rectangular overlap of a live grab window, the continuous grab is displayed live. When there is non-rectangular overlap (that is, the displayed portion of the occluded window is no longer rectangular), there is pseudo-live display.



* Window 1 is the active window and window 2 is the grab and display window.

Note that when DDraw is disabled (see *MsysAlloc()*), the continuous grab is displayed live only when the window has the focus (that is, is active).

Using an MGA VGA board

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

Using a VGA board other than MGA board

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

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

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

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

```
VgaPhysicalAddress=EF000000
```

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

```
VgaPitch=400
VgaFormat=M_BGR15+M_PACKED
```

❖ All values are hexadecimal.

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

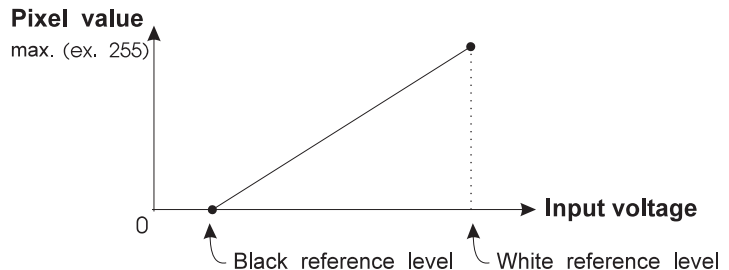
Reference levels, lookup tables, and scaling

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

Black and white reference levels

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

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



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

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

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

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

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

Color image reference levels

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

Mapping grabbed data through a LUT

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

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

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

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

Scaling

The *MdigControl()* function allows you to scale grabbed data horizontally and vertically. If you scale grabbed data, the stored image size is different from the original image by the specified factors in the X and/or Y direction. The scaled image is written in contiguous locations in the image buffer, starting from the top-left corner. For example, if you set both the X and Y scaling factors to 1/2, only one column and one row out of two are written to the image buffer.

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



0	0	0	0	0
0	115	0	0	0
0	243	196	0	0
0	0	115	87	87
0	0	0	92	0

Subsampled image

X subsampling factor = 2

Y subsampling factor = 2

Original image

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

To disable scaling, set scaling factors to 1.

Optimizing application performance when grabbing

Grab mode

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

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

Double buffering

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

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

Below is an example of how to perform double buffering:

```

/* ***** */
/* This example does double buffered grab with real time processing. */
/* Note: This assume that the processing operation is shorter than a grab */
/* and that the PC has sufficient bandwidth to support the 2 */
/* operations simultaneously. Also if the target processing buffer */
/* is not on the display, the processing speed is augmented. */
.
.
.
/* Image scale. */
#define IMAGE_SCALE 0.5

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

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

    long      NbProc = 0;

```

(cont...)

```

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

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

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

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

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

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

```

(cont...)

```

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

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

```

Multiple buffering

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

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

Grabbing a sequence of frames in real-time

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

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

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

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

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

Grabbing with triggers and exposures

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

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

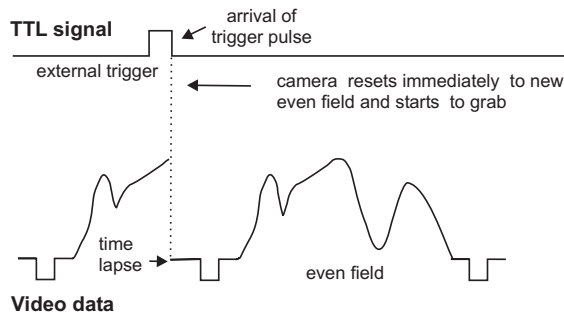
exposure signal should be generated (for the camera) upon the grab trigger event, the actual grab would only be triggered once the active exposure time was over.

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

Asynchronous reset mode

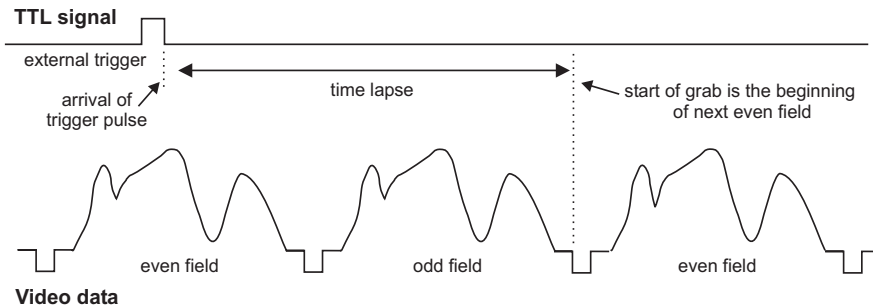
If your digitizer supports asynchronous reset mode, the digitizer resets the camera to begin a new frame when the trigger signal is received.

Asynchronous reset mode



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

Next valid frame (or field) mode



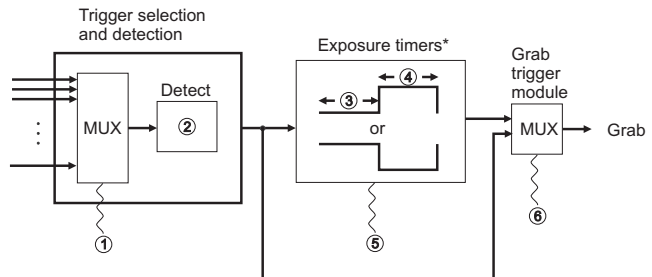
Triggers and exposures

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

Automatic exposure model

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

(M_GRAB_EXPOSURE_BYPASS set to M_DISABLE or M_DEFAULT)



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

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

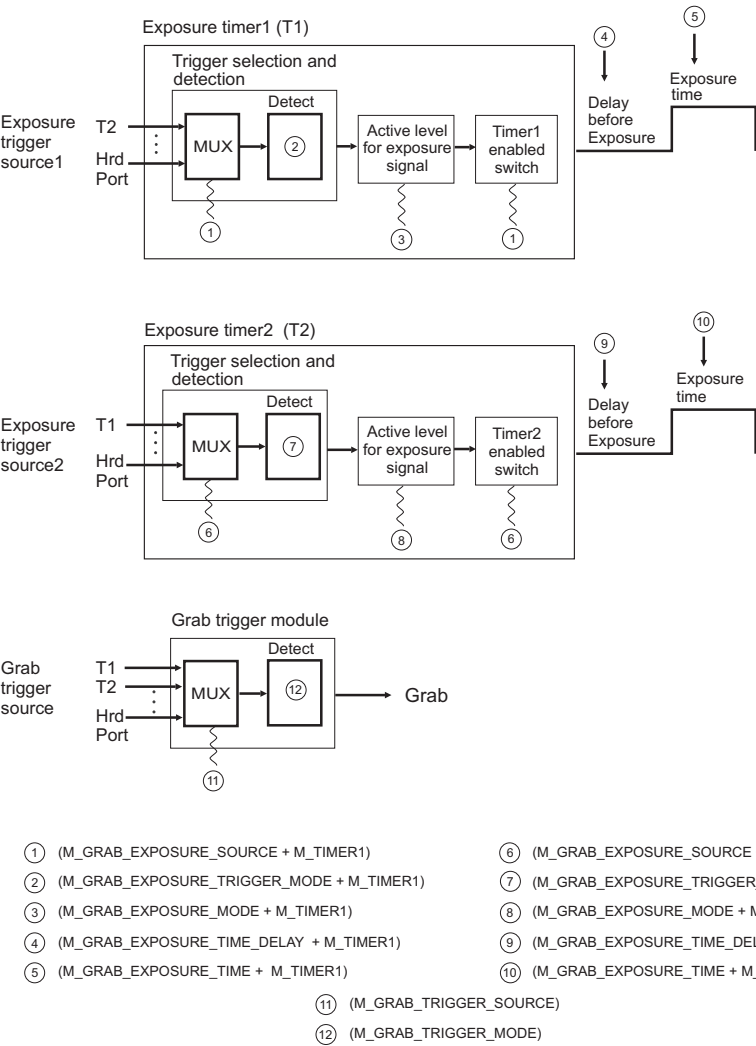
To summarize:

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

Manual exposure
bypass model

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

Manual exposure bypass model
(M_GRAB_EXPOSURE_BYPASS set to M_ENABLE)



Software triggers

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

In the automatic exposure model

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

In the manual bypass model

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

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

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 must be allocated on the same system and have compatible dimensions.

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

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

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

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

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

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 allows you to compress and decompress images. Compression allows you to store more images in memory than would normally be possible. In addition, it allows images to be transferred more quickly, since it reduces the amount of data that must be transferred. MIL can compress images using the JPEG lossless algorithm or the JPEG lossy algorithm.

❖ Under MIL-Lite, dedicated hardware is required to compress and decompress images.

JPEG lossless

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

JPEG lossy

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

Interlaced JPEG

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

Control options

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

**.avi files*

You can use *MbufExportSequence()* to export a sequence of image buffers to an audio video interleave (*.avi) file. You can use *MbufImportSequence()* to import a sequence of images from an *.avi file into separate buffers.

General steps

Compression

To compress an image:

1. Allocate a buffer in which to hold the compressed image. Use *MbufAlloc...()*, allocating the buffer with an `M_COMPRESS+CompressionType` attribute.
2. If necessary, change the control settings of the buffer, using *MbufControl()*. Specifically, for a lossy compression, you might want to change the quantization factor, which is one of the factors that determine the amount of compression.
3. If the image to compress is stored in a buffer, use *MbufCopy()* to compress it into the buffer allocated in step 1. If it is stored on file, use *MbufImport()*. Note that, if you want the compressed image stored on file rather than in a buffer, use *MbufExport()* instead of *MbufCopy()*. In this case, there is no need to allocate a destination buffer.

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

Decompression

To decompress an image, use *MbufCopy()*, *MbufImport()*, or *MbufExport()*, depending on where the source image is stored (in a buffer or on file) and where you want results written (to a buffer or file). Before the decompression, you should not change any control settings in the source image. This is because, in order for the reconstructed image to match the original, the same controls must be used to decompress. If you do change a control setting, the image data will be lost.

Multi-band buffers and color formats

When you allocate a multi-band buffer for a lossy compression, you can specify that the compressed image be stored in an RGB or YUV format. Note that most image-viewing software display compressed color images in YUV 4:2:2 format. When the chosen format differs from that of the source image, MIL internally converts the source image to the specified format, then performs the compression.

*Multi-band buffers
and control settings*

If you are compressing a multi-band buffer, you can specify different control settings for each band. To do so, create a child buffer from each band, using *MbufChildColor()*, then set controls for each child buffer, using *MbufControl()*.

Alternatively, if you are performing a lossy compression on a YUV image, you can use the `xx_LUMINANCE` and `xx_CHROMINANCE` control types. The `xx_LUMINANCE` control type affects the Y band, while `xx_CHROMINANCE` affects the U and V bands.

*Application-specific
markers*

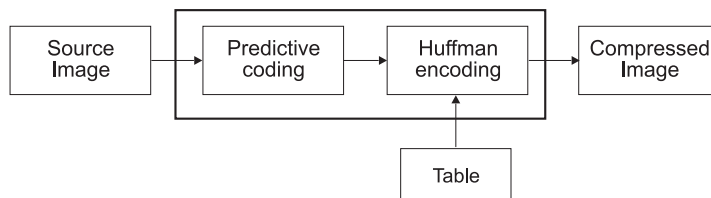
During a compression, MIL adds some application-specific markers to the resulting image. Most other packages will ignore these markers and therefore be able to decompress the file. MIL itself ignores unrecognized markers when it decompresses files.

Controlling a JPEG compression

This section provides a brief overview of the JPEG lossless and lossy algorithms and of the controls you have over these algorithms. In general, you should only change these controls if you are familiar with the algorithm you are using. For detailed information about the JPEG lossless and lossy algorithms, see the *JPEG Technical Specification Revision 8*.

JPEG lossless

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



Predictive coding

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

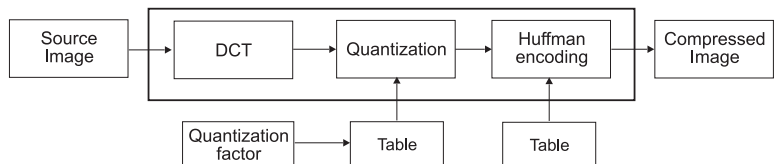
By default, MIL uses the pixel to the left to predict values. This is suitable for most images. However, you can specify that no predicting be done, using *MbufControl()*. In this case, the values after predictive coding will be the same as the original values. This can be useful if you have developed your own algorithm to take the place of predictive coding and only need your images Huffman encoded. Note that you must implement your own algorithm to use one of the other "predictors" supported by the JPEG lossless algorithm; MIL only supports predictor #0 (no predictor) and predictor #1 (the "pixel-to-the-left" predictor).

Huffman encoding

After an image has been predictive coded, Huffman encoding assigns a variable-length "code word" to each value. This code is based on the number of bits by which the difference between adjacent values differ. By storing the code word, rather than the actual difference value, further compression can be achieved. Values are assigned code words according to a DC Huffman table. You can use the default DC Huffman table or you can create your own table. If you want to use your own table, refer to the *Using your own table* section.

JPEG lossy

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



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

Using your own table

For a JPEG lossless compression, you can use your own DC Huffman table. For a JPEG lossy compression, you can use your own quantization, DC Huffman, or AC Huffman table. In order to use your own table:

1. Allocate a buffer with an `M_ARRAY` attribute and of the required size. Huffman tables are one-dimensional, so use *MbufAlloc1d()* to allocate the buffer. Quantization tables are two-dimensional, so use *MbufAlloc2d()*.
2. Transfer the table values to the buffer, using *MbufPut1d()* or *MbufPut2d()*, depending on the type of table.
3. Associate the `M_ARRAY` buffer to the required image buffer, using *MbufControl()*.

Note that you can associate a different table to each band of a multi-band buffer. To do so, create a child buffer from each band, using *MbufChildColor()*, then associate a table to each child buffer. Alternatively, for lossy compressions of YUV images, use the `xx_LUMINANCE` and `xx_CHROMINANCE` control types.

Restart markers

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

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

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

Chapter 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 force the thread to inherit the state of a specific existing MIL application by creating a child MIL application, using *MappChild()*. Although inheriting (upon allocation) the state of the parent application, the child application is subsequently considered a separate application and can be modified independently of the state of its parent.

You can have the thread's application initialized with a reset initial state by allocating a new application, using *MappAlloc()*.

Synchronization

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

This means that Host threads continue execution without waiting for the execution of the on-board commands to complete. In most cases, this is desirable to make the Host thread available for other tasks. However, for operations that necessitate the completion of a previous command(s) in order to return valid results (for example, *MbufGet()* after an *MdigGrab()*), MIL automatically synchronizes the threads to force the Host to wait for completion of the earlier command(s).

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

Thread control

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

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

Error reporting

Some functions in MIL are asynchronous, that is, they queue their command to the hardware and then immediately return control to the Host. For this reason, errors are only reported when detected and not necessarily before the end of the MIL function.

The most common way to check for errors is to use the *MappGetError()* function. This function returns the errors currently detected in a thread.

An example of using multiple threads or systems

Multiple threads

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

```

/* File name: mthread.c
 * Synopsis: This program shows how to use different threads and synchronize
 *           them with MIL. It creates 4 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_LENGTH_MAX   40
#define STRING_POS_X        10
#define STRING_POS_Y        220
#define STRING_TOP           "0"
#define STRING_BOTTOM        "0"

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

```

(cont...)

```

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);

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

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

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

```

(cont...)


```

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

/*Start rotate and edge detect threads. */

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

/* Send events to trigger operation of top left and top right threads. */
MappControlThread(EventWaitTopLeft, M_EVENT_SET, M_SIGNALED, M_NULL);
MappControlThread(EventWaitTopRight, M_EVENT_SET, M_SIGNALED, M_NULL);
/* Report what has happened to the Host screen. */
printf("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_LENHT_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) += 1;
        ltoa(*((THREAD_PARAM *) TParam)->NumberOfIterPtr, Text, 10);
        MgraColor(M_DEFAULT, 0xff);
        MgraText(M_DEFAULT, SrcImageId, STRING_POS_X, STRING_POS_Y, Text);
        MgraRectFill(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 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;
}

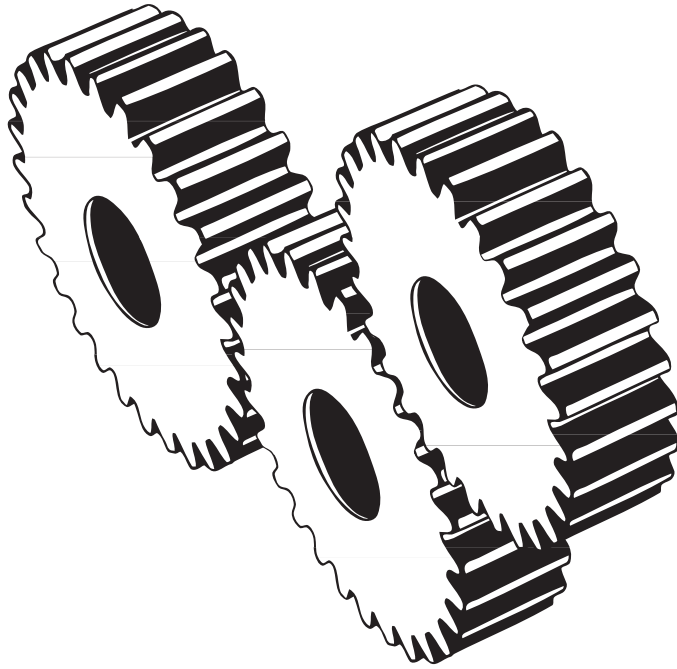
/* Free grab and warp buffers. */
case FREE:
{
    if (NativeGrabBufId)
        imBufFree(NativeSysThreadId, NativeGrabBufId);
    if (NativeWarpResultBufId)
        imBufFree(NativeSysThreadId, NativeWarpResultBufId);
    if (NativeWarpBufId)
        imBufFree(NativeSysThreadId, NativeWarpBufId);
 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

```

Part II: The MIL-Lite reference

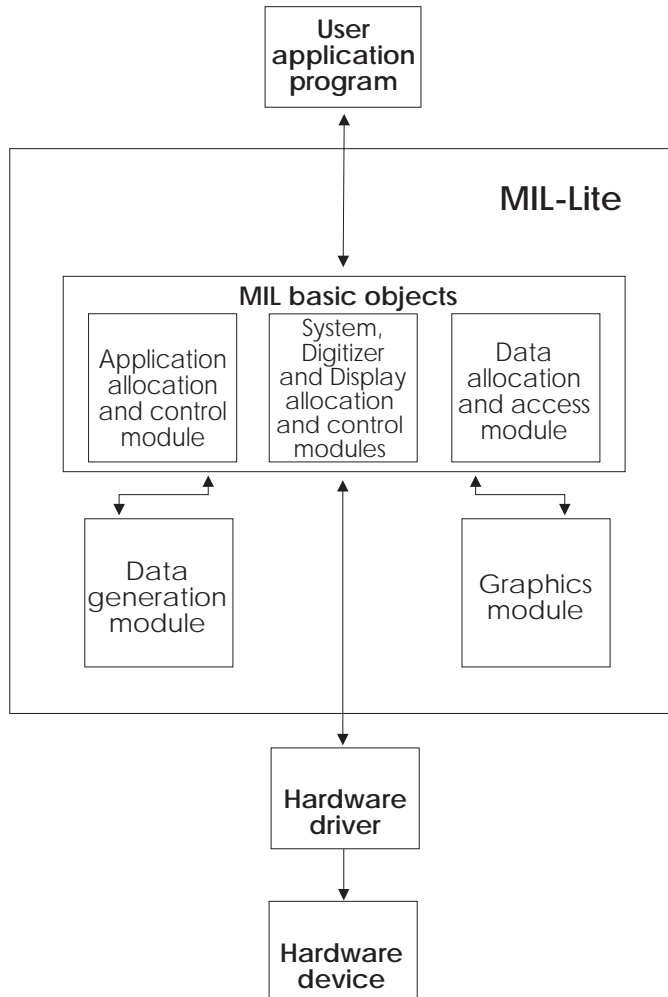


Putting thoughts into motion....

Chapter 13: Programming with MIL

A MIL overview

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



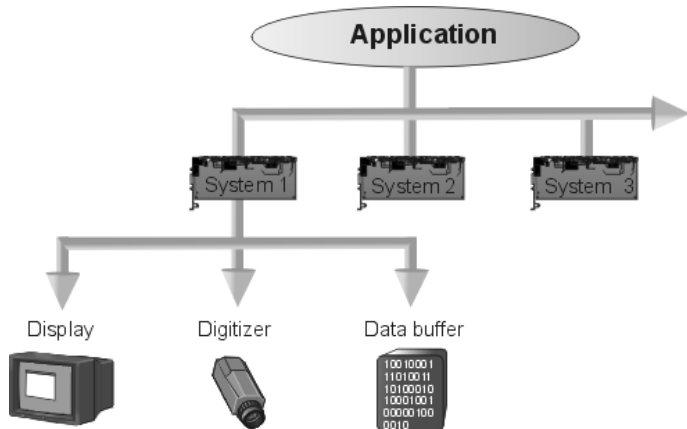
Part I of this manual, *Using MIL-Lite*, describes how to solve typical applications using the library. Code examples are also provided.

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. 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. You can 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 (OR USER-SPECIFIED)\MIL\LIBRARY\WINNT\MSC\DLL* 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 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. Note, several of these devices can be allocated at the same time; you use their identifiers to select between them. 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 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

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.

With MIL, you can detect errors by having them reported to the Host screen, and by checking the system error code for them. You enable or disable error reporting to the screen with **MappControl()**. By default, errors are reported to the screen.

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. Refer to *Appendix D: Troubleshooting* for further information on handling error messages.

Tracing an application

Debugging an application

When developing an application, it is often useful to trace the command calls made by the application in order 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 references description* 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.
MbufChildColor()	ParentBufId, Band, BufIdPtr	Allocate a child data buffer within a color parent buffer.
MbufChildColor2d()	ParentBufId, Band, OffX, OffY, SizeX, SizeY, BufIdPtr	Allocate a child data buffer within a color parent buffer.
MbufChild1d()	ParentBufId, OffX, SizeX, BufIdPtr	Allocate a 1D child data buffer.
MbufChild2d()	ParentBufId, OffX, OffY, SizeX, SizeY, BufIdPtr	Allocate a 2D child data buffer.
MbufClear()	DestImageBufId, Color	Clears a buffer to a specified color.
MbufControl()	BufId, ControlType, ControlValue	Control specified buffer features.
MbufControlNeighborhood()		
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, FileFormatBufId, 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, FileFormatBufId, 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.

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

The display allocation and control module

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

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

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

The basic data generation module

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

Basic data generation commands	Command parameters	Description
MgenLutFunction()	LutBufId, Func, a, b, c, StartIndex, StartXValue, EndIndex	Generate data into a LUT buffer using a specified standard mathematical function.
MgenLutRamp()	LutId, StartIndex, StartValue, EndIndex, EndValue	Generate ramp data into a LUT buffer.

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.

Basic graphics commands	Command parameters	Description
MgraFontScale()	GraphContId, XFontScale, YFontScale	Set the font scale of a graphics context.
MgraFree()	GraphContId	Free a graphics context.
MgraInquire()	GraphContId, InquireType, UserVarPtr	Inquire about the graphic parameters.
MgraLine()	GraphContId, DestImageBufId, XStart, YStart, XEnd, YEnd	Draw a line.
MgraRect()	GraphContId, DestImageBufId, XStart, YStart, XEnd, YEnd	Draw a rectangle.
MgraRectFill()	GraphContId, DestImageBufId, XStart, YStart, XEnd, YEnd	Draw a filled rectangle.
MgraText()	GraphContId, DestImageBufId, XStart, YStart, String	Write text.

The 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 14: 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)`

<code>long InitFlag;</code>	Initialization flag
<code>MIL_ID *ApplicationIdPtr;</code>	Storage location for application identifier
<code>MIL_ID *SystemIdPtr;</code>	Storage location for system identifier
<code>MIL_ID *DisplayIdPtr;</code>	Storage location for display identifier
<code>MIL_ID *DigIdPtr;</code>	Storage location for digitizer identifier
<code>MIL_ID *ImageBufIdPtr;</code>	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:

<code>M_COMPLETE</code>	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.
<code>M_PARTIAL</code>	Initialize MIL to its default state, but do not download any system's resident software.
<code>M_SETUP</code>	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.

The **ControlType** and **ControlValue** parameters specify the type of event to control and the flag with which to control the event. These parameters should be set according to the following combinations:

ControlType	ControlValue	Result
M_ERROR	M_PRINT_ENABLE	Enable printing of error messages (default)
M_ERROR	M_PRINT_DISABLE	Disable printing of error messages
M_TRACE	M_PRINT_ENABLE	Enable printing of function names and parameters
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
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.

ControlType	ControlValue	Result
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.

In multi-thread environments, an **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. 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 ignore calls from other threads. To restore all-thread trace printing, within the same thread call **MappControl**(`M_TRACE+M_THREAD_CURRENT`, `M_DEFAULT`).

If error printing is disabled, you can still check for error, using **MappGetError()**.

Note, if parameter checking is disabled to accelerate an application, unpredictable behavior can be expected when passing invalid parameters to a function.

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) *

Thread ControlType	ControlValue	Result
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.*

Event ControlType	ControlValue	Result
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).

Examplemthread.c

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.

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)**

long ErrorType;	Error type
void *ErrorPtr;	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
M_CURRENT	Get the error code returned by the last command call. The system current-error 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 trying to execute the function.
M_CURRENT_SUB_NB	Get the number of error subcodes associated with the current error.
M_CURRENT_SUB_1...3	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 M_NULL_ERROR .
M_CURRENT_FCT	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()**).

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.
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
<code>M_GLOBAL</code>	Error code.
<code>M_GLOBAL_SUB_NB</code>	Number of error subcodes.
<code>M_GLOBAL_SUB_1</code>	Error subcode 1.
<code>M_GLOBAL_SUB_2</code>	Error subcode 2.
<code>M_GLOBAL_SUB_3</code>	Error subcode 3.
<code>M_GLOBAL_FCT</code>	Function code that caused an error.
<code>M_MESSAGE+M_GLOBAL</code>	Error message.
<code>M_MESSAGE+M_GLOBAL_SUB_1</code>	Error submessage 1.
<code>M_MESSAGE+M_GLOBAL_SUB_2</code>	Error submessage 2.
<code>M_MESSAGE+M_GLOBAL_SUB_3</code>	Error submessage 3.
<code>M_MESSAGE+M_GLOBAL_FCT</code>	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
<code>M_CURRENT_FCT</code>	Code of the function that just started or ended.
<code>M_MESSAGE+M_CURRENT_FCT</code>	Name of the function that just started or ended.
<code>M_PARAM_NB</code>	Number of parameters associated to the function call.
<code>M_PARAM_TYPE+n.</code>	Data type of the n^{th} parameter. This can be: <code>M_TYPE_LONG</code> , <code>M_TYPE_SHORT</code> , <code>M_TYPE_CHAR</code> , <code>M_TYPE_DOUBLE</code> , <code>M_TYPE_PTR</code> , <code>M_TYPE_MIL_ID</code> , or <code>M_TYPE_STRING</code> . (The pointer to a string is invalid after exiting the hook function. For future use, copy and save it.)
<code>M_PARAM_VALUE+n</code>	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; on error, no regular MIL errors are logged.

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.

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:

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_MODIFIED_BUFFER +(BufId)	Call the hook-handler function each time the specified buffer is modified at the end of a MIL function.
M_ERROR_FATAL	Call the hook-handler function before a fatal-error exit.
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_MODIFIED_BUFFER +(BufId)	Detach the hook-handler function being called each time the specified buffer is modified at the end of a MIL function.
M_UNHOOK +M_ERROR_FATAL	Detach the hook-handler function being called before a fatal-error exit.

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);
```

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

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

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

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

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

MappInquire

Synopsis Inquire about the application parameter setting.

Format `long MappInquire(InquireType, UserVarPtr)`

<code>long InquireType;</code>	Type of information to inquire
<code>void *UserVarPtr;</code>	Storage location for inquired information

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

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

InquireType	Description
<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_TRACE</code>	Trace printing mode (<code>M_PRINT_ENABLE</code> or <code>M_PRINT_DISABLE</code>).
<code>M_PARAMETER</code>	Parameter checking mode (<code>M_CHECK_ENABLE</code> or <code>M_CHECK_DISABLE</code>).
<code>M_PROCESSING</code>	Processing compensation mode (<code>M_COMPENSATION_ENABLE</code> or <code>M_COMPENSATION_DISABLE</code>).
<code>M_MEMORY</code>	Memory compensation mode (<code>M_COMPENSATION_ENABLE</code> or <code>M_COMPENSATION_DISABLE</code>).
<code>M_VERSION</code>	Version of MIL library.

InquireType	Description
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_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_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()**

MappModify

Synopsis Modify specified MIL object(s).

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

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

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

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

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

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

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

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

MappTimer

Synopsis Control the MIL timer.

Format **void MappTimer(ControlValue, TimePtr)**

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

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

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

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

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

Example mpatrot.c

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)
<code>M_SIGNED</code>	Signed data	8, 16, or 32
<code>M_UNSIGNED</code>	Unsigned data (default)	1, 8, 16, or 32
<code>M_FLOAT</code>	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. 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.

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_LOSSY	The buffer will be used to hold JPEG lossless 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_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_OVR	Force the buffer in the overlay frame buffer.
M_NON_PAGED	Force the buffer in non-pageable memory.

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

Note Under MIL-Lite, dedicated hardware is required to compress and decompress images. This is not a restriction under MIL.

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

Status Current limitation:

- For M_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).

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.

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 lossy data.	1-band, 8- or 16-bit data.
M_JPEG_LOSSY	The buffer will be used to hold JPEG lossless data in separate fields.	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_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_OVR	Force the buffer in the overlay frame buffer.
M_NON_PAGED	Force the buffer in non-pageable memory.

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

Note Under MIL-Lite, dedicated hardware is required to compress and decompress images. This is not a restriction under MIL.

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

Status Current limitation:

- For **M_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 **MbufAlloc1d()** and **MbufAlloc2d()** to create single band one- or two-dimensional data buffers, respectively.

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

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

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

The **SizeBand** parameter specifies the number of (xy) 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.

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 formats: M_RGB24, and M_RGB48.
M_JPEG_LOSSLESS_INTERLACED	The buffer will be used to hold JPEG lossy data.	1-band, 8- or 16-bit data.
M_JPEG_LOSSY	The buffer will be used to hold JPEG lossless data in separate fields.	1-band 8-bit, and the 3-band 8-bit formats: M_RGB24, M_YUV12, M_YUV9, M_YUV16 + M_PLANAR, and 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, and the 3-band 8-bit format: 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_OVR	Force the buffer in the overlay frame buffer.
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 and one M_OVR+M_ON_BOARD buffer.

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

Note Under MIL-Lite, dedicated hardware is required to compress and decompress images. This is not a restriction under MIL.

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

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

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 also 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.

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 also 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.

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. The image buffer must be a 1-band 8-bit buffer.</p> <p>If and when the image buffer is selected to the display, the required changes occur to produce the display effect of the LUT, unless the display is also associated with a custom LUT (MdispLut0). In single-screen mode, MIL indirectly programs the physical output LUTs with the image's associated LUT (through the use of a Windows palette). In dual-screen mode, the associated LUT is automatically copied to the physical output LUTs.</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 deassociate 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, 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() . 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.

Note that, if the buffer contains any data, setting one of these control types automatically deletes the data. This is because, for MIL to decompress the buffer's data, it must know the control values that were used in the compression. If you change one of these controls, MIL will be unable to decompress the data and the data is therefore irrelevant.

ControlType	ControlValue	Description
M_HUFFMAN_AC	ID of buffer with M_ARRAY attribute	Associate an AC Huffman table to the buffer. Only used for lossy compressions. If the buffer is 3-band, the same table is applied to all bands.
M_HUFFMAN_AC_LUMINANCE	ID of buffer with M_ARRAY attribute	Associate an AC Huffman table to the buffer. Only used for lossy compressions of YUV buffers. The table is applied only to the Y band.

ControlType	ControlValue	Description
M_HUFFMAN_AC_CHROMINANCE	ID of buffer with M_ARRAY attribute	Associate an AC Huffman table to the buffer. Only used for 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. 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 available for YUV buffers. The table is applied only to the Y band. Can only be used if the compressed image buffer is of a lossy type.
M_HUFFMAN_DC_CHROMINANCE	ID of buffer with M_ARRAY attribute	Associate a DC Huffman table to the buffer. Only available for 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 lossless compressions, use predictor #0 (no prediction), predictor #1 (the "pixel-to-the-left" predictor), or predictor #2 (the "pixel-above" predictor). If the buffer is 3-band, the same predictor is applied to all bands.
M_Q_FACTOR	integer value between 1 and 99; default value is 50	Quantization factor for lossy compressions. The higher the factor, the more the compression, but the lower the image quality. If the buffer is 3-band, the same factor is applied to all bands.
M_Q_FACTOR_LUMINANCE	integer value between 1 and 99; default value is 50	Quantization factor for 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.

ControlType	ControlValue	Description
M_Q_FACTOR_CHROMINANCE	integer value between 1 and 99; default value is 50	Quantization factor for 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. Only used for lossy compressions. If the buffer is 3-band, the same table is applied to all bands.
M_QUANTIZATION_LUMINANCE	ID of buffer with M_ARRAY attribute	Associate a quantization table to the buffer. Only used for 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 lossy compressions of YUV buffers. The table is applied to the U and V bands.
M_RESTART_INTERVAL	any integer value; default value is 8	Place restart markers after every n rows of data (for lossless compressions) or after every n 8x8 blocks of data (for lossy compressions).

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

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 has an M_COMPRESS specifier and the destination buffer does not, the data will be automatically decompressed. If the destination buffer has an M_COMPRESS specifier and the source buffer does not, the data will be automatically compressed. If both buffers have M_COMPRESS specifiers but different compression types, the data will be re-compressed according to the settings in the destination buffer.

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.

If the source buffer has an M_COMPRESS specifier and the destination buffer does not, the data will be automatically decompressed. If the destination buffer has an M_COMPRESS specifier and the source buffer does not, the data will be automatically compressed. If both buffers have M_COMPRESS specifiers but different compression types, the data will be re-compressed according to the settings in the destination buffer.

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 to copy
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 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.

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 'long', 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 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 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.

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

Note Under MIL-Lite, dedicated hardware is required to compress and decompress images. This is not a restriction under MIL.

This function is optimized for packed binary buffers.

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

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 (xy) 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.

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.

M_RGB15+M_PACKED	16-bit packed pixels (XRGB 1:5:5:5). Note that when accessing an M_RGB15+M_PACKED buffer as a 3-band 8-bit buffer, the least significant bits are set to 0.
M_RGB16+M_PACKED	16-bit packed pixels (RGB 5:6:5). Note that when accessing an M_RGB16+M_PACKED buffer as a 3-band 8-bit buffer, the least significant bits are set to 0.
M_BGR24+M_PACKED	24-bit (BGR) packed pixels.
M_BGR32+M_PACKED	32-bit (BGR) packed pixels.
M_RGB24+M_PLANAR	24-bit (RGB) planar pixels
M_YUV9+M_PLANAR	YUV9 planar standard.
M_YUV12+M_PLANAR	YUV12 planar standard.
Board-dependent internal storage format specifiers:	
M_YUV16+M_PACKED	YUV16 packed (4:2:2) standard.
M_YUV16_UYVY+M_PACKED	YUV16 packed (4:2:2) standard.
M_YUV16_YUYV+M_PACKED	YUV16 packed (4:2:2) standard.
M_YUV24+M_PLANAR	YUV24 planar standard.

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

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

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

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

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

Note Under MIL-Lite, dedicated hardware is required to compress and decompress images. This is not a restriction under MIL.

This function is optimized for packed binary buffers.

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

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

MbufDiskInquire

Synopsis Inquire about the buffer data in a file.

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

char *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 isn't 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.

InquireType	Description
M_FILE_FORMAT	MIL identifier (MIL_ID) of the file format. See MbufExport() and MbufExportSequence() for all supported file formats.
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, FileFormatBufId, SrcBufId)`

<code>char *FileName;</code>	Destination file name
<code>MIL_ID FileFormatBufId;</code>	File format specification identifier
<code>MIL_ID SrcBufId;</code>	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, 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 **FileFormatBufId** parameter specifies the identifier of the information buffer containing the file conversion format. Predefined file format identifiers are available for the most commonly used file formats:

FileFormatBufId	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.

FileFormatBufId	Description
M_BMP	Save the buffer contents in BMP file format. The BMP file format that is used is the standard Windows format.
M_JPEG_LOSSLESS	Save the buffer contents in a JPEG-lossless file format. If the buffer is 3-band, 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 an a non 8-bit or a non 16-bit buffer in

the M_JPEG_LOSSLESS or M_JPEG_LOSSLESS_INTERLACED formats, 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 1-bit per pixel format (data is stored in 3-bands, 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 with M_MIL or M_TIFF file formats, M_PLANAR can be added (for example, M_TIFF+M_PLANAR) in order to save a color image in planar, rather than packed, mode.

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

Note This function is optimized for packed binary buffers.

Under MIL-Lite, dedicated hardware is required to export compressed images. This is not a restriction under MIL.

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

MbufExportSequence

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

Format **void MbufExportSequence(FileName, FileFormatId, BufArrayPtr, NumberOfImages, FrameRate, ControlFlag)**

char *FileName;	File name
MIL_ID FileFormatId;	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 **FileFormatId** parameter specifies the format of the file. It can be set to:

M_AVI_MJPG	An AVI format used to hold JPEG lossy interlaced, YUV16 packed image buffers. The image buffers must be in this format or in a non-compressed 8-bit format before calling this function (in the latter case, they will be converted appropriately). If the image buffers are in any other format, they will not be exported and an error will be generated.
M_AVI_DIB	An AVI format used to hold non-compressed 8-bit image buffers. If necessary, the image buffers will be converted to a non-compressed 8-bit format before exporting.
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 might not be readable by Windows NT's 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. It can be set to:

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

Note Under MIL-Lite, dedicated hardware is required to export compressed sequences. This is not a restriction under MIL.

See also **MbufImportSequence()**

MbufFree

Synopsis Free a data buffer.

Format **void MbufFree(BufId)**

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

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

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

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

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

MbufGet

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

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

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

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

The **SrcBufId** parameter specifies the identifier of the source buffer.

The **UserArrayPtr** parameter specifies the address of the user array in which to copy source buffer data. Ensure that the user array is large enough to accommodate the data from the source buffer. **MbufGet()** assumes that the array is of the same data type and depth as the source buffer's bands.

Note, for multi-band buffers, **MbufGet()** behaves like **MbufGetColor**(SrcBufId, M_PLANAR, M_ALL_BAND, UserArrayPtr). Refer to **MbufGetColor()** for more details.

Note This function is optimized for packed binary buffers.

See also **MbufGet1d()**, **MbufGet2d()**, **MbufGetColor()**, **MbufPut()**, **MbufPut1d()**, **MbufPut2d()**, **MbufPutColor()**

MbufGetColor

Synopsis Get data from one or all bands of a buffer and place it in a user-supplied array.

Format **void MbufGetColor(SrcBufId, DataFormat, Band, UserArrayPtr)**

MIL_ID SrcBufId;	Source buffer identifier
long DataFormat;	Data format of the user array
long Band;	Color band of source buffer
void *UserArrayPtr;	Destination user array

Description This function copies data from one or all color bands of a specified MIL source buffer to a user-supplied array.

The **SrcBufId** parameter specifies the identifier of the source buffer. The internal data format of the source buffer need not match the specified data format of the user-supplied array; an internal conversion will be performed if necessary. Note, however, if the formats do match the operation will be much faster.

The **DataFormat** parameter specifies the data format to use to save the data in the user array. Note that Sx and Sy denote the source width and height, respectively. This parameter must be set to one of the following values:

DataFormat	Description
M_SINGLE_BAND	Copy a single color band. The user array must be of the same type as the source buffer and have a size of Sx x Sy.
M_BGR24+M_PACKED	Copy three bands in an interleaved manner (BGRBGR). The source buffer must be a three-band, 8-bit buffer and the user array must have a size of Sx x Sy x 3 bytes (Sx x Sy x 3char).
M_BGR32+M_PACKED	Copy three bands in an interleaved manner (BGRXBGRX). The source buffer must be a three-band, 8-bit buffer and the user array must have a size of Sx x Sy x 4 bytes (Sx x Sy x long).
M_RGB15+M_PACKED	Copy three bands in an interleaved manner (RGB 5:5:5). The source buffer must be a single-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 single-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.

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).

DataFormat	Description
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 $S_x \times S_y \times 4$ bytes ($S_x \times S_y \times 4$ long).
M_RGB15+M_PACKED	Copy three bands in an interleaved manner (RGB 5:5:5). The source buffer must be a single-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 single-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 $S_x \times S_y \times \text{number of color band of the source buffer}$. 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.

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.

Note This function is optimized for packed binary buffers.

See also **MbufGet()**, **MbufGet1d()**, **MbufGet2d()**, **MbufPut()**, **MbufPut1d()**, **MbufPut2d()**, **MbufPutColor()**, **MbufPutColor2d()**

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()**

MbufImport

Synopsis Import data from a file into a data buffer.

Format `MIL_ID MbufImport(FileName, FileFormatBufId, Operation, SystemId, BufIdPtr)`

<code>char *FileName;</code>	Source file name
<code>MIL_ID FileFormatBufId;</code>	File format specification identifier
<code>long Operation;</code>	Import operation
<code>MIL_ID SystemId;</code>	System identifier
<code>MIL_ID *BufIdPtr;</code>	Buffer identifier (returned or given)

Description This function imports data, of the specified format, from a file into a MIL data buffer on the specified system. 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 tries to allocate one that can be used in all of the above operations except for acquisition (`M_GRAB`). If it is still unsuccessful, it tries to remove the `M_DISP` attribute, leaving the buffer with the `M_IMAGE` attribute only. If it still cannot allocate the image buffer, it generates an error. If this happens, you can use **MbufImport()** with `M_LOAD` to load the image into a previously allocated buffer.

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.

The **FileName** parameter specifies the name of the file from which to get the data.

The **FileFormatBufId** parameter specifies the identifier of the information buffer containing the file conversion format. Predefined file format identifiers are available for the most commonly used file formats:

M_MIL	Import data that is in MIL file format.
M_TIFF	Import data that is in TIFF file format (only available for image buffers). The TIFF 6.0 specification is used.
M_BMP	Import data that is in BMP file format (only available for image buffers). The standard Windows BMP format is used.
M_RAW	Import data that is in RAW file format.
M_JPEG_LOSSLESS	Import a JPEG-lossless image.
M_JPEG_LOSSY	Import a JPEG-lossy image.

M_JPEG_LOSSLESS_INTERLACED	Import a JPEG-lossless image stored in two separate fields. If the buffer is 3-band, the buffer will be stored in RGB format. Only available for image buffers.
M_JPEG_LOSSY_INTERLACED	Import a JPEG-lossy image stored in two separate fields. If the buffer is 3-band, the data will always be stored in YUV16 packed format. Only available for image buffers.
M_JPEG_LOSSY_RGB	Import a 3-band JPEG-lossy image that is in RGB format.
M_DEFAULT	Automatically determine the file format. If the file format is not supported, its data will be treated in RAW file format.

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

M_RESTORE	Data from the specified file is imported into an automatically allocated MIL data buffer .
M_LOAD	Data from the specified file is imported into a previously allocated MIL data buffer .

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.

Note, you cannot restore (M_RESTORE) a RAW data file (M_RAW) because its dimensions are unknown.

Using **MbufDiskInquire()**, you can inquire about the dimensions of a buffer saved in a file (except for RAW files) without importing it.

The **SystemId** parameter specifies the system on which the MIL buffer will be allocated. 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 **BuIdPtr** 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 specified variable 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 Under MIL-Lite, dedicated hardware is required to import compressed images. 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, FileFormatId, Operation, SystemId, BufArrayPtr, StartImage, NumberOfImages, ControlFlag)`

char *FileName;	File name
MIL_ID FileFormatId;	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 **FileFormatId** parameter specifies the format of the file. It can be set to:

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. It can be set to:

M_LOAD	Import the sequence into previously allocated buffers.
M_RESTORE	Import the sequence into automatically allocated buffers.

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

For an M_LOAD operation, set the **SystemId** parameter to M_NULL.

The **BufArrayPtr** parameter specifies the address of the array containing the buffer identifiers (for an M_LOAD operation) or the address of the array in which to store the new buffer identifiers (for an M_RESTORE operation).

For an M_LOAD operation, the destination buffers should be large enough to hold the imported images. If you are importing compressed images into buffers with only an M_IMAGE specifier, the images will be automatically decompressed. If you are importing decompressed images into buffers with an M_IMAGE+M_COMPRESS specifier, the images will be automatically compressed.

For an M_RESTORE operation, the destination buffers will be allocated with an appropriate size and type to hold the images. For example, if you are importing compressed images, the destination buffers will have an M_IMAGE+M_COMPRESS specifier. If an M_RESTORE operation fails, zero will be written for the buffer identifiers.

The **StartImage** parameter specifies the first image in the sequence to import. Images start at 0.

The **NumberOfImages** parameter specifies the number of images, starting at **StartImage**, to import. The array pointed to by **BufArrayPtr** should be at least as big as this number. Note that you can inquire about the number of images in an *.avi file using **MbufDiskInquire()**.

The **ControlFlag** parameter specifies the function's control flag. This parameter must be set to one of the following:

ControlFlag	Description
M_DEFAULT	Open the AVI file, read the specified images, and then close the file.
M_OPEN	Open the AVI file for reading, and set the pointer to the first image. BufArrayPtr , NumberOfImages , and StartImage should be set to M_NULL.

ControlFlag	Description
M_READ	Read the specified images in the AVI file, starting at the specified StartImage position. To read the image at the current read position, set StartImage to M_DEFAULT. After the read operation, the file pointer is left at the position of the next image, ready for the next M_READ operation.
M_CLOSE	Close the AVI file after reading, and (re)set the pointer position to the first image. BufArrayPtr , NumberOfImages , and FrameRate should be set to M_NULL.

Note Under MIL-Lite, dedicated hardware is required to import compressed sequences. This is not a restriction under MIL.

See also **MbufDiskInquire()**, **MbufExportSequence()**

MbufInquire

Synopsis Inquire about a data buffer parameter setting.

Format `long MbufInquire(BufId, InquireType, UserVarPtr)`

MIL_ID BufId;	Source buffer identifier
long InquireType;	Type of information about which to inquire
void *UserVarPtr;	Storage location for requested information

Description This function inquires about a specified MIL buffer parameter setting. This function is useful, for example, to check the size of a buffer restored from disk.

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

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

InquireType	Description
M_SIZE_X	Width of the buffer.
M_SIZE_Y	Height of the buffer.
M_SIZE_BAND	Number of buffer color bands.
M_SIZE_BIT	Depth per band, in bits.
M_SIZE_BYTE	Size of the buffer, in bytes.
M_SIZE_BYTE_PER_PIXEL	Depth per pixel, 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 only 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.</p> <p>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.

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.
M_RESTART_INTERVAL	Number of lines between restart markers (for lossless compressions) or number of 8x8 blocks of data between restart markers (for lossy compressions).
M_HUFFMAN_DC	Identifier of the array buffer containing the DC Huffman table which is associated with the image buffer. For YUV buffers, only the identifier of the array buffer associated with the luminance band (Y) is returned.
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 YUV buffers, only the identifier of the array buffer associated with the luminance band (Y) 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. If the image buffer is not YUV, M_ERROR is returned.
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. If the image buffer is not YUV, M_ERROR 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. If the image buffer is not YUV, M_ERROR is returned.

InquireType	Description
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. If the image buffer is not YUV, M_ERROR is returned.
M_Q_FACTOR	Quantization factor. For YUV buffers, only the quantization factor associated with the luminance band (Y) is returned.
M_Q_FACTOR_LUMINANCE	Quantization factor for the Y band of a YUV image buffer. If the image buffer is not YUV, M_ERROR is returned.
M_Q_FACTOR_CHROMINANCE	Quantization factor for the U and V bands of a YUV image buffer. If the image buffer is not YUV, M_ERROR is returned.
M_QUANTIZATION	Identifier of the array buffer containing the quantization table which is associated with the image buffer. For YUV buffers, only the identifier of the array buffer associated with the luminance band (Y) 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. If the image buffer is not YUV, M_ERROR is returned.
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. If the image buffer is not YUV, M_ERROR is returned.
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.

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_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.

Return value The returned value is the value that represents the setting of the requested MIL buffer attribute, cast as long.

MbufLoad

Synopsis Load data from a file into a data buffer.

Format **void MbufLoad(FileName, BufId)**

char *FileName;	Source file name
MIL_ID BufId;	Destination buffer identifier

Description This function loads data from a file into a previously allocated data buffer. The function detects the file format from the data.

Note, you can perform the same operation as **MbufLoad()** using **MbufImport()**, which uses the specified file format to open the file instead of trying to determine the format from the data.

The **FileName** parameter specifies the name of file from which to load the data buffer.

The **BufId** parameter specifies the identifier of the destination buffer. This buffer must be big enough in depth and dimensions to hold the data; if not, some data is clipped. For example, if the data is deeper than the buffer, the most-significant bits of the data are truncated when loaded into the buffer. If the buffer depth is greater than that of the data, the data is zero or sign-extended (depending on the data type) when loaded into the buffer. If the buffer is larger in size than the data, exceeding areas of the buffer are unaffected.

When loading an image file that was saved with an associated LUT (color palette), the LUT is also loaded and associated with the destination image buffer. You can obtain the identifier of the associated LUT, using **MbufInquire()**.

Note Under MIL-Lite, dedicated hardware is required to load compressed images. This is not a restriction under MIL.

See also **MbufImport(), MbufExport(), MbufSave(), MbufRestore(), MbufInquire(), MbufControl()**

MbufPut

Synopsis Put data from a user-supplied array into a data buffer.

Format **void MbufPut(DestBufId, UserArrayPtr)**

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

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

The **DestBufId** parameter specifies the identifier of the destination buffer.

The **UserArrayPtr** parameter specifies the address of the user array from which to copy data into the destination buffer. Ensure that there are enough entries in the user array to fill the destination buffer. **MbufPut()** assumes that the array is of the same data type and depth as the destination buffer's bands.

Note, for multi-band buffers, **MbufPut()** behaves like **MbufPutColor(DestBufId, M_PLANAR, M_ALL_BAND, UserArrayPtr)**. See **MbufPutColor()** for more details.

Example mconvol.c

See also **MbufPut1d()**, **MbufPut2d()**, **MbufPutColor()**, **MbufGet()**, **MbufGet1d()**, **MbufGet2d()**, **MbufGetColor()**

MbufPutColor

Synopsis Put data from a user-supplied array into one or all bands of a data buffer.

Format **void MbufPutColor(DestBufId, DataFormat, Band, UserArrayPtr)**

MIL_ID DestBufId;	Destination buffer identifier
long DataFormat;	Data format of source user array
long Band;	Color band in destination buffer
void *UserArrayPtr;	Source user array

Description This function copies data from a user-supplied array to one or all bands of a specified MIL destination buffer.

The **DestBufId** parameter specifies the identifier of the destination buffer. The internal data format of the destination buffer need not match the specified data format of the user-supplied array; an internal conversion will be performed if necessary. Note, however, if the formats do match the operation will be much faster.

The **DataFormat** parameter specifies the data format of the user-supplied array; this information is required to properly copy the data. Note that Dx and Dy denote the destination width and height, respectively. This parameter must be set to one of the following values:

DataFormat	Description
M_SINGLE_BAND	Copy to a single color band. The user array must be of the same type as the destination buffer and have a size of Dx x Dy.
M_BGR24+M_PACKED	Copy to three bands in an interleaved manner (BGRBGR). The destination buffer must be a three-band, 8-bit buffer and the user array must have a size of Dx x Dy x 3 bytes (Dx x Dy x 3char).
M_BGR32+M_PACKED	Copy to three bands in an interleaved manner (BGRXBGRX). The destination buffer must be a three-band, 8-bit buffer and the user array must have a size of Dx x Dy x 4 bytes (Dx x Dy x long).
M_RGB15+M_PACKED	Copy to three bands in an interleaved manner (RGB 5:5:5). The destination buffer must be a single-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 single-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.

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).

DataFormat	Description
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 single-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 single-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.

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)`

<code>char *FileName;</code>	Source file name
<code>MIL_ID SystemId;</code>	System identifier
<code>MIL_ID *BufIdPtr;</code>	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, this function tries to allocate one that can be used in all of the above operations except for acquisition (M_GRAB). If it is still unsuccessful, it tries to remove the M_DISP attribute, leaving the buffer with the M_IMAGE attribute only. If it still cannot allocate the image buffer, it generates an error. If this happens, you can use **MbufLoad()** to load the image in a previously allocated buffer.

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 Under MIL-Lite, dedicated hardware is required to restore compressed images. 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)**

char *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
char *DataFormat;	Data format name or file name
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 input device data format (generally, M_CH0). Some digitizers have multiple input channels. You can switch to another channel by 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 data format or the name of the file in which the data format of the input device can be found. Depending on the target system, different data formats can be supported.

See the appendix in this manual that applies to your specific board, the *read.me* file of the MIL drivers, or the user guide of your specific board for the valid values. This parameter can also be set to `M_CAMERA_SETUP`, which indicates to MIL to use the camera format specified in the *milsetup.h* file.

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_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.

ControlType	Description & ControlValue	
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.
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_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. <i>See MIL/MIL-Lite Board Specific Notes for exceptions.</i>

ControlType	Description & ControlValue	
M_GRAB_FIELD_NUM	Control the number of fields to grab when grabbing data with MdigGrab0 .	
M_GRAB_FRAME_NUM	Control the number of frames to grab when grabbing data with MdigGrab0 .	
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_HSYNC	Trigger on each Hsync signal.
	M_VSYNC	Trigger on each Vsync signal.
	M_TIMER1	Trigger on timer 1 signal.
	M_TIMER2	Trigger on timer 2 signal.

ControlType	Description & ControlValue	
M_GRAB_TRIGGER_MODE	Set the hardware trigger activation mode.	
	M_EDGE_RISING	Low to high signal variation (valid with exposure).
	M_EDGE_FALLING	High to low signal variation (valid with exposure).
	M_LEVEL_LOW	Minimum signal level (not valid with exposure).
	M_LEVEL_HIGH	Maximum signal level (not valid with exposure).
	M_DEFAULT	The trigger mode in the DCF file or, if none, M_EDGE_RISING.
M_GRAB_TRIGGER	Set the grab trigger detection state.	
	M_ENABLE	Enable trigger detection.
	M_DISABLE	Disable trigger detection.
	M_DEFAULT	The trigger state from the DCF file or, if none, M_DISABLE.
	M_ACTIVATE	Start the grab immediately (for software trigger). An asynchronous or continuous grab must be in progress.
M_GRAB_EXPOSURE_BYPASS (If the board supports exposures; See Matrox Board Specific Notes)	Activate the manual or automatic exposure model (see <i>Grabbing with triggers</i> in the <i>Matrox Imaging Library User Guide</i>):	
	M_ENABLE	Manual exposure model.
	M_DISABLE	Automatic exposure model.
	M_DEFAULT	Same as M_DISABLE.
For the following M_GRAB_EXPOSURE... control types, you can add M_TIMER1 or M_TIMER2 in manual exposure mode, to control the different on-board exposure timers. When omitted, Timer1 is assumed.		
M_GRAB_EXPOSURE (If the board supports exposures; See Matrox Board Specific Notes)	When using a software trigger source, use this control type to activate the specified grab exposure timer. When using a non-software trigger source, enable or disable the specified grab exposure timer. Note, the M_GRAB_EXPOSURE control type has no effect when grabbing using the automatic exposure model.	
	M_ACTIVATE	Activate a software trigger for the specified exposure timer.
	M_ENABLE	Enable exposure timer.
	M_DISABLE	Disable exposure timer.
	M_DEFAULT	same as .dcf (non-software trigger source).

ControlType	Description & ControlValue	
M_GRAB_EXPOSURE_TIME (If the board supports exposures; See Matrox Board Specific Notes)	<p>Set the time (in nsec) for the active portion of the exposure signal (that is, the exposure time). M_DEFAULT has the same effect as the setting in the digitizer's DCF.</p> <p>When using the automatic exposure model, if a single timer cannot generate the required exposure time, MIL automatically sets up connections with the second timer to generate the requested exposure time length. If ControlValue is set to 0, exposure is disabled and the grab is performed immediately.</p> <p>Note, an error is returned if the specified exposure time cannot be generated.</p>	
M_GRAB_EXPOSURE_MODE (If the board supports exposures; See MIL/MIL-Lite Board Specific Notes)	Set the exposure signal's polarity:	
	M_LEVEL_HIGH	
	M_LEVEL_LOW	
	M_DEFAULT	Same as DCF.
M_GRAB_EXPOSURE_TIME_DELAY (If the board supports exposures; See MIL/MIL-Lite Specific Notes)	<p>Set the delay (in nsec) between the trigger and the start of exposure. If M_DEFAULT, same value as DCF.</p> <p>Note, an error is returned if the specified delay cannot be generated.</p>	
M_GRAB_EXPOSURE_TRIGGER_MODE (If the board supports exposures; See MIL/MIL-Lite Board Specific Notes)	Set the trigger activation mode for specified timer.	
	M_DEFAULT	Same as the .dcf file.
	M_EDGE_RISING	Low-to-high signal variation.
	M_EDGE_FALLING	High-to-low signal variation.
M_GRAB_EXPOSURE_SOURCE (If the board supports exposures; See MIL/MIL-Lite Board Specific Notes)	<p>Select the trigger source for the specified exposure timer if the hardware supports it.</p> <p>The M_GRAB_EXPOSURE_SOURCE control type has no effect when grabbing using the automatic exposure model.</p>	
	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.

ControlType	Description & ControlValue	
M_GRAB_EXPOSURE_SOURCE (If the board supports exposures; See <i>Matrox Board Specific Notes</i>) (cont.)	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.

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.

Status Hardware limitations:

On certain platforms, the next MIL command called after **MdigGrabContinuous()** must be **MdigHalt()**; otherwise, errors can occur.

Examples mdispovr.c, mwindisp.c, mfocus.c, mdbproc.c, mgrabhk.c, mgrabseq.c, msubtrac.c, msurvey.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.

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. This function is not supported on all systems. See *MIL/MIL-Lite Board-Specific Notes* to verify if this function is supported on your board.

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, these defines can be combined with M_UNHOOK to unhook the function.

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 a camera is connected, but not grabbing, the 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.

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()).

InquireType	Description
M_HUE_REF	Digitizer hue reference level (MdigReference()).
M_SATURATION_REF	Digitizer saturation reference level (MdigReference()).
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_FRAME_NUM	Number of frames grabbed when MdigGrab() is called (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.

InquireType	Description
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).
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`.

<code>M_CH0_REF</code>	Inquire about reference level on channel 0 (default).
<code>M_CH1_REF</code>	Inquire about reference level on channel 1.
<code>M_CH2_REF</code>	Inquire about reference level on channel 2.
<code>M_CH3_REF</code>	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 a character array.
- `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 LUT buffer pixel depth and number of entries must match those of the digitizer, and the LUT buffer must either have a single color band or match the number of color bands of the digitizer. If the LUT buffer has a single color band, its data is loaded into the LUTs of each of the digitizer's color bands. 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 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
char *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 on the target system must be allocated in order to display an image buffer.

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 that is required. This parameter can be set to one of the following:

M_DEFAULT	Any available display.
M_DEV0	The first display on the specified system.
....,	
M_DEV15	The sixteenth display on the specified system.

The **DispFormat** parameter specifies the name of the display format or the name of the file in which the display format is to be found. Under Windows in single-screen mode, **DispFormat** must be set to M_DEFAULT, which when displaying from an imaging system with an on-board display, sets the display resolution of the main (underlay) frame buffer to that of the overlay (VGA) frame buffer. Under Windows in dual-screen mode, **DispFormat** can be set to a string that specifies the required display resolution; see the *MIL/MIL-Lite Board-Specific Notes* manual for the formats supported by

your board. Under Windows in dual-screen mode, **DispFormat** can also be set to M_DEFAULT, which indicates that MIL should use the format specified in the *milsetup.h* file.

The **InitFlag** parameter specifies the display mode of your system. Depending on your system’s display configuration, **InitFlag** will have a different default. This parameter can be set to one of the following:

M_WINDOWED	<p>The display has a window associated with it. The image buffer selected for display purposes is presented (on-screen) in its own window. 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.</p> <p>This mode is the default allocation mode in a single-screen configuration (M_DEFAULT). If your board has a display section and you are using it in a dual-screen configuration, you can still choose not to use it, and display an image, even a live grabbed image, in windowed mode. In this case, the display is on your Windows desktop.</p>
M_NON_WINDOWED	<p>The display has no window associated with it. You are responsible for moving and tracking this type of display, if required. This is the default for dual-screen mode. In single-screen mode, only 1 non-windowed display can be allocated in the underlay. In dual-screen mode, 2 non-windowed displays can be allocated; one can be allocated in the underlay and one can be allocated in the overlay.</p> <p>Note that this mode is only available on frame grabbers that have an on-board VGA adapter.</p>

MIL automatically selects the most appropriate display architecture, but you can force a particular display architecture by adding one of the following initialization flags to `M_WINDOWED`. See the *MIL User Guide* for a detailed description of these architectures.

<code>M_OVR</code>	Force the display architecture to an overlay/regular display. An overlay/regular display architecture is particularly useful, because, in general, you can associate a LUT with this type of display (Refer to MdispLut() for more details). When in dual-screen mode, your buffer must be allocated with an <code>M_IMAGE+M_OVR+...</code> attribute before it can be selected to an <code>M_OVR</code> display.
<code>M_UND</code>	Force the display architecture to a dedicated underlay display.
<code>M_DDRAW_UND</code>	Force the display architecture to a DirectDraw underlay-surface display.

In windowed mode, 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
<code>M_DISPLAY_ENHANCED</code> <code>M_DISPLAY_8_ENHANCED</code> <code>M_DISPLAY_24_ENHANCED</code> (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 M_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	
The following controls are only available with M_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 for only short periods of time or undesirable results can occur.	
M_DESKTOP_LOCK_TIMEOUT	Control the Windows desktop lock timeout. When debugging an application using DIRECTDRAW, the desktop locks when a breakpoint is found in the code. A timeout value for the lock can be specified as follows: any value in milliseconds, M_DEFAULT (a generally acceptable value), or M_INFINITE (no timeout value is assigned). The default ControlValue is M_INFINITE.	
M_THREAD_PRIORITY	Thread priority.	
	Range:	Priority class:
	1 - 6	Idle.
	7 - 10	Normal.
	11 - 15	High.
	16 - 31	Real-time.

ControlType	Description and ControlValue	
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.	
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	<p>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).</p> <p>If enabled, the MIL identifier of this buffer can be inquired, using MdispInquire().</p> <p>If disabled, the buffer is invalid.</p> <p>Note, this control is only supported on systems with an on-board display section (and are using it for display), and on systems using a Matrox VGA board.</p>	
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 leftmost X coordinate.	
M_WINDOW_INITIAL_POSITION_Y	Set the window client area's initial topmost Y coordinate.	

ControlType	Description and ControlValue
M_WINDOW_KEYBOARD_USE	Activate the keys associated with the display window: M_ENABLE (default) or M_DISABLE. <i>The default key usage is:</i>
	+ Increase the x and y zoom factors.
	- Decrease the x and y zoom factors.
	Pg-up Scroll the buffer up to the previous display section.
	Pg-dn Scroll the buffer down to the next display section.
	Up arrow Scroll the buffer up to the previous line.
	Dn arrow Scroll the buffer down to the next line.
	Left arrow Pan the buffer left by one pixel.
	Right arrow Pan the buffer right by one pixel.
	Ctrl Up arrow Scroll the buffer up to the previous display section.
M_WINDOW_MAXBUTTON	Make the window's maximize button visible: M_ENABLE or M_DISABLE.
M_WINDOW_MENU_BAR	Make the window's menu bar visible: M_ENABLE or M_DISABLE.
M_WINDOW_MENU_BAR_CHANGE	Allow toggling the menu bar presence: M_ENABLE or M_DISABLE.
M_WINDOW_MINBUTTON	Make the window's minimize button visible: M_ENABLE or M_DISABLE.
M_WINDOW_MOVE	Allow window movement: M_ENABLE or M_DISABLE
M_WINDOW_OVERLAP	Allow window to be overlapped by another: M_ENABLE or M_DISABLE (keep window on top).
M_WINDOW_OVR_DESTRUCTIVE	The overlay shown on top of the buffer is allowed to overwrite the buffer's content (to increase display speed or save memory): M_ENABLE or M_DISABLE (default).
M_WINDOW_OVR_FLICKER	The overlay shown on top of the buffer is allowed some flicker (to increase display speed or save memory): M_ENABLE or M_DISABLE (default).
M_WINDOW_PAINT	Force the window's update and paint the whole region: M_DEFAULT or M_NULL.

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 update. 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 leftmost 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 casted to long).	
M_WINDOW_UPDATE	Allow updating of the window display: M_ENABLE or M_DISABLE.	

ControlType	Description and ControlValue	
M_WINDOW_UPDATE_ON_PAINT	M_ENABLE	Update the display on reception of a WM_PAINT message in Windows.
	M_DISABLE	Update the display on reception of a WM_ERASEBKGND message in Windows.
	M_DEFAULT	Allow MILto decide which message to receive before updating the display.
M_WINDOW_ZOOM	Allow window zooming: M_ENABLE or M_DISABLE	
The following controls are only available with windowed displays, and non-windowed displays on a Matrox imaging board with a display section:		
M_WINDOW_OVR_LUT	Associate a LUT with the overlay buffer. 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	<p>Allow annotating the displayed image non-destructively, using MIL's overlay-display mechanism. When enabled in windowed mode, the display is associated with a temporary overlay buffer whenever a buffer is selected on the display. When enabled in non-windowed mode, the display is immediately associated with a temporary overlay buffer. This overlay buffer will annotate the underlying image with an effect called keying, which makes portions of the overlay show through.</p> <p>In windowed mode, the overlay buffer has the same number of bands and is the same size as the selected image buffer. A new temporary overlay buffer is created when a new buffer is selected on the display.</p> <p>In non-windowed mode, the overlay buffer is the same size as the display. The overlay buffer is not modified whenever a new buffer is selected on the display, and is freed when deselecting the image buffer from the display.</p> <p>The MIL identifier of this buffer can be inquired, using MdispInquire().</p> <p>If your board does not have two frame buffer surfaces, a simulated version of the overlay effect is produced through software</p>	
	M_ENABLE	Enable MIL's overlay-display mechanism.
	M_DISABLE (default)	Disable MIL's overlay-display mechanism.

ControlType	Description and ControlValue
The following controls are only available with an M_WINDOWED display on a Matrox MGA display card or a Matrox imaging board with a display section.	
M_HARDWARE_PAN	Use your system's hardware pan options (M_ENABLE) or the software pan options of the display's window (M_DISABLE). The default is M_DISABLE.
M_HARDWARE_ZOOM	Use your system's hardware zoom options (M_ENABLE) or the software zoom options of the display's window (M_DISABLE). The default is M_DISABLE.

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. In windowed mode, the display is closed. In non-windowed mode, the display is blanked.

Note, when displaying a parent buffer, you cannot remove one of its child buffers from the display.

Note, you do not have 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 an image buffer, with an M_DISP attribute, that is currently displayed.

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.

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:

M_FRAME_START	Call the hook-handler function each time a new frame is displayed.
---------------	--

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)`

<code>MIL_ID DisplayId;</code>	Display identifier
<code>long InquireType;</code>	Display parameter to inquire
<code>void *UserVarPtr;</code>	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
<code>M_DISPLAY_MODE</code>	Display mode. <code>M_WINDOWED</code> if the display object is bounded by a movable frame or <code>M_NON_WINDOWED</code> .
<code>M_FORMAT</code>	Display data format (MdispAlloc()).
<code>M_FORMAT_SIZE</code>	Number of characters in the data format string (MdispAlloc()).
<code>M_FRAME_START_HANDLER_PTR</code>	Handler pointer hooked using MdispHookFunction() to the start of a displayed frame (MdispSelect()).
<code>M_FRAME_START_HANDLER_USER_PTR</code>	User pointer hooked using MdispHookFunction() to the start of a displayed frame (MdispSelect()).
<code>M_INIT_FLAG</code>	Display initialization flag (MdispAlloc()).
<code>M_KEY_COLOR</code>	Keying color (MdispOverlayKey()).
<code>M_KEY_CONDITION</code>	Keying condition (MdispOverlayKey()).
<code>M_KEY_MASK</code>	Keying mask (MdispOverlayKey()).
<code>M_KEY_MODE</code>	State of keying mode (MdispOverlayKey()).
<code>M_KEY_SUPPORTED</code>	System support of true keying (<code>M_YES</code> or <code>M_NO</code>).
<code>M_LUT_ID</code>	The identifier of the LUT associated with the display (MdispLut()).
<code>M_LUT_SUPPORTED</code>	Whether a LUT is supported on the specified display (MdispLut()).
<code>M_NATIVE_ID</code>	The display's native identifier, if any.

InquireType	Description
M_NUMBER	Display rank in the system (MdispAlloc()).
M_OWNER_SYSTEM	The identifier of the system on which the display has been allocated (MdispAlloc()).
M_PAN_X	Pan X pixel offset (MdispPan()).
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. In windowed mode, 3 will be returned; in non-windowed mode, 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_THREAD_PRIORITY	Thread priority.
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_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.
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).

InquireType	Description
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_DDRAWSURFACE	Pointer to the DirectDraw primary surface (LPDIRECTDRAWSURFACE) used by a display window (if any) or M_NULL.
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).

InquireType	Description
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_WINDOW_UPDATE	State of window update (M_ENABLE or M_DISABLE).
M_WINDOW_ZOOM	State of display window zooming (M_ENABLE or M_DISABLE).
M_WINDOW_ZOOM_X	Window zoom X factor (controlled by zoom buttons).
M_WINDOW_ZOOM_Y	Window zoom Y factor (controlled by zoom buttons).
The following inquire types are only available with windowed displays, and non-windowed displays on a Matrox imaging board with a display section:	
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.
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 following inquire types are only available with an M_WINDOWED display on a Matrox MGA display card or a Matrox imaging board with a display section.	
M_HARDWARE_PAN	Whether your system's hardware pan options are enabled or disabled.
M_HARDWARE_ZOOM	Whether your system's hardware zoom options are enabled or disabled.

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 a character array.

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. In dual-screen mode, the LUT buffer is loaded into the physical LUTs. In single-screen mode, 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 17:Lookup tables* and *Chapter 18: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 in windowed mode, MIL provides a good default logical palette for the realization of the physical output LUTs. MIL takes into consideration the displayed image, the Windows display driver used, and the VGA physical output LUT capabilities, and produces the best "portability versus visual quality" compromise possible.

By default in non-windowed mode, MIL generates a ramp in the physical output LUTs, which uses the full range of available intensities. This type of mapping is also referred to as a pass-through LUT mapping (or transparent LUT mapping).

- A pseudo-color LUT (M_PSEUDO)

If you set **LutBufId** to M_PSEUDO in windowed mode, the data is loaded in each component of the logical palette. In non-windowed mode, the data is loaded into the physical output LUTs of the display.

- A custom LUT buffer identifier

You can associate a custom LUT (allocated with **MbufAlloc1d()** or **MbufAllocColor()**) 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-mode display and then select the display (**MdispSelect()**), the same data is loaded in each component of the logical palette. In non-windowed-mode, the same data is loaded into each of the physical output LUTs.

If you associate a three-band color LUT buffer (RGB) with a windowed mode display and then select the display, each band of the LUT buffer is loaded into its corresponding component of the logical palette. If you associate a three-band color LUT buffer (RGB) with a non-windowed mode display, each LUT buffer color band is loaded in a different physical output LUT (if a different LUT is available for each display output channel).

Refer to both *Chapter 17: Look-up tables (LUTS)*, as well as *Chapter 18: 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 **MdispLut()**. 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, a LUT buffer cannot be associated with an **M_UND** display.
- 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:

Some hardware systems do not support display LUTs.

See also **MbufAlloc1d()**, **MbufAllocColor()**, **MgenLutRamp()**, **MgenLutFunction()**, **MbufPut()**, **MbufPut1d()**

MdispOverlayKey

Synopsis Enable 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 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()**. Note, keying is only supported in non-windowed mode if you are using a system with an on-board display section.

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 overlay buffer only (no keying).
M_KEY_ON_COLOR	Display the image buffer selected on the display only where the pixels of the 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).

When keying is not enabled, set **KeyMask** to M_NULL.

The **KeyColor** parameter specifies the keying color when keying is enabled (M_KEY_ON_COLOR). When in an 8-bit display mode (display depth), set this parameter to the required 8-bit color index. When in any other display mode, you can set this parameter to:

- An 8-bit grayscale value. This value will be used for each band.
- An RGB value using the following macro:

M_RGB888(red component, green component, blue component)

When keying is not enabled, set **KeyColor** 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 screen 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.

Status Hardware limitations:

Some hardware systems do not support panning and some only support certain panning values.

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, mdispmfc.dsp

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 (if this feature is supported by the target system). 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.

Status Hardware limitations:

- Some hardware systems do not support zooming and some only support certain zoom factors.

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 data in the specified LUT buffer area (**StartIndex** to **EndIndex** inclusive) according to the function specified by **Func** and using the LUT location index and the **StartXValue** as the X value in the equation.

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 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	$a x^2 + bx + c$

The **a**, **b**, **c** parameters specify 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()**

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.

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
void *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 specified 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:

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()**

MgralInquire

Synopsis Inquire about the graphics parameters.

Format **void** **MgralInquire**(**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	void

InquireType	Pointer to a:
M_FONT_X_SCALE	double
M_FONT_Y_SCALE	double
M_OWNER_SYSTEM	MIL_ID

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
char *String;	Null terminated ASCII string

Description This function writes the specified ASCII 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, mwindib.c, mwindisp.c

See also **MgraFont()**, **MgraFontScale()**, **MgraControl()**

MsysAlloc

Synopsis Allocate a hardware system.

Format **MIL_ID MsysAlloc(SystemTypePtr, SystemNum, InitFlag, SystemIdPtr)**

void *SystemTypePtr;	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 or Host system) 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 **SystemTypePtr** parameter specifies the type of system to allocate. This parameter is a pointer to a function that allows communication with the specified system (board). Set this parameter to one of the following values:

SystemTypePtr	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_ORION	Orion type system.

SystemTypePtr	Type of system to allocate
M_SYSTEM_PULSAR	Pulsar type system.
M_SYSTEM_GENESIS	Genesis type system.
M_SYSTEM_CORONA	Corona 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 you want 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_DDRAW	Enable the use of DirectDraw by the system.
M_NO_DDRAW	Disable the use of DirectDraw by the system.
M_DEFAULT	Same as M_COMPLETE.

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 appendix), 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_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). When grabbing to an underlay frame buffer surface, this control type should be left to the default setting.	
	M_ENABLE	Pseudo-live grab is enabled (default).
	M_DISABLE	Pseudo-live grab is disabled.
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)

ControlType	ControlValue & Description	
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 G400 (or higher) video graphics adapter should be used. In addition, this mode can only be used when the grab buffer is selected to a display that is under a DirectDraw underlay-surface display architecture (M_WINDOWED+M_DDRAW_UNDERLAY). (Note that this is the default display mode when the hardware is available)	
	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, when the display is in windowed mode (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.
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.

ControlType	ControlValue & Description	
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_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. When grabbing to an underlay frame buffer surface, this control type should be left to the default setting.	
	M_ENABLE	Live grab is enabled (default).
	M_DISABLE	Live grab is disabled.

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_ORION_TYPE, M_SYSTEM_PULSAR_TYPE, M_SYSTEM_GENESIS_TYPE, or M_SYSTEM_CORONA_TYPE.
M_SYSTEM_NAME	The system name. This inquire type copies the system's name (that is, the board type) 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 displays available on the system.
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_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). Note that, under MIL-Lite, dedicated hardware is required to compress and decompress images. Under the full version of MIL, compression and decompression is supported, whether or not dedicated hardware is present.
M_DUAL_SCREEN_MODE	Whether the system is in dual-screen mode (M_ENABLE or M_DISABLE).
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_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 a character array. The character array must be larger 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_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_DEVO
#define M_DEF_DISPLAY_FORMAT   "M_DEFAULT"
#define M_DEF_DISPLAY_INIT     M_DEFAULT
#define M_DISPLAY_SETUP        M_DEF_DISPLAY_FORMAT
#define M_DEF_DISPLAY_KEY_COLOR 0L
#define M_DEF_DISPLAY_KEY_ENABLE_ON_ALLOC 0L
#define M_DEF_DISPLAY_KEY_DISABLE_ON_FREE 0L

```

The above defaults determine the display type that will be allocated if the **MappAllocDefault() DisplayIdVarPtr** parameter is not set to **M_NULL**. **MappAllocDefault()** macro calls the **MdispAlloc()** command to allocate the display.

M_DEF_DISPLAY_NUM specifies display number on your target system, and M_DEF_DISPLAY_FORMAT specifies the display format. M_DEF_DISPLAY_INIT should be set to M_DEFAULT.

Default digitizer

```
/* ..... */
/* DEFAULT DIGITIZER SPECIFICATION */
#define M_DEF_DIGITIZER_NUM          M_DEVO
#define M_DEF_DIGITIZER_FORMAT      "\\PULSARLIB\DCF\R170_LO.DCF"
#define M_DEF_DIGITIZER_INIT        M_DEFAULT
#define M_DEF_CAMERA_SETUP          M_DEF_DIGITIZER_FORMAT
```

The above defaults determine the digitizer type that will be allocated if the **MappAllocDefault() DigitizerIdVarPtr** parameter is not set to M_NULL. **MappAllocDefault()** macro calls the **MdigAlloc()** command to allocate the digitizer. M_DEF_DIGITIZER_NUM specifies digitizer number on your target system, and M_DEF_DIGITIZER_FORMAT specifies the input data format (or camera output data format). M_DEF_DIGITIZER_INIT should be set to M_DEFAULT.

Default image buffer

```
/* ..... */
/* DEFAULT IMAGE BUFFER SPECIFICATION */
#define M_DEF_IMAGE_NUMBANDS_MIN      1L
#define M_DEF_IMAGE_SIZE_X_MIN        512
#define M_DEF_IMAGE_SIZE_Y_MIN        480
#define M_DEF_IMAGE_SIZE_X_MAX        1280
#define M_DEF_IMAGE_SIZE_Y_MAX        1024
#define M_DEF_IMAGE_TYPE              8+M_UNSIGNED
#define M_DEF_IMAGE_ATTRIBUTE_MIN      M_IMAGE+M_PROC
```

The above defaults determine the image buffer that will be allocated if the **MappAllocDefault() ImageIdVarPtr** parameter is not set to M_NULL. By default, if a color digitizer was specified upon installation, a color buffer (three bands) will be allocated; otherwise, a monochrome buffer is allocated. The **MappAllocDefault()** macro calls the **MbufAlloc2d()** command to allocate a monochrome buffer or **MbufAllocColor()** to allocate a color buffer. The buffer width and height are the maximum between the default display image dimensions M_DEF_IMAGE_SIZE_X_MIN and M_DEF_IMAGE_SIZE_Y_MIN and the default display format size, but never exceed M_DEF_IMAGE_SIZE_X_MAX and M_DEF_IMAGE_SIZE_Y_MAX. M_DEF_IMAGE_TYPE specifies

the depth and range of the data buffer.

M_DEF_IMAGE_ATTRIBUTE_MIN specifies the minimum attributes for the buffer usage.

M_DEF_IMAGE_NUMBANDS_MIN specifies the number of color bands of the buffer.

When you do not want to use defaults

When you do not want to use **MappAllocDefault()**, you can individually specify the allocation of any MIL application, system, digitizer, buffer, or display. The individual allocations must respect the following:

- You must allocate the MIL application before using any other MIL function.
- You must allocate the MIL system after allocating the MIL application and before allocating any digitizer, buffer, or display. You can allocate multiple systems within an application.
- You can allocate multiple digitizers, buffers, or displays within a system.
- When freeing (de-allocating) individually, you must respect the reverse of the order required for allocation.

The following illustrates allocating individually, using a modification of the *mgrab.c* example (appearing in *Chapter 2*).

```
/* 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_DEV0, 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.

Code

```

/*****
/*
 * 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.
MfuncParamChar()	FunctionId, ParamIndex, ParamValue	Register a character parameter.
MfuncParamCheck()	FunctionId	Read the MIL application parameter checking flag.
MfuncParamDouble()	FunctionId, ParamIndex, ParamValue	Register a double parameter.

MIL developer functions	Command parameters	Description
MfuncParamId()	FunctionId, ParamIndex, ParamValue, ParamIs, ParamHasAttr	Register a MIL_ID parameter.
MfuncParamLong()	FunctionId, ParamIndex, ParamValue	Register a long parameter.
MfuncParamPointer()	FunctionId, ParamIndex, ParamValue	Register a pointer parameter.
MfuncParamRegister()	FunctionId	Read MIL application parameter registering flag.
MfuncParamShort()	FunctionId, ParamIndex, ParamValue	Register a short parameter.
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)`

<code>char *FunctionName;</code>	Function name
<code>long ParameterNumber;</code>	Number of parameters passed

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.

Return value The returned value is a MIL identifier for the function; M_NULL on error.

Example mnatfct.c

See also **MfuncStart()**, **MfuncFreeAndEnd()**, **MfuncParamChar()**, **MfuncParamDouble()**, **MfuncParamId()**, **MfuncParamLong()**, **MfuncParamPointer()**, **MfuncParamShort()**, **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
char *ErrorMessage;	Error message to log
char *ErrorSubMessage1;	Sub-error message 1 to log
char *ErrorSubMessage2;	Sub-error message 2 to log
char *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

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.

MfuncParamChar

Synopsis Register a character parameter.

Format **void MfuncParamChar(FunctionId, ParamIndex, ParamValue)**

MIL_ID FunctionId;	Function identifier
long ParamIndex;	Parameter index
char ParamValue;	Parameter value

Description This function allows you to register a character parameter of the current pseudo-MIL function. The **MfuncParamChar()** 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 character parameter.

See also **MfuncAlloc()**, **MfuncStart()**, **MfuncFreeAndEnd()**, **MfuncParamDouble()**, **MfuncParamId()**, **MfuncParamLong()**, **MfuncParamPointer()**, **MfuncParamShort()**, **MfuncParamString()**

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()**, **MfuncParamShort()**, **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(), MfuncParamChar(), MfuncParamDouble(), MfuncParamLong(), MfuncParamPointer(), MfuncParamShort(), 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()**, **MfuncParamChar()**, **MfuncParamDouble()**, **MfuncParamId()**, **MfuncParamPointer()**, **MfuncParamShort()**, **MfuncParamString()**

MfuncParamPointer

Synopsis Register a pointer parameter.

Format **void MfuncParamPointer(FunctionId, ParamIndex, *ParamValue)**

MIL_ID FunctionId;	Function identifier
long ParamIndex;	Parameter index
void *ParamValue;	Parameter value

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 function parameter list. The index of the first parameter is considered to be one.

The **ParamValue** parameter is the value of the pointer parameter.

See also **MfuncAlloc()**, **MfuncStart()**, **MfuncFreeAndEnd()**, **MfuncParamChar()**, **MfuncParamDouble()**, **MfuncParamId()**, **MfuncParamLong()**, **MfuncParamShort()**, **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.

MfuncParamShort

Synopsis Register a short parameter.

Format **void MfuncParamShort(FunctionId, ParamIndex, ParamValue)**

MIL_ID FunctionId;	Function identifier
long ParamIndex;	Parameter index
short ParamValue;	Parameter value

Description This function allows you to register a short parameter of the current pseudo-MIL function. The **MfuncParamShort()** 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 short parameter.

See also **MfuncAlloc()**, **MfuncStart()**, **MfuncFreeAndEnd()**, **MfuncParamChar()**, **MfuncParamDouble()**, **MfuncParamId()**, **MfuncParamLong()**, **MfuncParamPointer()**, **MfuncParamString()**

MfuncParamString

Synopsis Register a null terminated string parameter.

Format void MfuncParamString(FunctionId, ParamIndex, ParamValue)

MIL_ID FunctionId;	Function identifier
long ParamIndex;	Parameter index
void ParamValue;	Parameter value

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 function parameter list. The index of the first parameter is considered to be one.

The **ParamValue** parameter is the value of the string parameter.

See also MfuncAlloc(), MfuncStart(), MfuncFreeAndEnd(), MfuncParamChar(), MfuncParamDouble(), MfuncParamId(), MfuncParamLong(), MfuncParamPointer(), MfuncParamShort()

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()**

Appendix C: Troubleshooting

This appendix discusses error reporting, and suggests possible reasons for reported problems.

Error reporting

MIL has an error-reporting mechanism that is adaptable to your application development stage. MIL can report application errors to the screen or by using **MappGetError()**. During application development, it is probably best to have errors reported to the screen so that you can quickly debug the application. You control error reporting to the screen, using **MappControl()**; by default, error reporting to the screen is enabled.

In some circumstances, you might want your application to act on an error. You can do this by testing for the error and acting on it. For example, we recommend that it acts upon errors that occur during data buffer allocation. In this case, the application can inquire about the application error-code variable, using **MappGetError()**. You can also have your application act on errors by associating a function to them, using **MappHookFunction()**.

Did an error occur?

MappGetError() allows you to check for the success of the previous command call or that of a sequence of previous command calls. If this command returns an error code other than `M_NULL_ERROR`, you can use **MappGetError()** again to obtain a more detailed description of the error.

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.

Possible solutions

If the error messages do not provide sufficient information, you should refer to the next section for possible causes for the errors and suggested solutions. The error messages are in **alphabetical order**. Note, error messages for the more specific **blob analysis** and **pattern recognition** modules are explained in separate sections at the end of this appendix. If these suggestions don't work for you, and you cannot resolve the problem, see our website, www.matrox.com, or contact the Matrox Customer Support Group.

Error messages explained

☛ "Allocation error."

Error code: M_ALLOC_ERROR and M_ALLOC_ERROR_2

■ "Application already exists for this task."

You cannot allocate more than one MIL application in the same Host environment.

■ "Buffer type not supported."

You cannot allocate the buffer because the type (e.g. LUT) or depth (e.g. 16-bit) is not supported by the target platform.

■ "Cannot allocate system."

The application cannot allocate the requested system. This can occur if there is insufficient memory, or communication with the specified system cannot be established.

■ "Cannot allocate digitizer."

The application cannot allocate the requested digitizer. This can occur if there is insufficient memory, or the digitizer cannot be initialized as specified.

■ "Cannot allocate display."

The application cannot allocate the requested display. This can occur if there is insufficient memory, or the display cannot be initialized as specified.

■ "Cannot allocate temporary buffer in memory."

There is insufficient memory to allocate a required temporary buffer, or you have allocated the maximum number of buffers. Free all buffers that are no longer required.

■ "Not enough host memory to allocate buffer."

There is insufficient memory on the Host to allocate the specified buffer. Free Host memory or Host buffers that are no longer required.

- **"Not enough host memory to do operation."**

There is insufficient memory on the Host to perform the specified operation. Free Host memory or Host buffers that are no longer required.

- **"Not enough memory to allocate application."**

There is insufficient memory to allocate and start the MIL application.

- **"Not enough memory to allocate buffer."**

There is insufficient memory in the appropriate location to allocate the specified buffer, or you have allocated the maximum number of buffers. Free all buffers that are no longer required, or allocate large buffers before smaller ones.

☛ ***"Application free operation error."***

Error code: M_APP_FREE_ERROR

- **"Application still has system(s) associated to it."**

The application cannot be freed because it still has system(s) allocated. Free all systems allocated within the application, then free the application.

- **"Default host system still has buffer(s) associated with it"**

The Host system cannot be freed because it still has buffer(s) associated with it. Free all associated buffers, then free the Host system.

☛ ***"Buffer access error."***

Error code: M_ACCESS_ERROR

- **"Cannot export buffer."**

A MIL buffer cannot be exported to a file in the specified file format. Possible reasons for this error are:

- There might be insufficient Host RAM to allocate temporary work space.
- There might be insufficient disk space to save the buffer.
- Export of this type of MIL buffer might not be supported.

■ **"Cannot import buffer."**

A file cannot be imported into a MIL buffer for the same reasons indicated in the "Cannot export buffer" error, or due to one of the following:

- The specified file might not be a valid MIL file.
- The specified file might be corrupted.

■ **"Cannot M_RESTORE a M_RAW file format buffer."**

You are trying to restore a file that was stored in M_RAW format. A file stored in raw format does not have any header data identifying its buffer parameters. Therefore, to restore this data file, you must allocate an appropriate MIL buffer, then import the data, using M_LOAD.

■ **"Cannot restore buffer."**

A restore operation cannot be successfully completed for the same reasons indicated in the "Cannot import buffer" error.

■ **"Source buffer must be an M_IMAGE buffer."**

The buffer does not have the expected M_IMAGE attribute.

➡ ***"Buffer free operation error."***

Error code: M_BUFFER_FREE_ERROR

■ **"Buffer still has child(ren) associated to it."**

A MIL buffer cannot be freed because it still has child buffer(s) associated to it. Free all associated child buffers, then free the parent buffer.

■ **"Use MnatBufDestroy() on this kind of buffer."**

You are trying to use a standard MIL command to destroy a buffer allocated in native mode with **MnatBufCreate...()**.

☛ *"Call context error"*

Error code: M_CALL_CONTEXT_ERROR

■ **"Cannot allocate temporary buffer in memory."**

There is insufficient Host memory to allocate the temporary buffer required for the operation, or you have allocated the maximum number of buffers. Free Host memory or Host buffers that are no longer required.

☛ *"Child allocation error."*

Error code: M_CHILD_ERROR

■ **"Cannot allocate temporary child buffer in memory."**

There is insufficient memory to allocate a temporary child buffer required for the operation or you have allocated the maximum number of buffers. Free all buffers that are no longer required.

■ **"Not enough memory to allocate child buffer."**

There is insufficient memory left to allocate the specified buffer. Free all buffers that are no longer required.

☛ *"Digitizer error."*

Error code: M_DIGITIZER_ERROR

■ **"Digitizer and buffer must belong to same system."**

The grab buffer is not allocated on the same system as the digitizer. Allocate them on the same system.

■ **"Digitizer LUT dimensions are not compatible with the user LUT."**

The LUT buffer does not have the required number of entries to map the range of possible image buffer pixel values. Ensure that the LUT buffer and the digitizer LUTs have the same number of entries and color bands.

☛ *"Display error."*

Error code: M_DISPLAY_ERROR

■ **"Buffer not currently selected on display."**

You cannot de-select a buffer that is not currently selected on the display.

- **"Cannot associate a M_PSEUDO LUT with a monochrome display."**

The target platform does not support a pseudo-color LUT with a monochrome display.

- **"Display and buffer must belong to same system."**

The buffer to display is not allocated on the same system as the display. Allocate them on the same system.

- **"Display LUT dimensions are not compatible with the user LUT."**

The LUT buffer does not have the required number of entries to map the range of possible image buffer pixel values. Ensure that the LUT buffer and the target display output LUTs have the same number of entries and color bands.

- **"Zoom factors must be between -16 and 16 inclusive (except 0)."**

You cannot zoom with a factor outside the accepted range.

☞ ***"File access error."***

Error code: M_FILE_ERROR

- **"Cannot open input file."**

The file cannot be found or access is denied.

- **"Cannot open output file."**

The file cannot be found or access is denied.

- **"Cannot read file."**

This can occur if the specified file is read-protected or a disk-access error has occurred.

- **"Cannot write to file."**

Write-access is denied or a disk-access error has occurred.

- **"Not a MIL file."**

The specified file does not have a MIL file format.

☛ **"Function start error."**

Error code: M_FUNCTION_START_ERROR

■ **"No application allocated yet, allocate one."**

A function is called prior to a MIL application allocation. Use **MappAlloc()** or **MappAllocDefault()** to allocate the application before performing any other MIL operation.

☛ **"Inappropriate MIL ID."**

Error code: M_INVALID_NATURE

The specified MIL object does not have the appropriate attributes for the operation. For example, it might occur when an operation expects an image buffer identifier and it is given LUT buffer identifier instead.

"Invalid parameter *n*."

The *n*th parameter is not valid.

☛ **"Invalid attributes."**

Errorcode: M_INVALID_ATTRIBUTE

■ **"Invalid parameter *n*."**

The *n*th parameter does not have the appropriate attribute(s).

☛ **Invalid MIL ID."**

Error code: M_INVALID_ID

The specified system, digitizer, display, or buffer identifier is not valid. That is, its corresponding object was not successfully allocated before you tried to use it. If you have performed the object allocation, check to make sure that it was successful.

■ **"Invalid parameter *n*."**

The *n*th parameter is not valid.

☛ **"Invalid parameter."**

Error code: M_INVALID_PARAM_ERROR, M_INVALID_PARAM_ERROR_2 and M_INVALID_PARAM_ERROR_3

■ **"Bad parameter value."**

A parameter is set to an invalid value. Check that the given value is within the parameter's range.

■ **"For this operation the grab mode must be asynchronous."**

You cannot do an asynchronous operation when the grab mode setting is synchronous (see **MdigControl()**).

■ **"For this operation, you should supply a LUT buffer with at least 512 entries."**

Your LUT buffer has an insufficient number of entries for the target operation.

■ **"No graphic text fonts selected."**

The specified graphics context does not specify a font to use to draw text.

■ **"One parameter does not reside within the buffer's limits."**

A specified parameter exceeds the target buffer's limits. This is typically caused when you try to allocate a child partially outside its parent.

■ **"Param n not in supported list."**

The specified n^{th} parameter value is not one of the supported values for that parameter.

■ **"Pointer must be non Null."**

An M_NULL pointer is passed to a function that needs to return more than one element.

■ **"Scale factors out of supported range."**

The specified scale factor is outside the supported range of the target system, or no scaling is supported.

■ **"Specified center is outside buffer."**

You cannot specify a center of rotation that is outside the specified buffer coordinates.

- **"This type of conversion requires two 3 band buffers."**

"This type of conversion requires a 3-band source buffer."

"This type of conversion requires a 3-band destination buffer."

You cannot perform a conversion between buffers that do not have the appropriate number of bands.

☛ ***"MIL driver obsolete."***

- **"Version must be (*version #*) or higher."**

Your MIL driver is older than the specified version and is not supported by the current version of the library.

☛ ***"MIL file access error."***

Error code: M_MIL_FILE_ERROR, M_MIL_FILE_ERROR_2 and M_MIL_FILE_ERROR_3.

- **"Bad file format detected."**

- **Check the file to ensure it is not corrupted.**

- **"Cannot allocate temporary buffer in memory."**

There is insufficient memory to allocate the temporary buffer required for the operation or you have allocated the maximum number of buffers. Free all buffers that are no longer required.

- **"Cannot close file."**

A disk-access error has occurred.

- **"Cannot open file."**

The file is not found or access is denied.

- **"Cannot read file."**

The specified file is read protected or a disk-access error has occurred.

- **"Cannot seek in file."**

MIL cannot read the specified file.

- **"Cannot write to file."**

Write-access is denied or a disk-access error has occurred.

- **"Not a MIL file."**

A MIL file format is anticipated but not found.

- **"Only 8, 16 or 32 BitsPerSample supported."**

The file bit size setting is not 8, 16, or 32 bits/sample.

- **"Only compression type 1 supported."**

The target file is compressed and MIL cannot read it.

- **"Only identical BitsPerSample for every sample supported."**

The bits/sample are not identical for every sample in the TIFF file.

- **"Only PlanarConfiguration 2 supported for multi-band images."**

The PlanarConfiguration parameter is not equal to 2 in the TIFF file. MIL only supports planar mode for color images.

- **"PhotometricInterp incompatible with SamplePerPixel."**

The photometric interpolation setting of the file is incompatible with the sample/pixel supported by MIL (type 1 for monochrome buffers and type 2 for multi-band buffers). This occurs when the TIFF file contains a palletized image, since only grayscale or true color image formats are supported.

- **"The image file does not conform to the TIFF 6.0 specification."**

The image file has been created according to an older, unsupported, TIFF specification.

- **"Up to 8 Samples Per Pixel supported."**

The samples/pixel is greater than 8 in the TIFF file.

- **"Wrong Byte Order, Only INTEL Byte Ordering supported."**

The file has the wrong byte ordering. Only INTEL byte ordering is supported by the MIL TIFF handler.

☛ ***"Processing error."***

Error code: M_PROCESSING_ERROR

- **"All buffers do not have the same working system."**

"Cannot find any working system between buffers."

"Cannot process a HOST buffer as a whole and a temporary buffer."

The location of the specified buffers is not valid for the requested operation.

☛ ***"System command error."***

Error code: M_COMMAND_DECODER_ERROR

- **"Operation execution failed."**

The target system cannot execute the requested operation.

- **"Requested operation not supported."**

The target system does not support the requested operation.

☛ ***"System free operation error."***

Error code: M_SYSTEM_FREE_ERROR

- **"Cannot allocate temporary buffer in memory."**

There is insufficient memory to allocate the temporary buffer required for the operation or you have allocated the maximum number of buffers. Free all buffers that are no longer required.

- **"System still has buffer(s) associated to it."**

You cannot free a system that still has buffer(s) allocated on it. Free those buffers, then free the system.

- **"System still has digitizer(s) associated to it."**

You cannot free a system that still has digitizer(s) allocated on it. Free those digitizers, then free the system.

- **"System still has display(s) associated to it."**

You cannot free a system that still has display(s) allocated on it. Free those displays, then free the system.

☞ ***"TIFF file access error."***

Error code: M_TIFF_ERROR and M_TIFF_ERROR_2

- **"Cannot allocate temporary buffer in memory."**

There is insufficient memory to allocate the temporary buffer required for the operation or you have allocated the maximum number of buffers. Free all buffers that are no longer required.

- **"Cannot close file."**

A disk-access error has occurred.

- **"Cannot open file."**

The TIFF file is not found or access is denied.

- **"Cannot read file."**

This can occur if the specified TIFF file is read protected or a disk-access error has occurred.

- **"Cannot write to file."**

Write-access to the specified TIFF is denied or a disk-access error has occurred.

- **"Not a TIFF file."**

The specified file is not detected as a TIFF file. This can occur if the file is of the wrong type or if it is corrupted.

- **"Only 8, 16 or 32 BitsPerSample supported."**

The file bit size setting is not 8, 16, or 32 bits/sample.

- **"Only compression type 1 supported."**

The target TIFF file is compressed and MIL TIFF handler cannot read it.

- **"Only identical BitsPerSample for every sample supported."**

The bits/sample ratio is not identical for every sample in the TIFF file.

- **"Only PlanarConfiguration 2 supported for multi-band images."**

The PlanarConfiguration parameter is not equal to 2 in the TIFF file. MIL only supports planar mode for color images.

- **"PhotometricInterp incompatible with SamplePerPixel."**

The photometric interpolation setting of the file is incompatible with the sample/pixel supported by MIL (type 1 for monochrome buffers and type 2 for multi-band buffers). This occurs when the TIFF file contains a palletized image, since only grayscale or true color image formats are supported.

- **"The image file does not conform to the TIFF 6.0 specification."**

The image file has been created according to an older, unsupported, TIFF specification, or it is incomplete.

- **"Up to 8 Samples Per Pixel supported."**

The samples/pixel ratio is greater than 8 in the TIFF file.

- **"Wrong Byte order, only INTEL Byte Ordering supported."**

The file has the wrong byte ordering. Only INTEL byte ordering is supported by the MIL TIFF handler.

Driver error messages explained

☞ ***"Asynchronous grab mode not supported."***

Error code: M_ERROR_SYSTEM_START_CODE + 76L

The target system does not support asynchronous grab mode.

☞ ***"Board initialization failed."***

Error code: M_ERROR_SYSTEM_START_CODE + 11L

Communication with the specified board cannot be established or initialization is impossible.

☞ ***"Board selection failed."***

Error code: M_ERROR_SYSTEM_START_CODE + 12L

The specified board cannot be selected as the target of the requested operation. Communication is impossible or broken.

☞ ***"Buffer(s) still allocated on that system."***

Error code: M_ERROR_SYSTEM_START_CODE + 1L

You cannot free a system without freeing all currently allocated system buffers.

☞ ***"Buffer type not supported."***

Error code: M_ERROR_SYSTEM_START_CODE + 22L

The selected buffer type is not supported by the target system.

☞ ***"Can not allocate digitizer."***

Error code: M_ERROR_SYSTEM_START_CODE + 56L

The digitizer cannot be initialized as specified.

☞ ***"Can not allocate display."***

Error code: M_ERROR_SYSTEM_START_CODE + 65L

The display cannot be initialized as specified.

☞ ***"Can not allocate LUT buffer."***

Error code: M_ERROR_SYSTEM_START_CODE + 55L

The target system does not support LUT or allocation of a custom LUT.

☛ ***"Can not allocate temporary buffer."***

Error code: M_ERROR_SYSTEM_START_CODE + 85L

Insufficient memory available on the system boards or on the Host. Free any buffers that are not in use.

☛ ***"Can not display buffer."***

Error code: M_ERROR_SYSTEM_START_CODE + 63L

The buffer is allocated for display or the system does not support the display of that type of buffer.

☛ ***"Can not grab buffer."***

Error code: M_ERROR_SYSTEM_START_CODE + 70L

The buffer is not allocated as a grab buffer or the system does not support the grab.

☛ ***"Can not open specified .DCF file."***

Error code: M_ERROR_SYSTEM_START_CODE + 29L

The display configuration file is not found or access is denied.

☛ ***"Can not undisplay buffer."***

Error code: M_ERROR_SYSTEM_START_CODE + 66L

You cannot de-select a buffer that is not selected on the display.

☛ ***"Can not update display."***

Error code: M_ERROR_SYSTEM_START_CODE + 78L

This can occur when trying to update the display with new data. Access to the display may be impossible.

☛ ***"Can not update displayed buffer."***

Error code: M_ERROR_SYSTEM_START_CODE + 64L

Access to the displayed buffer might be impossible.

☛ ***"Character font not supported on system"***

Error code: M_ERROR_SYSTEM_START_CODE + 27L

The selected character font is not supported on the target system.

☞ ***"Continuous grab must be halted before next operation."***

Error code: M_ERROR_SYSTEM_START_CODE + 14L

You cannot execute the requested command while performing a continuous grab on the target system. Halt the grab before issuing another command.

☞ ***"Data format name or file name not found."***

Error code: M_ERROR_SYSTEM_START_CODE + 56L

Verify the selected data format name and file name.

☞ ***"Device(s) still allocated on that driver."***

Error code: M_ERROR_SYSTEM_START_CODE + 9L

You cannot free a system without freeing all of its devices (digitizer or display).

☞ ***"Digitizer channel not supported."***

Error code: M_ERROR_SYSTEM_START_CODE + 21L

The selected digitizer channel is not supported.

☞ ***"Digitizer configuration error."***

Error code: M_ERROR_SYSTEM_START_CODE + 13L

The digitizer cannot be initialized as specified.

☞ ***"Digitizer format not supported."***

Error code: M_ERROR_SYSTEM_START_CODE + 25L

The selected digitizer format is not supported.

☞ ***"Digitizer(s) still allocated on that system."***

Error code: M_ERROR_SYSTEM_START_CODE + 5L

You cannot free a system without freeing all of its digitizers.

☞ ***"Display configuration error."***

Error code: M_ERROR_SYSTEM_START_CODE + 18L

The display cannot be initialized as specified.

☛ ***Display(s) still allocated on that system."***

Error code: M_ERROR_SYSTEM_START_CODE + 6L

You cannot free a system without freeing all of its displays.

☛ ***"Error changing channel."***

Error code: M_ERROR_SYSTEM_START_CODE + 71L

The selected channel is invalid or there is no digitizer connected to that channel.

☛ ***"Error changing reference."***

Error code: M_ERROR_SYSTEM_START_CODE + 72L

Verify whether the system supports reference level changes.

☛ ***"Error setting LUT."***

Error code: M_ERROR_SYSTEM_START_CODE + 67L

This might be due to insufficient memory to perform the operation.

☛ ***"Incompatible buffer memory organization."***

Error code: M_ERROR_SYSTEM_START_CODE + 10L

The memory organization of the buffers used in the requested operation are not compatible with each other on the target system. This is determined by your hardware.

☛ ***"Input device not responding."***

Error code: M_ERROR_SYSTEM_START_CODE + 16L

The digitizer is not sending data to the system. Ensure that the digitizer is properly connected to the system.

The following errors occur when the requested item is inappropriate or outside the supported range for your system. Verify your system's restrictions on the specified item.

☛ ***"Invalid board number."***

Error code: M_ERROR_SYSTEM_START_CODE + 50L

☛ ***"Invalid digitizer channel."***

Error code: M_ERROR_SYSTEM_START_CODE + 15L

☛ ***"Invalid digitizer number."***

Error code: M_ERROR_SYSTEM_START_CODE + 80L

- ➡ ***"Invalid display number."***
 Error code: M_ERROR_SYSTEM_START_CODE + 79L
 - ➡ ***"Invalid horizontal scaling factor."***
 Error code: M_ERROR_SYSTEM_START_CODE + 74L
 - ➡ ***"Invalid initialization flag."***
 Error code: M_ERROR_SYSTEM_START_CODE + 28L
 - ➡ ***"Invalid number of fields."***
 Error code: M_ERROR_SYSTEM_START_CODE + 82L
 - ➡ ***"Invalid scaling factor."***
 Error code: M_ERROR_SYSTEM_START_CODE + 69L
 - ➡ ***"Invalid system number."***
 Error code: M_ERROR_SYSTEM_START_CODE + 51L
 - ➡ ***"Invalid vertical scaling factor."***
 Error code: M_ERROR_SYSTEM_START_CODE + 75L
 - ➡ ***"LUT is more than 256 elements."***
 Error code: M_ERROR_SYSTEM_START_CODE + 54L
 - ➡ ***"LUT not supported."***
 Error code: M_ERROR_SYSTEM_START_CODE + 73L
- end of group -----

☛ ***"Not enough host memory"***

Error code: M_ERROR_SYSTEM_START_CODE + 2L

There is insufficient host memory available. Free all unused buffers residing on the Host.

☛ ***"Not enough memory to allocate buffer."***

Error code: M_ERROR_SYSTEM_START_CODE + 23L

There is insufficient memory to allocate the specified buffer on the target system. Free all unused buffers.

☛ ***"Pan factor not supported."***

Error code: M_ERROR_SYSTEM_START_CODE + 19L

The requested pan factor is outside of the supported range.

☛ ***"Parameter to inquire not supported."***

Error code: M_ERROR_SYSTEM_START_CODE + 20L

Inquiry of this parameter is not supported or the type of inquiry is invalid.

☛ ***"Synchronous grab mode not supported."***

Error code: M_ERROR_SYSTEM_START_CODE + 77L

The target digitizer does not support synchronous grab. You must use asynchronous mode.

☛ ***"Too many buffers allocated on that system."***

Error code: M_ERR_SYSTEM_START_CODE + 3L

You have exceeded the number of allowable system buffers. Free unused buffers.

☛ ***"Too many digitizers allocated on that system."***

Error code: M_ERROR_SYSTEM_START_CODE + 7L

You have exceeded the number of allowable system digitizers.

☛ ***"Too many display allocated on that system."***

Error code: M_ERROR_SYSTEM_START_CODE + 8L

You have exceeded the number of allowable system displays.

☞ ***"Too many systems of that type allocated"***

Error code: m_ERROR_SYSTEM_START_CODE + 4L

You have exceeded the number of allowable systems of the specified type.

☞ ***"Zoom factor not supported."***

Error code: M_ERROR_SYSTEM_START_CODE + 17L

The requested zoom factor is outside the supported range.

Index

A

AC Huffman table 139

acquisition

- attribute 39, 209, 212, 216, 244, 247
- continuous 33, 308
- image 32, 104, 307
- input LUT 129
- precondition 114

address

- Host 55
- logical 55

allocate

- application 20, 183
- buffers 29, 169
- child buffer 42, 221, 223, 225–226
- data buffer 36
- defaults 21, 168, 185
- digitizer 32, 104, 169, 296
- display 66, 169, 320
- graphics context 98, 352
- image buffer 27, 39
- LUT buffer 61–62
- multi-band buffer 128, 216
- one-dimensional buffer 209
- pseudo-MIL function 399
- pseudo-mil identifier 400
- system 167, 372
- thread 190
- two-dimensional buffer 212

allocation error 41

analog reference levels 318

annotation

- image 98
- non destructive 77

application

- allocate 183
- building 20
- child 148
- control environment 188
- control module 171
- free 194
- inquire environment 205
- pseudo-MIL, parameter checking flag 413

- pseudo-MIL, parameter registering flag 419
- simultaneous processing 146
- starting 167

application context 148

architecture, display 322

arcs, draw 100, 354

arcs, draw filled 355

attributes, data buffer usage 209, 212,
216, 244, 247

avi files 252, 256, 273

B

background color

- associate to graphics context 99, 356
- inquire 366

Binary buffers 49

binary buffers, packed 38

blanking, display 71, 331

brightness, adjust on input 113

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, 104
- adjusting/focusing 33
- sophisticated 104
- specification 105, 296

character parameter

- pseudo-MIL 412

characters, text 102, 177, 371

child application 148

child buffers 42

- allocate 42, 221, 223, 225–226
- ancestor buffer 277
- attributes 221, 223, 225–226

- data buffer attributes 42
- definition 36
- dimensions 42
- display 43, 75
- display multiple 72
- inheriting parent features 42
- LUT 61
- multiple dimensions 221, 223, 226
- offset 221, 223, 225–226
- offset from parent 42
- parent buffer 277
- physical space 221, 223, 225–226
- purpose 169
- returned coordinates 42, 221, 223, 225–226
- size 42
- type 221, 223, 225–226
- circles, draw 100, 354
- clear
 - buffer 228, 357
 - display 71, 331
 - graphics image buffer 99
- clipping
 - borders 43
 - data 234
 - graphics 101
- color
 - handling techniques 127
 - input LUT 129
- color band 37, 216, 244, 247
 - LUT 87
- color images
 - allocate buffer 128, 216
 - allocate child buffer 221, 223
 - copy 236, 238
 - copy single band 43
 - create buffer 247
 - dealing with 128
 - displaying 130, 346–347
 - grabbing 128, 307–308
 - loading 131, 282, 295
 - put data in band 44
 - reference levels 114
 - saving 131, 282, 295
- command reference
 - order 180
 - quick reference 171
 - status section 180
- commands
 - Function DevelopersToolkit' 397
 - functions 21
 - MIL, command summary 171
 - predefined constants 180
 - pseudo-MIL 158, 394
- communication channels 20, 23, 167
- compensation
 - memory 189
 - processing 188
- compiling 22
- compressing images 136
- conditional buffer, creating 98
- continuous grab 33, 308
- contrast 113
 - image, adjusting 113
- control
 - application environment settings 188
 - areas processed 42
 - buffer features 229
 - digitizer 299
 - display 325
 - graphic context 359
 - messages, error 188, 190
 - parameter checking 188
 - processing compensation 188
 - system processing 374
 - thread 190
 - timer 208
 - trace mechanism 188
- conversion
 - data format 45, 253, 269
- coordinates
 - child buffer 42
 - of a pixel 58
 - text writing 102
- copy
 - bit truncation/extension 44
 - clip, and 43, 234
 - color band 43, 62, 236, 238
 - conditional 43, 241
 - data 43, 233, 236, 238, 241, 259–260, 262, 265, 283–284, 286, 291–292
 - data line 289
 - data to LUT 62
 - mask 43, 243
 - specific buffer areas 43

- Corona
 - exposure 120, 123–125
 - automatic model 122
 - bypass model 124
 - triggers 120, 123–125
- custom
 - window, VGA 81

D

- data allocation and access module 35, 172

- data buffer

- allocation 209, 212, 216, 244, 247
 - ancestor 277
 - attributes 38, 41, 209, 212, 216, 244, 247
 - automatically allocated 45
 - child 36, 42
 - clear 99, 228, 357
 - clip border 43, 234
 - color band 37, 128, 216, 244, 247
 - control 229
 - copy 233
 - copy color 236, 238
 - copy theoretical line 265
 - defined 36, 169
 - depth 38, 209, 212, 216, 244, 247
 - dimensions 37
 - display 65
 - export data 45, 253
 - free 37, 258
 - get data, put in array 44
 - handling 35
 - import 45
 - import data 269
 - incorrect usage 41
 - inquire 251, 276
 - integer 38
 - intended usage 39
 - load 253
 - location 38
 - LUT, see LUT buffer 61
 - management 44
 - multiple dimensions 216, 247
 - multiple, display 43
 - multiple, handling 43
 - packed binary 38
 - parent 277
 - pseudo-MIL, modification 411

- put data 44, 283–284, 286, 291–292
 - range 38, 209, 212, 216, 244, 247
 - restore 45, 293
 - retrieve data 259–260, 262, 267–268
 - save 45, 253, 295
 - sign 209, 212, 216, 244, 247
 - two-dimensional 244
 - type 38, 216, 244, 247
 - write data 289

- data format, input device 105, 296

- data generation

- LUT 61, 351
 - module 176

- data objects, manipulation concepts 168

- data type 29

- data, overwriting 29

- DC Huffman table 139

- dcf files 105

- debugging 170

- decompressing images 136

- default graphics context 98

- defaults

- application 185

- display 387

- free 195

- image buffer 18, 21, 29, 31, 388

- initialization flag 387

- initializing 18

- input device 104, 388

- input LUT 115

- setup 386

- system 387

- depth

- data buffer 38

- display 70

- destination buffer 29

- device

- control module 103

- digitizer

- allocate 32, 104, 296

- color data format 128

- configuration format 105

- control 299

- data format 296

- event hook 311

- free 104, 306

- input channel 106, 298

- inquire 105, 313

- LUT 64, 114, 129, 317
 - number 106, 296
 - reference levels 113, 318
- digitizer configuration files 105
- DirectDraw 88, 91
- DirectDraw underlay-surface 91
- DirectDraw underlay-surface display 91
- DirectDraw underlay-surface display architecture 91, 322
- display 88
 - allocation 20, 320
 - annotating 76
 - Windows GDI 78
 - architectures 322
 - border handling 66
 - buffer 29, 43
 - clear 71
 - color 86
 - color image 130
 - control behavior 325
 - control module 65, 175
 - dual-screen 67
 - example 77, 82
 - format 320
 - free 82, 332
 - image buffer 346–347
 - image location 66
 - inquire 335
 - LUT 63, 86, 130, 340
 - mode
 - non-windowed 69
 - windowed 69
 - monochromatic effect 63
 - monochrome buffer 28
 - multi-head 68
 - multiple buffers 72
 - non 8-bit buffers 70
 - number 320
 - pan 75, 345
 - pseudo-color effect 63
 - psuedo color LUT 87
 - scroll 75, 345
 - single-screen 67
 - size and depth 70
 - true color effect 64
 - user-defined window 81
 - VGA 67
 - VGA system 81

- Windows, VGA 81
 - zoom 75, 348
- display architecture 322
- display architectures 88
- display mode
 - windowed 92
- Displayable buffers 39
- dots, draw 100, 361
- double buffering, definition 116
- DrawDIBDraw()
 - VGA 92–94, 322, 324
- drawing 98, 100
- dual-screen configuration
 - displaying in 67
 - VGA 67
- dynamic range 113

E

- edge
 - rising/falling 302–303
- error reporting
 - appendix 424
 - automatic 31
 - code 196
 - hook 202, 333
 - memory, insufficient 41
 - message control 21, 188, 190
 - messages 170, 196, 424
 - pseudo-MIL function 401, 405
 - screen 169, 424
 - suberror code 196
 - thread 148–149
 - use 169
- examples
 - color, run with 129
 - display in user-defined window 82
 - display multiple buffers 72
 - display with overlay 77
 - Function DeveloppersToolkit' 394
 - general information 18, 23
 - grab 32
 - image allocation/display 30
 - installing 18
 - MIL sample program 23
 - mmultdis.c 72
 - modify for color 28
 - mstart.c 23

- mwindisp.c 82
- Native Mode ProgrammersToolkit' 159
- pseudo-function development 394
- standard defaults 28
- zooming 75
- export data buffer 45, 253
- exposure 125
 - automatic model
 - Corona 122
 - bypass model
 - Corona 124
 - Corona 120

F

- field grabbing 107, 302
- file format 45, 253
- files
 - avi 252, 256, 273
- filled-in shapes 101, 355
 - boundary-type seed fill 362
- font
 - associate to graphics context 102, 363
 - inquire 366
 - predefined 102
 - scale 102, 364
 - size 102, 364
- foreground color
 - associate to graphics context 99, 358
 - fill with 101
 - inquire 366
- frame
 - grabbing 33, 107
- frame buffer 66
- frame buffer, display 77
- free
 - application 194
 - application defaults 195
 - buffer, data 37, 258
 - defaults 168, 195
 - digitizer 306
 - display 332
 - graphics context 98, 177, 365
 - pseudo-MIL function 403
 - pseudo-MIL identifier 404
 - system 378

- function
 - development 159, 394
 - execution success 21
 - hook 170, 202, 311, 333
 - pseudo-MIL, allocate 399
 - pseudo-MIL, example 394
 - pseudo-MIL, free 403
 - pseudo-MIL, start 422
 - user-created 15
- Function Developers Toolkit 393
- Function DevelopersToolkit
 - command summary 397
 - example 394
- functions
 - commands 21
- functions See also, commands 180

G

- gamma correction 64
- Genesis
 - system 373
- global library state 386
- Grab 39
- grab 115
 - color images 128
 - continuous 33, 308
 - data buffer 39, 128
 - example 32
 - fields 107
 - frames 33, 107
 - halt 33, 310
 - image 32, 104, 307
 - mode 116
 - monochrome 32
 - multi-dimensional buffers 128
 - scale 299
 - synchronization 116
 - wait 309
- Grab buffers 39
- GrabAndWarp()
 - example 159
- graphics 98
 - arcs, draw 100, 354
 - boundary type seed fill 101, 362
 - buffer, clear 99, 357
 - capabilities 14
 - circles, draw 100, 354

- clipping 101
- dots, draw 100, 361
- filled elliptic arcs, draw 100, 355
- filled rectangles, draw 100, 177, 370
- filled-in shapes 100
- lines, draw 100, 177, 368
- module 97
- non-destructive annotation 77
- outline, draw 100
- parameters 99
- rectangles, draw 100, 369
- text, write 102, 177, 371
- graphics context
 - allocate 98, 352
 - background color, associate 99, 356
 - control 359
 - default 98, 353
 - definition 98
 - font scale, associate 102, 364
 - foreground color, associate 99, 358
 - free 98, 177, 365
 - inquire 366
 - object parameters 99
 - text font, associate 102, 363

H

- halt grabbing 33, 310
- header file 22
- hook
 - digitizer event 311
 - error 202, 333
 - get information 199
 - to an event 202
 - trace 202, 333
 - user-defined function 170
- Host
 - communication 167
 - screen 169
- host
 - communication 20
 - CPU 14
 - default system 37, 183, 186, 209, 212, 216, 271, 352, 372
 - screen 21
 - system 21, 62, 351, 372

- hue 224, 236, 239, 261, 263, 285, 287
 - HLS 221
- Huffman encoding 138–139

I

- identifier, MIL objects 180
- image
 - grabbing 32
- image buffer
 - acquisition 39
 - allocation 27, 29–30, 38–39
 - clear 228, 357
 - color 28, 33
 - conditional 98
 - default 21, 29, 129
 - defined 29
 - destination 29
 - display 30, 39
 - display border 66
 - display multiple 72
 - display position 66
 - free 37
 - map through LUT 60
 - removing from display 71, 331
 - select for display 66, 346
 - select window for display 347
 - size 29
 - source 29
 - two-dimensional 28–29, 33
 - uses 29
- import data 269
- include file 22
- initialization
 - default 168, 185
 - input device 104
 - system 18, 104, 167
- input device
 - brightness 113
 - contrast 113
 - control module 174
 - defaults 104
 - frequency 105
 - line-scan 107
 - LUT 114
 - reference level 316

- resolution 105
- subsampling 115
- using 32
- input signal 300
- inquire
 - application environment 205
 - data buffer 251, 276
 - digitizer 105, 313
 - display 335
 - graphics context 366
 - system 379
- installation
 - MIL 18
 - test program 22
- integer buffers 38
- Intellicam 105
- intensity
 - correction 62
- interlaced JPEG compression 136

J

- JPEG compression 136

K

- kernels
 - buffer allocation 39
- keying 77
 - inquire 335

L

- lines
 - draw 100, 177, 368
- line-scan device 107
- link program with library 22
- load
 - color image 131
 - data 44–45, 282
 - LUT data 62
- look-up table 85
 - 1-band custom 87
 - 3-band custom 88
 - changing default 86
 - control loading into physical output LUTs 94

- control loading into Windows palette 92, 94
- pseudo-color 87
- lossless compression 136
- lossy compression 136
- luminance
 - HLS 221, 224, 236, 239, 261, 263, 285, 287
- LUT 85
 - 1-band custom 87
 - 3-band custom 88
 - changing default 86
 - control loading into physical output LUTs 94
 - control loading into Windows palette 92, 94
 - pseudo-color 87
- LUT buffer
 - allocation 38, 61
 - child buffer 61
 - color bands 61–62, 129
 - data generation 61–62
 - dimensions 61
 - load 62
 - management 61
 - one-dimensional 61
 - restore 62
- LUTs
 - custom 87
 - data generation 349, 351
 - definition 60
 - display 63, 130, 340
 - display color, change 86
 - general information 59
 - index 61
 - input 113–114, 129, 317
 - input mapping 64
 - intensity correction 62
 - monochromatic effect 63
 - multiple-color-band 88
 - one-color-band 87
 - pseudo-color effect 63
 - ramp 61
 - true-color effect 64
 - usage 63

M

- M_DISP 39
- M_GRAB 39
- macros 181
- Mapp...() 171, 183
- MappAlloc() 21, 98, 148, 167, 183
 - example 82
- MappAllocDefault() 18, 20, 29, 31–32, 66, 98, 104, 128–130, 168, 386
 - example 22, 30, 32–33, 72, 159, 394
- MappChild() 148
- MappControl() 21, 169–170, 424
- MappControlThread() 149
- MappFree() 21, 167
 - example 82
- MappFreeDefault() 20, 168
 - example 23, 30, 32–33, 72, 159, 394
- MappGetError() 21, 149, 170, 424
 - example 30
- MappHookFunction() 21, 170, 424
- MappModify() 207
- mask, copy 43, 243
- Mbuf...() 172
- MbufAlloc() 54
- MbufAlloc1d() 36, 61, 87, 169
 - example 394
- MbufAlloc2d() 29, 36, 72, 169
 - example 30, 82
- MbufAllocColor() 21, 36, 46, 61, 128, 169
- MbufChild1d() 42
- MbufChild2d() 42, 72
 - example 72
- MbufChildColor() 42, 138
- MbufClear() 99
 - example 72, 82
- MbufControl 78
- MbufControl() 78, 87, 137–138, 140
 - example 159
- MbufCopy() 62, 137
- MbufCopyClip() 43
- MbufCopyColor() 43, 62
- MbufCopyCond() 43
- MbufCopyMask() 43
- MbufCreate2d() 55
- MbufCreateColor() 55
- MbufExport() 45, 131, 137
- MbufExportSequence 136
- MbufExportSequence() 256
- MbufFree() 21, 37, 42, 61
 - example 72, 394
- MbufGet() 44, 54
 - example 394
- MbufGet1d() 44
- MbufGetColor() 44
- MbufImport() 45, 131, 137
- MbufImportSequence() 136, 273
- MbufInquire() 55–56, 78
 - example 159, 394
- MbufLoad() 45, 62, 131
 - example 72, 394
- MbufPut() 44, 54, 62, 87
 - example 394
- MbufPut1d() 44, 62
- MbufPutColor() 44, 62
- MbufRestore() 45, 62, 131
- MbufSave() 45, 131
- Mdig...() 174
- MdigAlloc() 21, 32, 104–106, 128, 296
 - example 82
- MdigChannel() 106, 298
- MdigControl() 33, 116, 147, 299
- MdigFree() 21, 104, 128, 169, 306
 - example 82
- MdigGrab() 33, 107, 116, 128, 137, 307
 - example 32
- MdigGrabContinuous() 33, 128, 308
 - example 33, 82
- MdigGrabWait() 116, 309
- MdigHalt() 33, 116, 310
 - example 33, 82
- MdigHookFunction() 147, 311
- MdigInquire() 105, 313
 - example 159
- MdigLut() 64, 114, 129, 317
- MdigReference() 114, 129, 318
- Mdisp...() 175
- MdispAlloc() 21, 29, 66–67, 77, 130, 320
 - example 82
- MdispControl() 70, 77, 325
 - example 77
- MdispDeselect() 29–30, 71, 130, 331
 - example 72, 82
- MdispFree() 21, 71, 169, 332
 - example 82
- MdispHookFunction() 79, 333

- MdispInquire() 79, 87, 335
 - example 77
- MdispLut() 63, 87, 130, 340
- MdispOverlayKey() 77, 343
- MdispPan() 75, 345
- MdispSelect() 29, 39, 43, 66, 72, 130, 346
 - example 72
 - VGA 81
- MdispSelectWindow() 81, 347
 - example 82
- MdispZoom() 75, 348
 - example 72
- memory
 - compensation 189
 - insufficient 41
 - resources 20–21
- messages, error 21, 31
- Meteor
 - system 372
- MfuncAlloc() 399, 412, 414–415, 417–418, 420
 - example 394
- MfuncAllocId() 400, 407, 409–410
- MfuncErrorReport() 401
 - example 394
- MfuncFreeAndEnd() 401, 403
 - example 394
- MfuncFreeId() 404
- MfuncGetError() 405
- MfuncIdGetObjectType() 407
- MfuncIdGetUserPtr() 408
- MfuncIdSetObjectType() 409
- MfuncIdSetUserPtr() 410
- MfuncModified() 411
- MfuncParamChar() 412
 - example 394
- MfuncParamCheck() 413
 - example 394
- MfuncParamDouble() 414
- MfuncParamId() 415, 422
 - example 394
- MfuncParamLong() 417
- MfuncParamPointer() 418
- MfuncParamRegister() 419
- MfuncParamShort() 420
- MfuncParamString() 421
- MfuncStart() 399, 403, 412, 414–415, 417–418, 420–422
 - example 394
- Mgen...() 176
- MgenLutFunction() 61–62, 349
- MgenLutRamp() 61, 87, 351
 - example 394
- Mgra...() 176
- MgraAlloc() 98, 352
- MgraArc() 100, 354
- MgraArcFill() 100, 355
- mgrab.c 390
- MgraBackColor() 99, 356
- MgraClear() 99, 357
- MgraColor() 99, 358
- MgraControl() 359
- MgraDot() 100, 361
- MgraFill() 100–101, 362
- MgraFont() 102, 363
- MgraFontScale() 102, 364
- MgraFree() 98, 365
- MgraInquire() 366
- MgraLine() 100, 368
- MgraRect() 100, 369
- MgraRectFill() 100, 370
- MgraText() 102, 371
 - example 22, 77, 82
- MIL
 - file format 45, 253
 - header file 22
 - include file 22
 - objects 15, 180, 207
 - running application 22, 167
 - structure 166
- MIL modules
 - application 171
 - data allocation and access 172
 - data generation 176
 - digitizer control 174
 - display allocation 175
 - display control 65, 175
 - graphics 97, 176
 - I/O device control 103
 - system device 177
- mil.h 22, 168, 181
- mil.ini
 - Meteor-II 112

- milsetup.h 18, 20–21, 28, 104, 129, 168, 185, 195, 386
- MimBinarize()
 - example 72
- MimHistogramEqualize() 62
- MimLutMap()
 - example 394
- mmultdis.c 72
- MMX Technology, Intel 16
- mnatfct.c 394
- mnatgen.c 159
- monochromatic effect 63
- monochrome image buffer 29
- mstart.c 22
- Msys...() 177
- MsysAlloc() 21, 167, 372
 - example 82
- MsysControl() 374
 - example 159
- MsysFree() 21, 167, 378
 - example 82
- MsysInquire() 379
 - example 82, 159
- multi-dimensional buffers 128
- multi-head configuration
 - displaying in 68
- multiple buffers
 - displaying 72
- multi-processing 145–146
 - definition 146
- multi-threading 145, 147
 - definition 147
- MvgaDispDeselectClientArea()
 - VGA 81
- mwindisp.c 82

N

- native mode 157, 393
 - example code 159
 - flag 386
 - integrating with MIL 158
 - interface 158
 - portability 394
- non 8-bit buffers
 - displaying 70

O

- object identifier 180
- object type
 - pseudo-MIL function 407
 - pseudo-MIL, assign 409
- open communication 20, 23, 167
- overlay
 - example 77
 - simulated 77
 - usage 77
- Overlay/regular display 90
- Overlay/regular display architecture 90
- overlay/regular display architecture 322
- overwriting data 29

P

- packed binary buffers 38
- palette
 - image 61
- panning, display 75, 345
- parameter
 - double, pseudo-MIL 414
 - long, pseudo-MIL 417
 - MIL_ID, pseudo-MIL 415
 - null-terminated string, pseudo-MIL 421
 - pointer, pseudo-MIL 418
 - short, pseudo-MIL 420
- parameter checking control 188
- parent buffer 36, 42, 169
 - display 72, 75
- physical memory 29
 - buffer allocation 38
- picth 55
- pixel
 - coordinates 58
 - depth 15
 - value, minimum/maximum 113
- pointer
 - pseudo-MIL object 408
 - pseudo-MIL object, assign 410
- portability
 - native mode 158
- portability, native mode 394
- predictive coding 138–139
- preprocess
 - input data 114

- processing
 - attribute 209, 212, 216, 244, 247
 - compensation 374
 - control 188
 - limiting 42
 - system, force 374
- program examples 18
- pseudo-color
 - effect 63
- pseudo-MIL commands 394
- pseudo-MIL functions 394, 399
- Pulsar
 - system 373
- put data
 - 1D data buffer 291
 - 2D data buffer 292
 - array, from 44
 - data buffer 283–284, 286

Q

- quantization 140

R

- ramp, LUT 61
- read.me 18–19, 22–23
- rectangles, draw 100, 369
- rectangles, draw filled 177, 370
- reference level
 - analog 113
 - black/white 113, 316, 318
 - controls 113, 318
 - digitizer 316
 - input channel 113, 316
- reporting errors 169
- resident software, required 386
- restart markers 141
- restore
 - data buffer 293
 - LUT buffer 62
- retrieve data
 - 1D data buffer 267
 - 2D data buffer 268
 - color bands 260, 262
 - data buffer 259–260, 262
- RGB
 - buffers 47

S

- sample program 22
- saturation
 - HLS 221, 224, 236, 239, 261, 263, 285, 287
- save
 - color image 131
 - data 44–45, 253, 295
- scale, input 115, 299
- scaling 115
- scrolling, display 75, 345
- seed fill, boundary-type 362
- select
 - digitizer input channel 298
 - image to display 346
- setup flag 386
- single-screen configuration
 - displaying in 67
 - VGA 67
- size
 - child buffers 42
 - data buffer 37
 - display 70
 - image buffer 29
 - LUT buffer 61
 - system display 67
 - text character 102
- software triggers
 - Corona 125
- source buffer 29
- speed
 - multi-threading 147
- stop grabbing 33, 310
- storage area 29
- strobe device 105
- structure, MIL 166
- structuring elements
 - buffer allocation 39
- subsampling input 115
- synchronization
 - of grab 116, 301
 - thread 147–148
 - with grab end 301

system

- allocation 20, 372
- buffers 29
- configuration 18
- control behavior 374
- default 15, 18
- default setup configuration 386
- definition 14
- device 63, 128, 130, 167–168
- display criteria 29
- free 378
- Genesis 373
- grab criteria 33
- Host 372
- initialization 18, 104
- inquire 366, 379
- Meteor 372
- module 177
- multiple 21
- multi-processing capabilities 146
- number 372
- Pulsar 373
- type 372
- VGA 372

T

- target system
 - system 15
- test installation program 23
- text
 - character font 102, 363
 - character size 364
 - graphics 102
 - support 98
 - write 177, 371
- theoretical data line 265, 289
- thread
 - allocate or control 190
 - application context 148
 - data sharing 147
 - error reporting 148–149
 - multi-threading 145, 150
 - synchronization 148
- TIFF file format 253
- timer control 208

toolkit

- Function Developers' 157, 393
- Native Mode Programmers' 393

trace

- application 170
- hook 202, 333
- mechanism control 188
- transforming data 45, 253, 269
- trigger device 105
- triggers 123, 125
 - Corona 120, 124
- true color effect 64

U

- Underlay display 89
- Underlay display architecture 89
- underlay display architecture 322
- user-allocated buffer 55

V

- VGA
 - system 372

W

- wait, grab 309
- Window occlusion
 - Meteor-II 113
- Windows
 - custom window, VGA 81

X

- xfontscale, inquire 366

Y

- yfontscale, inquire 366
- YUV buffers 49

Z

- zoom
 - display 75, 348
 - example 75

Product Assistance Request Form

[illegible]