OpenML® Media Library Software Development Kit Programmer's Guide

007–4504–002

New Features in this Guide

This guide contains the following changes:

- Technical changes to support the 1.1.2 release were incorporated.
- Documentation errors were corrected based on OpenML 1.0 specification errata sheet.
- Examples were updated and ML parameter descriptions were clarified.
- Information about the unused `ML_AUDIO_COMPANDING_INT32` parameter was removed.
- A glossary of ML terms was added. See Glossary on page 149.

Record of Revision

Version	Description
001	November 2001 Supports the 1.0 release of the OpenML Media Library Software Development Kit (ML).
002	October 2005 Supports the 1.0 release of the OpenML Media Library Software Development Kit (ML) and ML 1.1.2 for Linux IA32, SGI ProPack 4 Service Pack 2 or later running on the SGI Altix 350 system and Graphics Prism Visualization Systems

Contents

About This Guide . . . . . **xxiii**
Scope of this Guide xxiii
Related Publications xxiii
Obtaining Publications xxv
Conventions xxvi
Reader Comments xxvi

1. Introduction **1**
ML Terminology 1
Getting Started with ML 3
Simple Audio Output Program 3
Step 1: Include the `ml.h` and `mlu.h` Files 4
Step 2: Locate a Device 4
Step 3: Open the Device Output Path 5
Step 4: Set Up the Audio Device Path 5
Step 5: Set Controls on Audio Device Path 6
Step 6: Send Buffer to Device for Processing 7
Step 7: Begin Message Processing 7
Step 8: Receive the Reply Message 8
Step 9: Close the Path 8
Realistic Audio Output Program 8
Step 1: Include the `ml.h` and `mlu.h` Files 9
Step 2: Locate a Device 9
Step 3: Open the Device Output Path 9

Step 4: Allocate Buffers . . . 9
Step 5: Send Buffers to the Open Path . . . 10
Step 6: Begin the Transfer . . . 10
Step 7: Receive Replies from the Device . . . 11
Step 8: Refill the Buffer for Further Processing . . . 12
Step 9: End the Transfer . . . 12
Step 10: Close the Path . . . 12
Audio/Video Jacks . . . 13
Open a Jack . . . 13
Construct a Message . . . 14
Set Jack Controls . . . 14
Close a Jack . . . 15

2. Parameters . . . 17
`param/value` Pairs in an `MLpv` Message . . . 17
Scalar Values . . . 18
Set Scalar Values . . . 18
Get Scalar Values . . . 19
Array Values . . . 20
Set the Value of an Array Parameter . . . 20
Get the Size of an Array Parameter . . . 21
Get the Value of an Array Parameter . . . 21
Pointer Values . . . 22

3. Capabilities . . . 25
Capabilities Tree . . . 25
Utility Functions for Capabilities . . . 26
Manual Access to Capabilities . . . 26

Accessing Capabilities . . . 27
Get Local System Capabilities . . . 27
Get Physical Devices . . . 28
Get Logical Devices . . . 28
Get Parameters Accepted by a Path . . . 28
Query Individual Parameters of Logical Devices . . . 29
Query Parameters That Describe Parameters . . . 29
Identification Numbers . . . 30
Summary of Capabilities . . . 31
System Capabilities . . . 31
Physical Device Capabilities . . . 32
Logical Device Capabilities . . . 33
Jack Logical Device Capabilities . . . 33
Path Logical Device Capabilities . . . 34
Transcoder Logical Device Capabilities . . . 36
Pipe Logical Device Capabilities . . . 37
Device Open Options . . . 38
Jack Open Parameters . . . 38
Path Open Parameters . . . 39
Transcoder Open Parameters . . . 40
Accessing and Freeing Capabilities . . . 41
Finding a Parameter in a Capabilities List . . . 41
Obtaining Parameter Capabilities . . . 42
Freeing Capabilities Lists . . . 44

4. Audio/Visual Paths . . . 45
Opening a Logical Path . . . 45

Constructing a Message 46
Processing Out-of-Band Messages 46
Sending In-Band Messages 47
Processing In-Band Messages 48
Processing Exception Events 49
Processing In-Band Reply Messages 50
Beginning and Ending Transfers 51
Closing a Logical Path 51

5. Transcoders 53
Finding a Suitable Transcoder 54
Opening a Logical Transcoder 54
Controlling the Transcoder 54
Sending Buffers 55
Starting a Transfer 56
Changing Controls During a Transfer 56
Receiving a Reply Message 57
Ending Transfers 57
Closing a Transcoder 58
Work Functions 58

6. Video Parameters 59
Temporal Video Sampling 59
Progressive Sampling 60
Interlaced Sampling 60
Video Parameter Descriptions 61
`ML_VIDEO_ALPHA_SETUP_INT32` 61
`ML_VIDEO_BLUE_SETUP_INT32` 61

ML_VIDEO_BRIGHTNESS_INT32 61
ML_VIDEO_COLORSPACE_INT32 61
ML_VIDEO_CONTRAST_INT32 62
ML_VIDEO_DITHER_FILTER_INT32 62
ML_VIDEO_FILL_ALPHA_REAL32 62
ML_VIDEO_FILL_BLUE_REAL32 62
ML_VIDEO_FILL_Cb_REAL32 63
ML_VIDEO_FILL_Cr_REAL32 63
ML_VIDEO_FILL_GREEN_REAL32 63
ML_VIDEO_FILL_RED_REAL32 63
ML_VIDEO_FILL_Y_REAL32 63
ML_VIDEO_FLICKER_FILTER_INT32 63
ML_VIDEO_GENLOCK_SIGNAL_PRESENT_INT32 64
ML_VIDEO_GENLOCK_SOURCE_TIMING_INT32 64
ML_VIDEO_GENLOCK_TYPE_INT32 64
ML_VIDEO_GREEN_SETUP_INT32 64
ML_VIDEO_H_PHASE_INT32 64
ML_VIDEO_HEIGHT_F1_INT32 64
ML_VIDEO_HEIGHT_F2_INT32 65
ML_VIDEO_HUE_INT32 65
ML_VIDEO_INPUT_DEFAULT_SIGNAL_INT64 65
ML_VIDEO_NOTCH_FILTER_INT32 65
ML_VIDEO_OUTPUT_DEFAULT_SIGNAL_INT64 65
ML_VIDEO_OUTPUT_REPEAT_INT32 66
ML_VIDEO_PRECISION_INT32 66
ML_VIDEO_RED_SETUP_INT32 66
ML_VIDEO_SAMPLING_INT32 67
ML_VIDEO_SATURATION_INT32 67
ML_VIDEO_SIGNAL_PRESENT_INT32 67

ML_VIDEO_START_X_INT32 67
ML_VIDEO_START_Y_F1_INT32 67
ML_VIDEO_START_Y_F2_INT32 68
ML_VIDEO_TIMING_INT32 68
Standard Definition (SD) Timings 68
High Definition (HD) Timings 69
ML_VIDEO_V_PHASE_INT32 69
ML_VIDEO_WIDTH_INT32 69
Video Example 70

7. Image Buffer Parameters 71

Image Buffer Layouts 71
Image Buffer Parameters Summary 73
ML_IMAGE_BUFFER_POINTER 73
ML_IMAGE_BUFFER_SIZE_INT32 74
ML_IMAGE_COLORSPACE_INT32 74
ML_IMAGE_COMPRESSION_FACTOR_REAL32 76
ML_IMAGE_COMPRESSION_INT32 76
ML_IMAGE_DOMINANCE_INT32 77
ML_IMAGE_HEIGHT_1_INT32 78
ML_IMAGE_HEIGHT_2_INT32 78
ML_IMAGE_INTERLEAVE_MODE_INT32 78
ML_IMAGE_ORIENTATION_INT32 79
ML_IMAGE_PACKING_INT32 79
ML_IMAGE_ROW_BYTES_INT32 82
ML_IMAGE_SAMPLING_INT32 82
ML_IMAGE_SKIP_PIXELS_INT32 84
ML_IMAGE_SKIP_ROWS_INT32 85

ML_IMAGE_TEMPORAL_SAMPLING_INT32 . . . 85
ML_IMAGE_WIDTH_INT32 . . . 85
ML_SWAP_BYTES_INT32 . . . 85

8. Audio Parameters . . . 87
Audio Buffer Layout . . . 87
Audio Parameters Summary . . . 89
ML_AUDIO_BUFFER_POINTER . . . 89
ML_AUDIO_CHANNELS_INT32 . . . 89
ML_AUDIO_COMPRESSION_INT32 . . . 90
ML_AUDIO_FORMAT_INT32 . . . 90
ML_AUDIO_FRAME_SIZE_INT32 . . . 92
ML_AUDIO_GAINS_REAL64_ARRAY . . . 92
ML_AUDIO_PRECISION_INT32 . . . 92
ML_AUDIO_SAMPLE_RATE_REAL64 . . . 92
Uncompressed Audio Buffer Size Computation . . . 93

9. ML Processing . . . 95
ML Program Structure . . . 96
MLstatus Return Value . . . 97
Device States . . . 99
Opening a Jack, Path, or Transcoder . . . 100
Set Controls . . . 102
Get Controls . . . 103
Send Controls . . . 104
Send Buffers . . . 106
Query Controls . . . 109
Get Wait Handle . . . 111
Begin Transfer . . . 112

Transcoder Work . . . 113
Get Message Count . . . 114
Receive Messsage . . . 115
End Transfer . . . 116
Close Processing . . . 116
Utility Functions . . . 117
Get Returned Parameters . . . 117
Get Version . . . 118
Status Name . . . 118
Message Name . . . 119
`MLpv` String Conversion Routines . . . 119
Example: Printing the Interpretation of a Video Timing Parameter . . . 122

10. Synchronization . . . 123
Time Representation . . . 123
Get Unadjusted System Time (UST) . . . 124
UST/MSC/ASC Parameters . . . 124
Unadjusted System Time (UST) Parameters . . . 125
Media Stream Count (MSC) Parameters . . . 125
Application Stream Count (ASC) Parameters . . . 126
UST/MSC and Corresponding Messages . . . 126
UST/MSC Example . . . 127
UST/MSC for Input . . . 127
UST/MSC for Output . . . 128
Predicate Controls . . . 130

Appendix A. Pixels in Memory . . . 133
Greyscale Examples . . . 133

8-bit Greyscale (1 Byte Per Pixel) . . . 133
Padded 12-bit Greyscale (1 Short Per Pixel) . . . 134
RGB Examples . . . 134
8-bit RGB (3 Bytes Per Pixel) . . . 135
8-bit BGR (3 Bytes Per Pixel) . . . 135
8-bit RGBA (4 Bytes Per Pixel) . . . 135
8-bit ABGR (4 Bytes Per Pixel) . . . 136
10-bit RGB (One 32-bit Integer Per Pixel) . . . 136
10-bit RGBA (One 32-bit Integer Per Pixel) . . . 136
12-bit RGBA (6 Bytes Per Pixel) . . . 137
Padded 12-bit RGB (Three 16-bit shorts per pixel) . . . 137
Padded 12-bit RGBA (Four 16-bit Shorts Per Pixel) . . . 137
CbYCr Examples . . . 138
8-bit CbYCr (3 Bytes Per Pixel) . . . 138
8-bit CbYCrA (4 Bytes Per Pixel) . . . 138
10-bit CbYCr (One 32-bit Integer Per Pixel) . . . 139
10-bit CbYCrA (One 32-bit Integer Per Pixel) . . . 139
Padded 12-bit CbYCrA (Four 16-bit Shorts Per Pixel) . . . 139
422*x* CbYCr Examples . . . 140
10-bit 422 CbYCr (5 Bytes Per 2 Pixels) . . . 140
10-bit 422 CbYCr (5 Bytes Per 2 Pixels) . . . 141
Padded 12-bit 422 CbYCr (Four 16-bit Shorts Per 2 Pixels) . . . 141
10-bit 4224 CbYCrA (Two 32-bit Integers Per 2 Pixels) . . . 142

Appendix B. Common Video Standards . . . 143

Glossary . . . 149

Index **153**

Figures

Figure 3-1 Capabilities Tree 26
Figure 6-1 Progressive Sampling: Film at 60 Frames-per-Second 60
Figure 6-2 Interlaced Sampling: Video at 60 Frames-per-Second 61
Figure 7-1 General Image Buffer Layout 72
Figure 7-2 Simple Image Buffer Layout 73
Figure 7-3 Field Dominance 77
Figure 8-1 Different Audio Sample Frames 88
Figure 8-2 Layout of an Audio Buffer with 4 Channels 89
Figure 10-1 Sample Input Rate 128
Figure 10-2 Actual Sample Rate 128
Figure 10-3 Determining Underflow 129
Figure 10-4 System Sequence Count 129
Figure 10-5 Predict when the Data will Hit the Output Jack 129
Figure B-1 525/60 Timing (NTSC) 144
Figure B-2 625/50 Timing (PAL) 145
Figure B-3 1080i Timing (High Definition) 146
Figure B-4 720p Timing (High Definition) 147

Tables

Table 3-1 System Capabilities 31

Table 3-2 Physical Device Capabilities 32

Table 3-3 Jack Logical Device Capabilities 33

Table 3-4 Path Logical Device Capabilities 34

Table 3-5 Transcoder Logical Device Capabilities 36

Table 3-6 Pipe Logical Device Capabilities 37

Table 3-7 Jack `mlOpen` Options 38

Table 3-8 Path `mlOpen` Options 39

Table 3-9 Transcoder `mlOpen` Options 40

Table 3-10 Parameters Returned by `mlPvGetCapabilities` 43

Table 7-1 Mapping Colorspace *representation* Parameters 74

Table 7-2 Effect of Sampling and Colorspace on Component Definitions 84

Table 7-3 Bit Reordering 86

Examples

Example 5-1 Set Image Width/Height on Pipes 55
Example 5-2 Send Source/Destination Buffers to Source/Destination Pipes 56
Example 7-1 `ML_COLORSPACE_RGB_709_FULL` 75
Example 8-1 Buffer Size Computation 93

About This Guide

This document provides information about the SGI OpenML Media Library Software Development Kit (*ML*). ML provides a cross-platform library for controlling digital media hardware. It supports audio and video I/O devices and transcoders.

This document is a general user's guide. For a more detailed treatment of a particular function, see the online reference pages for ML.

Scope of this Guide

This guide tells you how to use the ML function calls It contains the following chapters:

- Chapter 1, "Introduction" on page 1
- Chapter 2, "Parameters" on page 17
- Chapter 3, "Capabilities" on page 25
- Chapter 4, "Audio/Visual Paths" on page 45
- Chapter 5, "Transcoders" on page 53
- Chapter 6, "Video Parameters" on page 59
- Chapter 7, "Image Buffer Parameters" on page 71
- Chapter 8, "Audio Parameters" on page 87
- Chapter 9, "ML Processing" on page 95
- Chapter 10, "Synchronization" on page 123

Related Publications

For details about ML, see following man pages:

`mlAudioParameters`(3dm)
`mlBeginTransfer`(3dm)
`mlClose`(3dm)

mlEndTransfer(3dm)
mlFreeCapabilities(3dm)
mlGetCapabilities(3dm)
mlGetControls(3dm)
mlGetMessageCount(3dm)
mlGetSystemUST(3dm)
mlGetVersion(3dm)
mlGetWaitHandle(3dm)
mlImageParameters(3dm)
mlIntro(3dm)
mlMessageName(3dm)
mlOpen(3dm)
mlParameters(3dm)
mlPixel(3dm)
mlPvCopy(3dm)
mlPvFind(3dm)
mlPvGetCapabilities(3dm)
mlPvToString(3dm)
mlQueryControls(3dm)
mlReceiveMessage(3dm)
mlSendBuffers(3dm)
mlSendControls(3dm)
mlSetControls(3dm)
mlStatusName(3dm)
mlSynchronization(3dm)
mlVideoParameters(3dm)
mlXcodeGetOpenPipe(3dm)
mlXcodeWork(3dm)
mluCapabilities(3dm)
mluDefaults(3dm)
mluImageBufferSize(3dm)
mluPv(3dm)
mluSizes(3dm)
mluTCAddTC(3dm)
mluTCFramesBetween(3dm)
mluTCFramesPerDay(3dm)
mluTCToSeconds(3dm)
mluTCToString(3dm)

Also see the following:

- *ISO/IEC 13818-2 GENERIC CODING OF MOVING PICTURES AND ASSOCIATED AUDIO: SYSTEMS.*
- IEC 61834-1 (1997). Recording - *Helical-Scan Digital Video Cassette Recording System Using 6.35 mm Magnetic Tape for Consumer Use (525-60, 625-50, 1125-60, and 1250-50 Systems) - Part 1: General Specifications*
- IEC 61834-2 (1997). Recording - *Helical-Scan Digital Video Cassette Recording System Using 6.35 mm Magnetic Tape for Consumer Use (525-60, 625-50, 1125-60, and 1250-50 Systems) - Part 2: SD Format for 525-60 and 625-50 Systems*
- *SMPTE 314M Television - Data Structure for DV-Based Audio, Data and Compressed Video - 25 and 50 Mb/s* THE SOCIETY OF MOTION PICTURE AND TELEVISION ENGINEERS, 1997
- *A Technical Introduction to Digital Video* by Charles Poynton, published by John Wiley & Sons, 1996 (ISBN 0-471-12253-X, hardcover.

Obtaining Publications

You can obtain SGI documentation as follows:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- On IRIX systems, you can use InfoSearch (if installed), an online tool that provides a more limited set of online books, release notes, and man pages. Enter `infosearch` at a command line or select **Help > InfoSearch** from the Toolchest.
- On IRIX systems, you can view release notes by entering either `grelnotes` or `relnotes` at a command line.
- On Linux systems, you can view release notes on your system by accessing the README file(s) for the product. This is usually located in the `/usr/share/doc/`*productname* directory, although file locations may vary.
- On IRIX and Linux systems, you can view man pages by typing `man` *title* at a command line.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

  http://docs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1500 Crittenden Lane, M/S 535
  Mountain View, California 94043–1351

SGI values your comments and will respond to them promptly.

Introduction

The SGI OpenML Media Library Software Development Kit (*ML*) provides a cross-platform library for capturing, transporting, processing, and displaying synchronized media streams. The media streams could be uncompressed or compressed audio, video, and metadata (such as timecode) streams. The ML API provides a platform-independent interface to digital media hardware for media-rich application developers and digital content creators.

Note: The material in this guide assumes that ML is installed on your workstation, and that you have access to the online ML example programs.

This chapter discusses the following:

- "ML Terminology" on page 1
- "Getting Started with ML" on page 3
- "Simple Audio Output Program" on page 3
- "Realistic Audio Output Program" on page 8
- "Audio/Video Jacks" on page 13

ML Terminology

The following terms are used throughout this document, and some are used in the ML code:

Term	Definition
graphics / video	In ML, these terms are not synonymous: *graphics* indicates the graphical display used for the user-interface on a computer; *video* indicates the type of signal sent to a video cassette recorder or received from a camcorder.
capabilities tree	The hierarchy of all ML devices in the system, containing information about each ML device. An

	application may search a capability tree to find suitable media devices for operations you wish to perform.
system	The highest level in the capability tree hierarchy. It is the machine on which your application is running. This machine is given the name `ML_SYSTEM_LOCALHOST`. Each system contains one or more physical or logical devices.
physical device	A device that corresponds to device-dependent modules in ML. Typically, each device-dependent module supports a set of software transcoders or a single piece of hardware. Examples of devices are audio cards on a PCI bus, DV camcorders on the 1394 bus, and software DV modules. Each device-dependent module may expose a number of logical devices.
logical device	Jacks, paths, or transcoders.
jack	A logical device that is an interface in/out of the system. Examples of jacks are composite video connectors and microphones. Jacks often, but not necessarily, correspond to a physical connector — it is possible for a single ML jack to refer to several such connectors. It is also possible for a single physical connector to appear as several logical jacks.
path	A logical device that provides logical connections between memory and jacks. For example, a video output path transports data from buffers to a video output jack. Paths are logical entities. Depending on the device, it is possible for more than one instance of a path to be open and in use at the same time.
pipe	The connections from memory to the transcoder, and from the transcoder to memory.
transcoder	A logical device that takes data from buffers via an input pipe or pipes, performs an operation on the data, and returns the data to another buffer via an output pipe. Example transcoders are DV compression and JPEG decompression.

UST	Unadjusted system time. UST is a special system clock that runs continuously without adjustment. This clock is used to synchronize media streams.
MSC	Media stream count. MSC is a measure of the number of media samples that have passed though a jack. This measure is useful to synchronize media streams.

Getting Started with ML

The first thing you should do is examine your system with the `mlquery`(1ml) tool. This tool prints a list of all supported ML devices on the system.

Following is an example `mlquery` on the system mediaworks:

```
% mlquery

         SYSTEM: mediaworks.mycompany.com
     active UST: (default software UST source)

        DEVICES:
                 nullXcode:0
                 dm12-digvid:0
```

This output indicates that there are two installed devices:

- A software null transcoder
- An SGI DmediaPro DM12 — SD/HD Digital Video/Audio I/O device

Other options to `mlquery` allow you to gather more information about the installed devices. Use the `-h` option to `mlquery` for help.

Simple Audio Output Program

This example program outputs a short beep. To keep it simple, a few details (primarily error-checking) are skipped. This program only includes the operations required to produce the beep. The steps are as follows:

- "Step 1: Include the `ml.h` and `mlu.h` Files"
- "Step 2: Locate a Device" on page 4

- "Step 3: Open the Device Output Path" on page 5
- "Step 4: Set Up the Audio Device Path" on page 5
- "Step 5: Set Controls on Audio Device Path" on page 6
- "Step 6: Send Buffer to Device for Processing" on page 7
- "Step 7: Begin Message Processing" on page 7
- "Step 8: Receive the Reply Message" on page 8
- "Step 9: Close the Path" on page 8

Note: Consult the online example code for more advanced programs.

Step 1: Include the `ml.h` and `mlu.h` Files

To begin, you will need the following files:

File	Description
`ml.h`	Provides the core ML library functionality
`mlu.h`	Provides simple utility functions built on the core library

You may choose to use only the core library or you may find it convenient to use the simpler utility functions.

Include the files as follows:

```
#include <ML/ml.h>
#include <ML/mlu.h>
```

Step 2: Locate a Device

You must query the capabilities of the system to find a suitable digital media device with which to perform your audio output task. To do that, you must search the ML capability tree, which contains information on every ML device on the system.

In your search, you should start at the top of the tree as follows:

1. Query the local system to find the first physical device that matches your desired device name.
2. Look in that device to find its first output jack.
3. Find an output path that goes through that jack.

In this case, assuming that the device name is being passed in as a command-line argument, you can use some of the utility functions to find a suitable output path:

```
MLint64 devId=0;
MLint64 jackId=0;
MLint64 pathId=0;

mluFindDeviceByName( ML_SYSTEM_LOCALHOST, argv[1], &devId );
mluFindFirstOutputJack( devId, &jackId );
mluFindPathToJack( jackId, &pathid, memoryAlignment );
```

Step 3: Open the Device Output Path

An open device output path provides your application with a dedicated connection to the hardware. It also allocates system resources for use in subsequent operations. The device path is opened with an `mlOpen` call as follows:

```
mlOpen( pathId, NULL, &openPath );
```

If the `mlOpen` call is successful, you will get an open path identifier. All operations using that path must use its identifier.

Note: Sometimes an `mlOpen` call can fail due to insufficient resources (typically because too many applications may already be using the same physical device).

Step 4: Set Up the Audio Device Path

Set up the path you just opened for your operation. In this case, you will use signed 16-bit audio samples with the following:

- A single (mono) audio channel
- A gain of –12dB

- A sample rate of 44.1 kHz

In ML, applications communicate with devices using messages. These messages are known as `MLpv` messages because they consist of a list of `param/value` pairs. An `MLpv` ends with an `ML_END` to indicate completion.

For example:

```
mlpv controls[5];
MLreal64 gain = -12; /* decibels */

controls[0].param = ML_AUDIO_FORMAT_INT32;
controls[0].value.int32 = ML_AUDIO_FORMAT_S16;
controls[1].param = ML_AUDIO_CHANNELS_INT32;
controls[1].value.int32 = 1;
controls[2].param = ML_AUDIO_GAINS_REAL64_ARRAY;
controls[2].value.pReal64 = &gain
controls[2].length = 1;
controls[3].param = ML_AUDIO_SAMPLE_RATE_REAL64;
controls[3].value.real64 = 44100.0;
controls[4].param = ML_END;
```

Notice that this message contains both scalar parameters (for example, the number of audio channels) and an array parameter (the array of audio gains).

Step 5: Set Controls on Audio Device Path

After you have constructed the `MLpv controls` message, you must set the controls on the open audio path as follows:

```
mlSetControls(openPath, controls);
```

This call makes all the desired control settings and does not return until those settings have been sent to the hardware. If it returns successfully, it indicates that all of the control changes have been committed to the device (and you are free to delete or alter the `controls` message).

Note: All control changes within a single controls message are processed atomically: either the call succeeds (and they are all applied) or the call fails (and none are applied).

Assuming that the call succeeded, the path is now set up and ready to receive audio data.

Step 6: Send Buffer to Device for Processing

This example assumes that you have already allocated a buffer in memory and filled it with audio samples. To send that buffer to the device for processing, do the following:

1. Construct an `MLpv` message that describes the buffer. That message must include both a pointer to the buffer and the length of the buffer (in bytes):

```
MLpv msg[2];
 msg[0].param = ML_AUDIO_BUFFER_POINTER;
 msg[0].value.pByte = ourAudioBuffer;
 msg[0].length = sizeof(ourAudioBuffer);
 msg[1].param = ML_END;
```

2. Send the buffers message to the opened path:

```
mlSendBuffers(openPath, msg);
```

When the message is sent, it is placed on a queue of messages going to the device. The `mlSendBuffers` call does very little work: it gives the message a cursory look before sending it to the device for later processing.

Note: Unlike the `mlSetControls` call, the `mlSendBuffers` call does not wait for the device to process the message, it simply enqueues it and then returns.

Step 7: Begin Message Processing

You must tell the device to start processing enqueued messages. This is done with the `mlBeginTransfer` call as follows:

```
mlBeginTransfer(openPath);
```

The program can sleep while the device is busy working on the message as follows:

```
sleep(5)
```

Using `sleep` is simple, but the example in "Realistic Audio Output Program" on page 8 shows a better approach. See "Step 6: Begin the Transfer" on page 10.

Step 8: Receive the Reply Message

As the device processes each message, it generates a reply message that is sent back to our application. By examining that reply, you can confirm that the buffer was transferred successfully, as follows:

```
MLint32 messageType;
MLpv* message;

mlReceiveMessage(openPath, &messageType, &Message );

if( messageType == ML_BUFFERS_COMPLETE )
  printf("Buffer transferred!\n");
```

Step 9: Close the Path

After you have verified that the buffer transferred successfully, you can close the path as follows:

```
mlClose(openPath);
```

Closing the path ends active transfer and frees any resources allocated when the path was opened.

Realistic Audio Output Program

The procedure in "Simple Audio Output Program" on page 3 was for a single audio buffer. In the example in this section, you will process millions of audio samples, using the following procedure:

- "Step 1: Include the `ml.h` and `mlu.h` Files"
- "Step 2: Locate a Device"
- "Step 3: Open the Device Output Path" on page 9
- "Step 4: Allocate Buffers" on page 9
- "Step 5: Send Buffers to the Open Path" on page 10
- "Step 6: Begin the Transfer" on page 10
- "Step 7: Receive Replies from the Device" on page 11

- "Step 8: Refill the Buffer for Further Processing" on page 12
- "Step 9: End the Transfer" on page 12
- "Step 10: Close the Path" on page 12

Step 1: Include the `ml.h` and `mlu.h` Files

See "Step 1: Include the `ml.h` and `mlu.h` Files" on page 4.

Step 2: Locate a Device

See "Step 2: Locate a Device" on page 4.

Step 3: Open the Device Output Path

Open the device output path just as in the previous example in "Step 3: Open the Device Output Path" on page 5:

```
mlOpen( pathId, NULL, &openPath );
```

Opening the path also allocates memory for the message queues used to communicate with the device. One of those queues will hold messages sent from our application to the device, and one will hold replies sent from the device back to our application.

Step 4: Allocate Buffers

If you were only processing a short sound, you could preallocate space for the entire sound and perform the operation straight from memory. However, for a more general and efficient solution, you must allocate space for a small number of buffers and reuse each buffer many times to complete the whole transfer.

Assume that memory has been allocated for 12 audio buffers and that those buffers have been filled with the first few seconds of audio data to be output.

Step 5: Send Buffers to the Open Path

Send each of the 12 buffers to the open path. Here the queue of messages between application and device becomes more interesting. The following code segment enqueues all the audio buffers to the device:

```
int i;
for ( i=0, i < 12; ++i )
  {
    MLpv msg[3];
    msg[0].param = ML_AUDIO_BUFFER_POINTER;
    msg[0].value.pByte = (MLbyte*)buffers[i];
    msg[0].length = bufferSize;
    msg[1].param = ML_AUDIO_UST_INT64;
    msg[1].param = ML_END;
    mlSendBuffers( openPath, msg );
}
```

Notice that each audio buffer is sent in its own message. This is because each message is processed atomically, and therefore refers to a single instant in time. In addition to the audio buffer, this message also contains space for an audio unadjusted system time (UST) time stamp. That time stamp will be filled in as the device processes each message. It will indicate the time at which the first audio sample in each buffer passed out of the machine.

Step 6: Begin the Transfer

Tell the device to begin the transfer. It reads messages from its input queue, interprets the buffer parameters within them, and processes those buffers with the following:

```
mlBeginTransfer(openPath);
```

At this point, you could tell the program to sleep while the device processes the buffers, as was done in "Simple Audio Output Program" on page 3. However, a more efficient approach is to select the file descriptor for the queue of messages sent from the device back to your application. In ML terminology, that file descriptor is called a *wait handle* on the receive queue:

```
MLwaitable pathWaitHandle;
 mlGetReceiveWaitHandle(openPath, &pathWaitHandle);
```

Having obtained the wait handle, you can wait for it to fire by using `select` on IRIX or Linux, or `WaitForSingleObject` on Windows, as follows:

On IRIX or Linux:

```
fd_set fdset;
 FD_ZERO( &fdset);
 FD_SET( pathWaitHandle, &fdset);

 select( pathWaitHandle+1, &fdset, NULL, NULL, NULL );
```

On Windows:

```
WaitForSingleObject( pathWaitHandle, INFINITE );
```

Step 7: Receive Replies from the Device

After the `select` call fires, a reply will be waiting. Retrieve the reply from the receive queue as follows:

```
MLint32 messageType;
 MLpv* replyMessage;

 mlReceiveMessage(openPath, &messageType, &replyMessage );

 if( messageType == ML_BUFFERS_COMPLETE )
   printf("Buffer received!\n");
```

This reply has the same format and content as the buffers message that was originally enqueued, plus any blanks in the original message will have been filled in. In this case, the reply message includes the location of the audio buffer that was transferred, as well as a UST time stamp indicating when its contents started to flow out of the machine:

```
MLbyte* audioBuffer = replyMessage[0].value.pByte;
 MLint64 audioUST    = replyMessage[1].value.int64;
```

Note: The UST time stamp is useful to synchronize several different media streams (for example, to make sure the sounds and pictures of a movie match up).

Step 8: Refill the Buffer for Further Processing

You can refill the buffer with more audio data and send it back to the device to be processed again with the following:

```
mlSendBuffers(openPath, replyMessage);
```

In this case, you are making a small optimization. Rather than construct a whole new buffers message, simply reuse the reply to your original message.

At this point, you have processed the reply to one buffer. If you wish, you can now go back to the `select` call and wait for another reply from the device. This can be repeated indefinitely.

Step 9: End the Transfer

After enough buffers have been transferred, you can end the transfer as follows:

```
mlEndTransfer(openPath);
```

In addition to ending the transfer, this call performs the following:

- Flushes the queue to the device
- Aborts any remaining unprocessed messages
- Returns any replies on the receive queue to the application

The `mlEndTransfer` call is a blocking call. When it returns, the queue to the device will be empty, the device will be idle, and the queue from the device to your application will contain any remaining replies.

If you wish, you can send more buffers to the path (see "Step 5: Send Buffers to the Open Path" on page 10).

Step 10: Close the Path

Use the following to close the path:

```
mlClose(openPath);
```

Note: This section has provided only a quick introduction to an audio output device. Through a similar interface, ML also supports audio input, video input, video output, and memory-to-memory transcoding operations.

Audio/Video Jacks

ML is concerned with the following types of interfaces:

- Jacks for control of external adjustments
- Paths for audio and video through jacks in/out of the machine
- Pipes to/from transcoders

All share common control, buffer, and queueing mechanisms. This section describes these mechanisms in the context of operating on a jack and its associated path:

- "Open a Jack" on page 13
- "Construct a Message" on page 14
- "Set Jack Controls" on page 14
- "Close a Jack" on page 15

Open a Jack

To open a connection to a jack, call `mlOpen`:

```
MLstatus mlOpen(const MLint64 objectId, MLpv* options, MLopenid* openId);
```

A jack is usually an external connection point and most often one end of a path. Jacks may be shared by many paths or they may have other exclusivity inherent in the hardware. For example, a common video decoder may have a multiplexed input shared between composite and S-video. If only one can be in use at a given instance, then there is an implied exclusiveness between them.

Many jacks do not support an input message queue because an application cannot send data to a jack (it must be sent via a path). Therefore, `mlSendControls` and `mlSendBuffers` are not supported on a jack; you must use `mlSetControls` to adjust controls. Typically, the adjustments on a path affect hardware registers and can

be changed while a data transfer is ongoing (on a path that connects the jack to memory). Examples are brightness and contrast.

Some controls are not adjustable during a data transfer. For example, the timing of a jack cannot usually be changed while a data transfer is in effect. Reply messages may be sent by jacks and usually indicate some external condition, such as synchronization lost or gained.

Construct a Message

Messages are arrays of parameters, where the last parameter is always `ML_END`. For example, you can adjust the flicker and notch filters with a message such as the following:

```
MLpv message[3];
message[0].param = ML_VIDEO_FLICKER_FILTER_INT32;
message[0].value.int32 = 1;
message[1].param = ML_VIDEO_NOTCH_FILTER_INT32;
message[1].value.int32 = 1;
message[2].param = ML_END
```

Set Jack Controls

Jack controls deal with external conditions and not processing associated with data transfers. Therefore, applications use `mlSetControls` or `mlGetControls` calls to manipulate these controls. Following is an example of how you can obtain the external synchronization signal (genlock) vertical and horizontal phase immediately:

```
MLpv message[3];
message[0].param = ML_VIDEO_H_PHASE_INT32;
message[1].param = ML_VIDEO_V_PHASE_INT32;
message[2].param = ML_END;
if( mlGetControls( aJackConnection, message))   handleError();
 else
   printf("Horizontal offset is %d, Vertical offset is %d\n",
      message[0].value.int32, message[1].value.int32);
```

`mlSetControls` and `mlGetControls` are blocking calls. If the call succeeds, the message has been successfully processed.

Note: Not all controls may be set via `mlSetControls`. The access privilege in the `param` capabilities can be used to verify when and how controls can be modified.

Close a Jack

When an application has finished using a jack, it should close it with `mlClose`:

```
MLstatus mlClose(MLopenid openId);
```

All controls previously set by this application normally remain in effect although they may be modified by other applications.

Parameters

This chapter describes the ML parameter syntax and semantics. These parameters define variables including control values (such as the frame rate or image width) and location of data (such as a single video field).

This chapter contains the following sections:

- "`param/value` Pairs in an `MLpv` Message"
- "Scalar Values" on page 18
- "Array Values" on page 20
- "Pointer Values" on page 22

`param/value` Pairs in an `MLpv` Message

The fundamental building block of ML is the `param/value` pair that makes up an `MLpv` message, as shown here:

```
typedef struct {
    MLint64 param;
    MLvalue value;
    MLint32 length;
    MLint32 maxLength;
} MLpv;
```

The `param` is a unique numeric identifier for each parameter and the `value` is a `union` of several possible types. For example:

```
typedef union {
      MLbyte    byte;    /* 8-bit  signed byte values */
      MLint32   int32;   /* 32-bit signed integer values */
      MLint64   int64;   /* 64-bit signed integer values */
      MLbyte*   pByte;   /* pointer to an array of bytes */
      MLreal32* real32;  /* 32-bit floating point value */
      MLreal64* real64;  /* 64-bit floating point value */
      MLint32*  pInt32;  /* pointer to an array of 32-bit signed integer values */
      MLint64*  pInt64;  /* pointer to an array of 64-bit signed integer values */
      MLreal32* pReal32; /* pointer to an array of 32-bit floating point values */
      MLreal64* pReal64; /* pointer to an array of 64-bit floating point values */
      struct_MLpv*pPv;   /* pointer to a message of param/value pairs*/
      struct_MLpv** ppPv;/* pointer to an array of messages */
}MLvalue;
```

In ML, applications communicate with devices using messages. Each message is a simple array of `param/value` pairs. An `MLpv` message ends with the `ML_END` parameter to indicate completion.

For example, the following is a message that sets image width to 1920 and image height to 1080:

```
MLpv controls[3];
controls[0].param = ML_IMAGE_WIDTH_INT32;
controls[0].value.int32 = 1920;
controls[1].param = ML_IMAGE_HEIGHT_INT32;
controls[1].value.int32 = 1080;
controls[2].param = ML_END;
```

Scalar Values

This section shows you how to set and get scalar values.

Set Scalar Values

To set the values of scalar parameters, you must enter the `param` and `value` fields of each `MLpv` and send the result to a device. If the value is valid, the returned length

will be `1`. If the value is invalid, or if the parameter is not recognized by the device, an error status will be returned and length will be set to `-1`.

Note: Do not set the `length` or `maxLength` fields because they are ignored when setting scalars. However, on return (`mlReceiveMessage`), a length parameter that equals `-1` indicates that this parameter was in error.

For example, to set video timing:

```
MLpv message[2];
message[0].param = ML_VIDEO_TIMING_INT32;
message[0].value.int32 = ML_TIMING_525;
message[1].param = ML_END;
if( mlSetControls( someOpenVideoPath, message) )
  fprintf(stderr, "Error, unable to set timing\n");
```

Get Scalar Values

To get scalar values, you again construct a `MLpv` list, but you do not need to set the `value` field. As the device processes the `MLpv` list, it fills in the `value` and `length` fields. If the `value` is valid, the returned `length` is `1`. If the `value` is invalid, or the parameter is not recognized by the device, an error status will be returned and `length` is set to `-1`.

For example, to get video timing:

```
MLpv message[2];
message[0].param = ML_VIDEO_TIMING_INT32;
message[1].param = ML_END;
mlGetControls( someOpenVideoPath, message);
if( message[0].length == 1 )
  printf("Timing is %d\n", message[0].value.int32);
else
  fprintf(stderr, "Unable to determine timing\n");
```

Array Values

An array in ML is much like an array in C:

- `value` of the `MLpv` is a pointer to the first element of the array
- `length` is the number of valid elements in the array
- `maxLength` is the total length of the array

Each element increases the length of the array by `1`, so an array of four 32-bit integers would require a `maxLength` of four.

Set the Value of an Array Parameter

To set the value of an array parameter, fill out the `param`, `value`, `length`, and `maxLength` fields. If the values are valid, the returned length will be unaltered. If the values are invalid or if the parameter is not recognized at all by the device, an error status will be returned and length will be set to `-1`.

For example, use `ML_AUDIO_GAINS_REAL64_ARRAY` to set the gain on a 4-channel audio path:

```
MLreal64 data[] = { -12.0, 1.0, 1.0, 12.0 };
MLpv message[2];
message[0].param = ML_AUDIO_GAINS_REAL64_ARRAY;
message[0].value.pReal64 = data;
message[0].length = sizeof(data) / sizeof(MLreal64)
message[1].param = ML_END;
mlSetControls( someOpenPath, message );
```

Note: Do not set the `maxLength` field because it is ignored when setting an array parameter.

In the preceding example, you are free to modify the data array at any time before calling `mlSetControls`. You regain that right as soon as `mlSetControls` returns.

If you have a multithreaded application, your application must ensure that the data array is not accessed by some other thread while the `SetControls` call is in progress.

Get the Size of an Array Parameter

To get the size of an array parameter, set `maxLength` to `0`. The device will fill in `maxLength` to indicate the minimal array size to hold that value. If the parameter is not recognized by the device, an error status will be returned, `maxLength` will be set to `0`, and `length` will be set to `-1`.

For example:

```
MLpv message[2];
message[0].param = ML_PATH_TYPE_REAL64_ARRAY;
message[0].length = 0;
message[0].maxLength = 0;
message[1].param = ML_END;
mlGetControls( someOpenPath, message );
printf("Size of real array is %d\n", message[0].maxLength);
```

Get the Value of an Array Parameter

To get the value of an array parameter, create an array with `maxLength` entries to hold the result, and set `length` to `0`. The device will fill in no more than `maxLength` array elements and set `length` to indicate the number of valid entries. If the values are invalid or if the parameter is not recognized at all by the device, an error status will be returned and `length` will be set to `-1`.

For example:

```
MLint32 data[10];
MLpv message[2];
message[0].param = ML_PATH_TYPE_INT32_ARRAY;
message[0].value.pInt32 = data;
message[0].length = 0;
message[0].maxLength = 10;
message[1].param = ML_END;
mlGetControls( someOpenPath, message );
if( message[0].length > 0 )
{
  printf("Received %d array entries\n", message[0].length);
  printf("The first entry is %d\n", data[0]);
}
```

Note: Your application controls memory allocation. If you want to get the whole array, but do not know the maximum size, you must do the following:

1. Query for `maxLength`.
2. Allocate space for the result.
3. Query for the value.

Pointer Values

The distinction between array values and pointer values in ML is subtle, but important.

Array values are copied when they are passed to or received from a device. Thus, your application owns the array memory and is nearly always free to modify or free it.

A pointer parameter is a special type of array parameter that is used to send and receive data buffers (as arrays of bytes.) **Pointer values are not copied.** Instead, only the location of the data is passed to the device. The application sends a buffer by calling `mlSendBuffer`. `mlSendBuffer` places the controls and buffer pointer in the data payload area and inserts a header on the send queue for the device.

This is much more efficient, but it imposes a restriction: after a pointer value is given to a device, that memory cannot be touched until the device has finished processing it.

Note: For efficient processing, all buffers must be pinned in memory.

For example, the following code fragment shows how a pointer parameter might be initialized to send an image to a video input path:

```
MLpv message[2];
message[0].param = ML_IMAGE_BUFFER_POINTER;
message[0].value.pByte = someBuffer;
message[0].maxLength = sizeof(someBuffer);
message[1].param = ML_END;
if( mlSendBuffers( someOpenPath, message ) )
  fprintf(stderr, "Error sending buffers\n");
```

The above `mlSendBuffers` call places the message on a queue to be processed by the device, and then returns. It does not wait for the device to finish with the buffer. Thus, even after the call to `mlSendBuffers`, the device still owns the image buffer. Your application must not touch that memory until it is notified that processing is complete.

When you send a buffer to be filled, the device uses `maxLength` to determine how much it may write. It returns `length` set to indicate the amount of the buffer it actually used.

When you send a buffer for output, the device will interpret the length as the maximum number of bytes of valid data in the buffer. In this case, `maxLength` is ignored.

Capabilities

This chapter describes the ML capabilities tree, the repository of information on all installed ML devices. The capabilities tree tells you everything from the hardware location of a physical device to the range of legal values for supported parameters.

This chapter contains the following sections:

- "Capabilities Tree" on page 25
- "Utility Functions for Capabilities" on page 26
- "Manual Access to Capabilities" on page 26
- "Identification Numbers" on page 30
- "Summary of Capabilities" on page 31

Capabilities Tree

The capabilities tree forms a hierarchy that describes the installed ML devices in the following order from top to bottom, as shown in Figure 3-1:

1. Physical system
2. Physical devices
3. Logical devices
4. Supported parameters on the logical devices

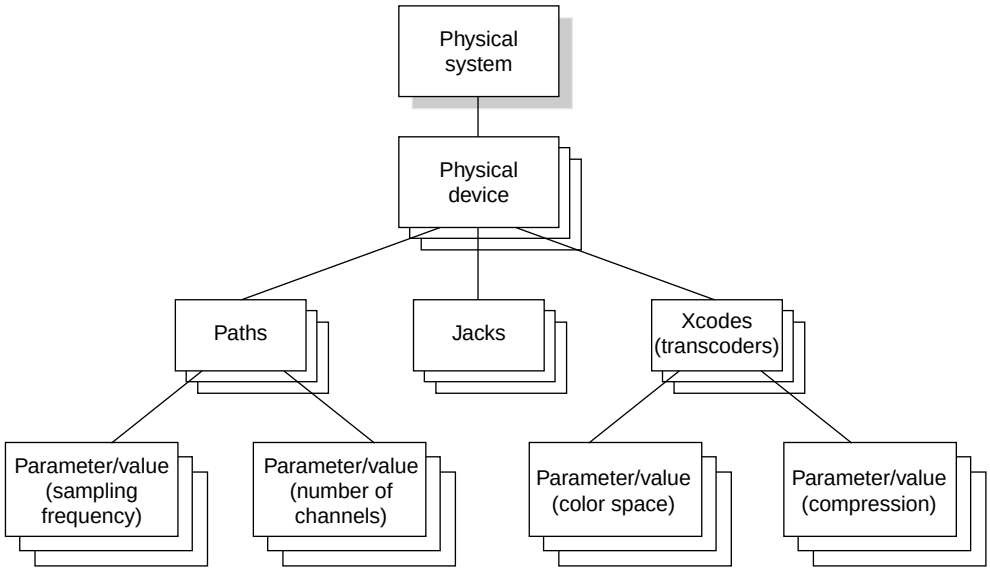

Figure 3-1 Capabilities Tree

Utility Functions for Capabilities

To access the capability hierarchy, you may either search the capability tree directly or make use of convenient utility functions to perform the search for you. This chapter discusses the baseline functionality provided in the core ML library, but you may also wish to examine the utility library and example code for pre-written alternatives.

Manual Access to Capabilities

Direct access to the ML capabilities tree is via the following functions:

Function Call	Description
`mlGetCapabilities`	Calls the capabilities for an ML object
`mlPvGetCapabilities`	Calls the capabilities for a parameter on a given device

`mlFreeCapabilities`	Releases a set of capabilities when you have finished using them

This section discusses the following:

- "Accessing Capabilities" on page 27
- "Query Individual Parameters of Logical Devices" on page 29
- "Query Parameters That Describe Parameters" on page 29

Accessing Capabilities

The following code examples show you how to query for the capabilities of your entire capability tree:

- "Get Local System Capabilities" on page 27
- "Get Physical Devices" on page 28
- "Get Logical Devices" on page 28
- "Get Parameters Accepted by a Path" on page 28

Note: All objects in ML are referred to via 64-bit identifying numbers. For example, the 64-bit ID number for the system on which your application is running is `ML_SYSTEM_LOCALHOST`.

Get Local System Capabilities

The following example shows how you can get the capabilities of the local system. The following code will give you an `MLpv` list that includes an array of identifiers for all of the physical devices installed on this system:

```
MLpv* systemCap;
mlGetCapabilities( ML_SYSTEM_LOCALHOST, &systemCap);
```

Get Physical Devices

To list the number of physical devices on your system, use the following example :

```
MLpv* deviceIds = mlPvFind( systemCap, ML_SYSTEM_DEVICE_IDS_INT64_ARRAY);
printf("There are %d physical devices\n", deviceIds->length );
if( deviceIds->length > 0 )
  printf("The first device has id %llx\n", deviceIds->value.pInt64[0]);
mlFreeCapabilities( systemCap );
```

Get Logical Devices

To examine a physical device for its supported I/O paths and transcoders (that is, its logical devices), use the following example:

```
MLpv* deviceCap, *pathIds, *xcodeIds;
mlGetCapabilities( someDeviceId, &deviceCap);
pathIds = mlPvFind( deviceCap, ML_DEVICE_PATH_IDS_INT64_ARRAY);
xcodeIds = mlPvFind( deviceCap, ML_DEVICE_XCODE_IDS_INT64_ARRAY);
printf("Device supports %d i/o paths and %d transcoders\n",
pathIds->length, xcodeIds->length);
if ( pathIds->length > 0 )
  printf("The first i/o path has id %llx\n", pathIds->value.pInt64[0]);
mlFreeCapabilities( deviceCap );
```

Get Parameters Accepted by a Path

Descending still further down the capability tree, you can obtain the capabilities of any particular logical device by again calling `mlGetCapabilities`. The following example shows how to find the number of parameters that are accepted by a path:

```
MLpv* pathCap, *paramIds;
mlGetCapabilities( somePathId, &pathCap);
paramIds = mlPvFind( pathCap, ML_PARAM_IDS_INT64_ARRAY);
printf("Path supports %d parameters\n", paramIds->length);
if (paramIds->length > 0)
  printf("The first parameter has id %llx\n",paramIds->value.pInt64[0]);
mlFreeCapabilities( pathCap );
```

Query Individual Parameters of Logical Devices

At this point, you have descended from the system to the logical device. Still there is one more level: the parameter. Querying the capabilities of a parameter is subtly different because the interpretation of parameters in ML is device-dependent (for example, the legal values for `ML_IMAGE_WIDTH_INT32` may be `1920` on one device and `720` on another). Thus, you must pass both a logical device ID and a parameter ID as shown in the following example:

```
MLpv* paramCap, *paramName;
mlPvGetCapabilities(someLogicalDeviceId, someParamId, &paramCap);
paramName = mlPvFind( paramCap, ML_NAME_BYTE_ARRAY );
if( paramName != NULL )
    printf("Param has name %s\n", (char *) ( paramName->value.pByte ));
mlFreeCapabilities( paramCap );
```

Note: Because the name of the parameter is being queried on a particular device, the above code will work for all parameters. This includes new device-dependent parameters.

See the `mlPvToString`(3dm) man page for a simpler way to find a parameter name.

Query Parameters That Describe Parameters

In addition to obtaining the capabilities of device parameters, you may also obtain the capabilities of the parameters used to describe the capabilities themselves. Because the capabilities parameters are not device-dependent, `deviceID` may be left empty. The following example shows how to find a text name for the capability parameter `ML_PARENT_ID_INT64`:

```
MLpv* paramCap, *paramName;
mlPvGetCapabilities(someLogicalDeviceId, someParamId, &paramcap);
paramName = mlPvFind( paramCap, ML_NAME_BYTE_ARRAY);
if( paramName != NULL )
    printf("Param has name %s\n", (char *) ( paramName->value.pByte ));
mlFreeCapabilities( paramCap );
```

You can get the same result by using `mlPvToString`, which itself calls `mlPvGetCapabilities`.

Identification Numbers

There are the following types of ID numbers in ML:

ID Type	Definition
constant ID	Constant IDs have defined names and may be hard-coded. They are system-independent. Examples of constant IDs are `ML_SYSTEM_LOCALHOST` and `ML_IMAGE_WIDTH_INT32`.
static ID	Static IDs are allocated by the ML system as new hardware is added. They are machine-dependent and may change after a reboot or as the system is reconfigured by adding or removing devices. The static ID of a device may change if it is removed from the system and then reconnected. **Note:** Static IDs should never be written to a file or passed between machines. Examples of static IDs are the physical and logical device IDs returned in calls to `mlGetCapabilities`. If you must share such information between machines, you should use the text names (system-independent) that correspond to the static IDs.
open ID	Open IDs are allocated when logical devices are opened. They are machine-dependent, and have a limited lifetime — from when `mlOpen` is called until `mlClose` is called. **Note:** Open IDs should never be written to a file or passed between machines.

You can call `mlGetCapabilities` (or `mlPvGetCapabilities`) for any type of ID, but the list that is returned will always be static.

Summary of Capabilities

The following sections describe the capabilities of each type of ML object:

- "System Capabilities" on page 31
- "Physical Device Capabilities" on page 32
- "Logical Device Capabilities" on page 33
- "Device Open Options" on page 38
- "Accessing and Freeing Capabilities" on page 41

Note: The capabilities are not necessarily used in the order shown.

In these tables, the string in the **Parameter** column of the table is a shortened form of the full parameter name. The full parameter name is of the form `ML_`*parameter_type*, where *parameter* and *type* are the strings listed in the **Parameter** and **Type** columns, respectively. For example, the full name of `ID` is `ML_ID_INT64`.

System Capabilities

Table 3-1 contains the system capabilities listed when a system ID is queried.

Table 3-1 System Capabilities

Parameter	Type	Description
`ID`	`INT64`	Resource ID for this system.
`NAME`	`BYTE_ARRAY`	`NULL`-terminated ASCII string containing the hostname for this system.
`SYSTEM_DEVICE_IDS`	`INT64_ARRAY`	Array of physical device IDs (these need not be sorted or sequential). For more details on a particular device ID call `mlGetCapabilities`. This array could be of length zero.

Physical Device Capabilities

Table 3-2 shows the physical device capabilities. The only defined physical system ID is ML_SYSTEM_LOCALHOST.

Table 3-2 Physical Device Capabilities

Parameter	Type	Description
ID	INT64	Resource ID for this physical device.
NAME	BYTE_ARRAY	NULL-terminated ASCII description of this physical device (for example, HD Video I/O or AVC/1394).
PARENT_ID	INT64	Resource ID for the system to which this physical device is attached.
DEVICE_VERSION	INT_32	Version number for this particular physical device.
DEVICE_INDEX	INT32	Index for this physical device. This is used to distinguish multiple identical physical devices (indexes are generated with a consistent algorithm). Identical machine configurations will have identical indexes. For example, plugging a particular card into the first 64-bit, 66-MHz PCI slot in any system will give the same index number. Uniquely identifying a device in a system-independent way requires using both the name and index.
DEVICE_LOCATION	BYTE_ARRAY	Physical hardware location of this physical device (on most platforms, this is the hardware graph entry). This makes it possible to distinguish between two devices on the same I/O bus and two devices each with its own I/O bus.
DEVICE_JACK_IDS	INT64_ARRAY	Array of jack IDs. For more details on a particular jack ID, call mlGetCapabilities. This array could be of length zero.
DEVICE_PATH_IDS	INT64_ARRAY	Array of path IDs. For more details on a particular path ID, call mlGetCapabilities. This array could be of length zero.
DEVICE_XCODE_IDS	INT64_ARRAY	Array of transcoder device IDs (these need not be sorted or sequential). For more details on a particular transcoder ID, call mlGetCapabilities. This array could be of length zero.

Logical Device Capabilities

This section discusses the following:

- "Jack Logical Device Capabilities" on page 33
- "Path Logical Device Capabilities" on page 34
- "Transcoder Logical Device Capabilities" on page 36
- "Pipe Logical Device Capabilities" on page 37

Jack Logical Device Capabilities

Table 3-3 shows the capabilities for a jack logical device.

Table 3-3 Jack Logical Device Capabilities

Parameter	Type	Description
ID	INT64	Resource ID for this jack.
NAME	BYTE_ARRAY	NULL-terminated ASCII description of this jack (for example, Purple S-video).
PARENT_ID	INT64	Resource ID for the physical device to which this jack is attached.
JACK_TYPE	INT32	Type of logical jack: • ML_JACK_TYPE_ADAT (digital ADAT standard jack) • ML_JACK_TYPE_AES (digital AES standard jack) • ML_JACK_TYPE_ANALOG_AUDIO (analog audio jack) • ML_JACK_TYPE_AUDIO (generic audio jack) • ML_JACK_TYPE_AUX (generic auxiliary jack) • ML_JACK_TYPE_COMPOSITE (composite video jack) • ML_JACK_TYPE_DUALLINK (SDI dual link jack) • ML_JACK_TYPE_GENLOCK (genlock jack) • ML_JACK_TYPE_GFX (digital graphics jack) • ML_JACK_TYPE_GPI (general purpose interface jack) • ML_JACK_TYPE_SDI (serial digital interface SDI jack) • ML_JACK_TYPE_SERIAL (generic serial control jack) • ML_JACK_TYPE_SVIDEO (SVideo jack) • ML_JACK_TYPE_VIDEO (generic video jack)

Parameter	Type	Description
JACK_DIRECTION	INT32	Direction of data flow through this jack: • ML_JACK_DIRECTION_BOTH (jack with data to and from memory) • ML_JACK_DIRECTION_IN (input jack with data for memory) • ML_JACK_DIRECTION_OUT (output jack with data from memory)
JACK_COMPONENT_SIZE	INT32	Maximum number of bits of resolution per component for the signal through this jack. Stored as an integer, therefore 8 means 8 bits of resolution.
JACK_PATH_IDS	INT64_ARRAY	Array of path IDs that may use this jack. (These need not be sorted or sequential.) For more details on a particular path ID, call mlGetCapabilities. This array could be of length zero.
PARAM_IDS	INT64_ARRAY	List of resource IDs for parameters that may be set and/or queried on this jack.
OPEN_OPTION_IDS	INT64_ARRAY	List of resource IDs for open option parameters that may be used when this jack is opened
JACK_FEATURES	BYTE_ARRAY	Double NULL-terminated list of ASCII feature strings. Each string represents a specific feature supported by this jack. Entries are separated by NULL characters (there are 2 NULL characters after the last string).

Path Logical Device Capabilities

Table 3-4 shows the parameters for path logical device capabilities.

Table 3-4 Path Logical Device Capabilities

Parameter	Type	Description
ID	INT64	Resource ID for this path.
NAME	BYTE_ARRAY	NULL-terminated ASCII description of this path (for example, Memory to S-Video Out).
PARENT_ID	INT64	Resource ID for the physical device on which this path resides.

Parameter	Type	Description
PARAM_IDS	INT64_ARRAY	List of resource IDs for parameters that may be set and/or queried on this path.
OPEN_OPTION_IDS	INT64_ARRAY	List of resource IDs for open option parameters that may be used when this path is opened.
PRESET	MSG_ARRAY	Each entry in the array is a message (a pointer to the head of an MLpv list, where the last entry in the list is ML_END). Each message provides a single valid combination of all setable parameters on this path. In particular, it should be possible to call mlSetControls using any of the entries in this array as the control's message. Each path is obligated to provide at least one preset parameter.
PATH_TYPE	INT32	Type of this path: • ML_PATH_TYPE_DEV_TO_DEV (path from device to another device) • ML_PATH_TYPE_DEV_TO_MEM (path from device to memory) • ML_PATH_TYPE_MEM_TO_DEV (path from memory to a device)
PATH_COMPONENT_ALIGNMENT	INT32	The location in memory of the first byte of a component (either an audio sample or a video line), must meet this alignment. Stored as an integer in units of bytes.
PATH_BUFFER_ALIGNMENT	INT32	The location in memory of the first byte of an audio or video buffer must meet this alignment. Stored as an integer in units of bytes.
PATH_SRC_JACK_ID	INT64	Resource ID for the jack that is the source of data for this path (unused if path is of type ML_PATH_TYPE_MEM_TO_DEV). For details on the jack ID, call mlGetCapabilities.
PATH_DST_JACK_ID	INT64	Resource ID for the jack that is the destination for data from this path (unused if path is of type ML_PATH_TYPE_DEV_TO_MEM). For details on the jack ID, call mlGetCapabilities.
PATH_FEATURES	BYTE_ARRAY	Double NULL-terminated list of ASCII features strings. Each string represents a specific feature supported by this path. Entries are separated by NULL characters (there are 2 NULL characters after the last string).

Transcoder Logical Device Capabilities

Table 3-5 shows parameters for the transcoder logical device capabilities.

Table 3-5 Transcoder Logical Device Capabilities

Parameter	Type	Description
ID	INT64	Resource ID for this transcoder.
NAME	BYTE_ARRAY	NULL-terminated ASCII description of this transcoder (for example, Software DV and DV25).
PARENT_ID	INT64	Resource ID for the physical device on which the transcoder resides.
PARAM_IDS	INT64_ARRAY	List of resource IDs for parameters that may be set and/or queried on this transcoder (may be of length 0).
OPEN_OPTION_IDS	INT64_ARRAY	List of resource IDs for open option parameters that may be used when this transcoder is opened
PRESET	MSG_ARRAY	Each entry in the array is a message (a pointer to the head of an MLpv list, where the last entry in the list is ML_END). Each message provides a single valid combination of all setable parameters on a transcoder. In particular, it should be possible to call mlSetControls using any of the entries in this array as the controls message. Each transcoder is required to provide at least one preset for each transcoder.
XCODE_ENGINE_TYPE	INT32	Type of the engine in this transcoder. The only defined transcoder type is: ML_XCODE_ENGINE_TYPE_NULL
XCODE _IMPLEMENTATION _TYPE	INT32	How this transcoder is implemented: • ML_XCODE_IMPLEMENTATION_TYPE_HW (hardware implementation) • ML_XCODE_IMPLEMENTATION_TYPE_SW (software implementation)
XCODE _COMPONENT _ALIGNMENT	INT32	The location in memory of the first byte of a component (either an audio sample or a video pixel) must meet this alignment. Stored as an integer in units of bytes.

Parameter	Type	Description
XCODE _BUFFER _ALIGNMENT	INT32	The location in memory of the first byte of an audio or video buffer must meet this alignment. Stored as an integer in units of bytes.
XCODE_FEATURES	BYTE_ARRAY	Double NULL-terminated list of ASCII features strings. Each string represents a specific feature supported by this transcoder. Entries are separated by NULL characters (there are 2 NULL characters after the last string).
XCODE_SRC_PIPE_IDS	INT64_ARRAY	List of pipe IDs from which the transcode engine may obtain buffers to be processes.
XCODE_DEST_PIPE_IDS	INT64_ARRAY	List of pipe IDs from which the transcode engine may obtain buffers to be filled with the result of its processing.

Pipe Logical Device Capabilities

Table 3-6 shows the parameters for the pipe logical device capabilities.

Table 3-6 Pipe Logical Device Capabilities

Parameter	Type	Description
ID	INT64	Resource ID for this path.
NAME	BYTE_ARRAY	NULL-terminated ASCII description of this pipe (DV Codec Input Pipe).
PARENT_ID	INT64	Resource ID for the transcoder on which this pipe resides.
PARAM_IDS	INT64_ARRAY	List of resource IDs for parameters that may be set and/or queried on this transcoder (may be of length 0).
PIPE_TYPE	INT32	Type of this pipe: • ML_PIPE_TYPE_ENGINE_TO_MEM (transcoder output pipe with data flow from engine to memory) • ML_PIPE_TYPE_MEM_TO_ENGINE (transcoder input pipe with data flow from memory to engine)

Device Open Options

This section discusses the following:

- "Jack Open Parameters" on page 38
- "Path Open Parameters" on page 39
- "Transcoder Open Parameters" on page 40

Jack Open Parameters

Table 3-7 describes the parameters that are supported when opening a jack.

Table 3-7 Jack `mlOpen` Options

Parameter	Type	Description
`OPEN_MODE`	`INT32`	Application's intended use for the device. Defined values are: • `ML_MODE_RO` (read only access) • `ML_MODE_RWE` (exclusive access.) • `ML_MODE_RWS` (shared read/write access) The default is defined by the device's capabilities.
`OPEN _RECEIVE _QUEUE_COUNT`	`INT32`	Application's preferred size (number of messages) for the receive queue. The size influences the amount of memory allocated for this queue when the device is opened. Default is device-dependent. A NULL value indicates that the application does not expect to receive any events from the jack.
`OPEN _EVENT _PAYLOAD_COUNT`	`INT32`	Application's preferred size (number of messages) for the queue event payload area. This payload area holds the contents of event messages on the receive queue. Default is device dependent. A `NULL` value indicates that the application does not expect to receive any events from the jack.

The `ML_OPEN_OPTION_IDS_INT64_ARRAY` returned by a `mlGetCapabilities` call using the jack ID returns a list of these parameters. `mlPvGetCapabilities` can then be used to discover allowable values.

Path Open Parameters

The following open parameters are supported when opening a path:

Table 3-8 Path `mlOpen` Options

Parameter	Type	Description
`OPEN_MODE`	`INT32`	Application's intended use for the device. Defined values are: • `ML_MODE_RO` (read only access) • `ML_MODE_RWE` (exclusive access) • `ML_MODE_RWS` (shared read/write access) The default is defined by the device's capabilities.
`OPEN_SEND_QUEUE_COUNT`	`INT32`	Application's preferred size (number of messages) for the send queue. This influences the amount of memory allocated for this queue when the device is opened. Default is device-dependent.
`OPEN_RECEIVE_QUEUE_COUNT`	`INT32`	Application's preferred size (number of messages) for the receive queue. This influences the amount of memory allocated for this queue when the device is opened. Default is device-dependent.
`OPEN_MESSAGE_PAYLOAD_SIZE`	`INT32`	Application's preferred size (in bytes) for the queue message payload area. The payload area holds messages on both the send and receive queues. Default is device-dependent.
`OPEN_EVENT_PAYLOAD_COUNT`	`INT32`	Application's preferred size (number of messages) for the queue event payload area. This payload area holds the contents of event messages on the receive queue. Default is device-dependent.
`OPEN_SEND_SIGNAL_COUNT`	`INT32`	Application's preferred number of empty message slots in the send queue. When the device dequeues a message and causes the number of empty slots to exceed this level, then the device will signal the send queue event. Default is device-dependent.

Transcoder Open Parameters

Table 3-9 describes the parameters that are supported when opening a transcoder.

Table 3-9 Transcoder `mlOpen` Options

Parameter	Type	Description
`OPEN_MODE`	`INT32`	Application's intended use for the device. Defined values are: • `ML_MODE_RO` (read only access) • `ML_MODE_RWE` (exclusive access) • `ML_MODE_RWS` (shared read/write access) The default is defined by the device's capabilities.
`OPEN _SEND _QUEUE_COUNT`	`INT32`	Application's preferred size (number of messages) for the send queue. The size influences the amount of memory allocated for this queue when the device is opened. Default is device-dependent.
`OPEN _RECEIVE _QUEUE_COUNT`	`INT32`	Application's preferred size (number of messages) for the receive queue. The size influences the amount of memory allocated for this queue when the device is opened. Default is device-dependent.
`OPEN _MESSAGE _PAYLOAD_SIZE`	`INT32`	Application's preferred size (in bytes) for the queue message payload area. The payload area holds messages on both the send and receive queues. Default is device-dependent.
`OPEN _EVENT _PAYLOAD_COUNT`	`INT32`	Application's preferred size (number of messages) for the queue event payload area. This payload area holds the contents of event messages on the receive queue. Default is device-dependent.
`OPEN _SEND _SIGNAL_COUNT`	`INT32`	Application's preferred l number of empty message slots in the send queue. When the device dequeues a message and causes the number of empty slots to exceed this level, then the device will signal the send queue event. Default is device-dependent.

Parameter	Type	Description
OPEN_XCODE_MODE	INT32	Application's preferred mode for controlling a software transcoder. This parameter does not apply to paths. Defined values are: • ML_XCODE_MODE_AYNCHRONOUS (processing by a software transcoder is to be initiated by ML). (Default.) • ML_XCODE_MODE_SYNCHRONOUS (processing by a software transcoder is to be initiated by the application).
OPEN_XCODE_STREAM	INT32	Selects between single- and multi-stream transcoders. In single-stream mode, source and destination buffers are processed at the same rate. In multi-stream mode, the source and destination pipes each have their own queue of buffers and may run at different rates (this is more complicated to program, but may be more efficient for some intra-frame codecs). Defined values are: • ML_XCODE_STREAM_MULTI (Default) • ML_XCODE_STREAM_SINGLE

Accessing and Freeing Capabilities

This section discusses the following:

- "Finding a Parameter in a Capabilities List" on page 41
- "Obtaining Parameter Capabilities" on page 42
- "Freeing Capabilities Lists" on page 44

Finding a Parameter in a Capabilities List

A parameter within a message or capabilities list may be found using the following:

```
MLpv* mlPvFind(MLpv* msg, MLint64 param);
```

where:

- msg points to the first parameter in an ML_END terminated array of parameters
- param is the 64-bit unique identifier of the parameter to be found.

`mlPvFind` returns the address of the parameter if successful; otherwise it returns `NULL`.

Obtaining Parameter Capabilities

All objects in ML are referred to via 64–bit identifying numbers. For example, the 64–bit ID number for the system running the application is `ML_SYSTEM_LOCALHOST`. Details on the interpretation of a particular device-dependent parameter are obtained using:

```
MLstatus mlPvGetCapabilities(MLint64 objectId, MLint64 parameterId, MLpv** capabilities);
```

where:

- `objectId` is the 64-bit unique identifier for the object whose parameter is being queried. An example is the `openId` returned from a call to `mlOpen`. The status `ML_STATUS_INVALID_ID` is returned if the specified object ID was invalid.
- `parameterId` is the 64-bit unique identifier for the parameter whose capabilities are being queried. The status `ML_STATUS_INVALID_ARGUMENT` is returned if the capabilities pointer is invalid.
- `capabilities` is a pointer to the head of the resulting capabilities list. This list should be treated as read-only by the application. If the call was successful, then the status `ml_STATUS_NO_ERROR` is returned.
- `objectid` may be either a static ID (obtained from a previous call to `mlGetCapabilities`) or an open ID (obtained by calling `mlOpen`). Querying the capabilities of an opened object is identical to querying the capabilities of the corresponding static object.

  It is also possible to get the capabilities of the capabilities parameters themselves. Those parameters are not tied to any particular object and so the `objectId` should be 0.
- `capabilities` contains the following parameters, though not necessarily in this order. Table 3-10 is a shortened form of the full parameter name.

Table 3-10 Parameters Returned by `mlPvGetCapabilities`

Parameter	Type	Description
`ID`	`INT64`	Resource ID for this parameter.
`NAME`	`BYTE_ARRAY`	`NULL`-terminated ASCII name of this parameter. This is identical to the enumerated value. For example, if the value is `ML_XXX`, then the name is `"ML_XXX"`.
`PARENT_ID`	`INT64`	Resource ID for the logical device (video path or transcoder pipe) on which this parameter is used.
`PARAM_TYPE`	`INT32`	Type of this parameter: • `ML_TYPE_BYTE` • `ML_TYPE_BYTE_ARRAY` • `ML_TYPE_BYTE_POINTER` • `ML_TYPE_INT32` • `ML_TYPE_INT32_ARRAY` • `ML_TYPE_INT32_POINTER` • `ML_TYPE_INT64` • `ML_TYPE_INT64_ARRAY` • `ML_TYPE_INT64_POINTER` • `ML_TYPE_REAL32` • `ML_TYPE_REAL32_ARRAY` • `ML_TYPE_REAL32_POINTER` • `ML_TYPE_REAL64` • `ML_TYPE_REAL64_ARRAY` • `ML_TYPE_REAL64_POINTER`
`PARAM_ACCESS`	`INT32`	Access controls for this parameter. Bitwise "or" of the following flags: • `ML_ACCESS_DURING_TRANSFER` • `ML_ACCESS_IMMEDIATE` (use in set/get) • `ML_ACCESS_OPEN_OPTION` • `ML_ACCESS_PASS_THROUGH` (ignored by device) • `ML_ACCESS_QUEUED` (use in send/query) • `ML_ACCESS_READ` • `ML_ACCESS_SEND_BUFFER` (only in `mlSendBuffers`) • `ML_ACCESS_WRITE`

Parameter	Type	Description
PARAM_DEFAULT	Same type as param	Default value for this parameter of type ML_PARAM_TYPE. This parameter may be of length 0 if there is no default.
PARAM_MINS	Array of same type as param	Array of minimum values for this parameter (may be missing if there are no specified minimum values). Each set of minimum and maximum values defines one allowable range of values. If the minimum equals the maximum, the allowable range is a single value. If the length component is 1, there is only one legal range of values. The length component will be 0 if there are no specified minimum values.
PARAM_MAXS	Array of same type as param	Array of maximum values for this parameter. There must be one entry in this array for each entry in the PARAM_MINS array.
PARAM_INCREMENT	Same type as param	Legal param values go from a minimum to a maximum in increments. The length will be 0 if there are no specified minimum values. Otherwise, length will be non-zero.
PARAM_ENUM_VALUES	Same type as param	Array of enumerated values for this parameter. The length component will be 0 if there are no enumeration values.
PARAM_ENUM_NAMES	BYTE_ARRAY	Array of enumeration names for this parameter (must have the same length as the PARAM_ENUM_VALUES array). The length of the parameter is the total length of all of the strings, including the NULL separators, and the double-NULL terminator.

Freeing Capabilities Lists

To return a capabilities list (obtained from either mlGetCapabilities or mlPvGetCapabilities) to the system, use the following:

```
MLstatus mlFreeCapabilities (MLpv *capabilities);
```

where capabilities is the capabilities list

Return status:

ML_STATUS_INVALID_ARGUMENT	Capabilities pointer is invalid
ML_STATUS_NO_ERROR	The call was successful

Audio/Visual Paths

In ML, the logical connections between jacks and memory are called *paths*. For example, a video output path provides the means to transfer video information from buffers in memory through a video output jack.

This chapter discusses the following:

- "Opening a Logical Path" on page 45
- "Constructing a Message" on page 46
- "Processing Out-of-Band Messages" on page 46
- "Sending In-Band Messages" on page 47
- "Processing In-Band Messages" on page 48
- "Processing Exception Events" on page 49
- "Processing In-Band Reply Messages" on page 50
- "Beginning and Ending Transfers" on page 51
- "Closing a Logical Path" on page 51

Opening a Logical Path

Before you send messages to a device, you must open a processing path that goes through it. This is done by calling `mlOpen`(3dm) as follows:

```
MLstatus mlOpen (MLint64 pathId, MLpv* options,
                    MLopenid* openid);
```

where:

- `objectId` is the 64-bit unique identifier for the path object to be opened. Obtain these identifiers by calling `mlGetCapabilities`.
- `options` is a pointer to a list of optional parameters. This list may be `NULL`. Obtain these parameters by calling `mlGetCapabilities`.

- `openid` is the resulting open device handle. Use this to refer to the open instance of the path object.

Think of a path as a logical device; a physical device (for example, a PCI card) may simultaneously support several such paths. A side effect of opening a path is that space is allocated for queues of messages from your application to the device and replies from the device back to your application. All of the messages sent to a queue share a common payload area and are required to observe a strictly ordered relationship. That is, if message A is sent before message B, then the reply to A must arrive before the reply to B.

Constructing a Message

Messages are arrays of parameters, where the last parameter is always `ML_END`. For example, set the image width to be 720 and the image height to be 480 as follows:

```
MLpv message[3];
message[0].param = ML_IMAGE_WIDTH_INT32;
message[0].value.int32 = 720;
message[1].param = ML_IMAGE_HEIGHT_INT32;
message[1].value.int32 = 480;
message[2].param = ML_END
```

Processing Out-of-Band Messages

In some cases, an application wishes to influence a device without first waiting for all previously enqueued messages to be processed. These cases are known as *out-of-band messages*. They are performed with the `mlSetControls`(3dm) or `mlGetControls`(3dm) calls.

Following is an example of how you can immediately get the width and height of an image:

```
MLpv message[3];
message[0].param = ML_IMAGE_WIDTH_INT32;
message[1].param = ML_IMAGE_HEIGHT_INT32;
message[2].param = ML_END
if( mlGetControls( somePath, message))
   handleError();
else
```

```
printf("Image size is %d x %d\n",
        message[0].value.int32,
        message[1].value.int32);
```

where:

- `somePath` is `pathID` from a previously opened path
- `message` is an array of parameter/value (`MLpv`) pairs

Out-of-band messages work well for simple sets and queries. They are blocking calls. If the call succeeds, the message has been successfully processed.

Sending In-Band Messages

Out-of-band messages are appropriate for simple control changes, but they provide no buffering between your application and the device. For most applications, processing real-time data will require using a queuing communication model. ML supports this with the following calls:

`mlSendControls`(3dm)
`mlQueryControls`(3dm)
`mlSendBuffers`(3dm)
`mlReceiveMessage`(3dm)

For example, to send a controls message to a device input queue, use the following:

```
MLstatus mlSendControls( MLopenid openId, MLpv* message);
```

where:

- `openId` is a previously-opened digital media object
- `message` is an array of parameter/value (`MLpv`) pairs

Devices interpret messages in the order in which they are enqueued. Because of this, the time relationship is explicit between, for example, video buffers and changes in video modes.

Note: The `mlSendControls` and `mlSendBuffers` calls do not wait for a device to process the message. Rather, they copy it to the device input queue and then return.

When your application sends a message, it is copied into the send queue. The message is then split between a small fixed header on the input list and a larger, variable-sized space in the data area.

Sometimes, there is not enough space in the data area and/or send list for new messages. In that case, the return code indicates that the message was not enqueued. As a rule, a full input queue is not a problem — it simply indicates that the application is generating messages faster than the device can process them.

For some devices, the system may use device-specific knowledge to best manage messaging transactions. For example, when you call `mlSendBuffers`, the system may copy the message exactly as described above, or it may send part or all of the message directly to the hardware. Regardless of what happens, the system always looks to your application as described here.

Each message you send is guaranteed to result in at least one reply message from the device, allowing you know when your message is interpreted and what is the result:

- In the case of control parameters, you should check the return message to make sure your control executed correctly.
- In the case of video buffers, you should allocate buffer space in your application and then send an indirect reference to that buffer in a message. Once your application receives a reply message, you can be certain the device has completed your request and finished with the memory, so you are free to reuse it.

Some devices can send messages to advise your application of important events (for example, some video devices can notify you of every vertical retrace). However, no notification messages will be generated unless you explicitly request them.

Processing In-Band Messages

The device processes messages as follows:

1. Removes the message header from the send queue.
2. Processes the message and writes any response into the payload area.
3. Places a reply header on the receive queue.

In general, your application must allow space in the message for any reply you expect to be returned.

Note: The device performs no memory allocation, but rather uses the memory allocated when the application enqueued the input message. This guarantees that there will never be any need for the device to block because it did not have enough space for the reply.

Processing Exception Events

In some cases, an exception event occurs that requires the device to pass a message back to your application. Your application must explicitly ask for such events.

Possible exception events are:

`ML_EVENT_AUDIO_SAMPLE_RATE_CHANGED`

The audio input sampling frequency changed.

`ML_EVENT_AUDIO_SEQUENCE_LOST`

An audio buffer was not available for an I/O transfer.

`ML_EVENT_DEVICE_ERROR`

Device encountered an error and is unable to recover.

`ML_EVENT_DEVICE_UNAVAILABLE`

The device is not available for use.

`ML_EVENT_VIDEO_SEQUENCE_LOST`

A video buffer was not available for an I/O transfer.

`ML_EVENT_VIDEO_SIGNAL_GAINED`

Device detected a valid input video signal.

`ML_EVENT_VIDEO_SIGNAL_LOST`

Device lost the video input signal.

`ML_EVENT_VIDEO_SYNC_GAINED`

Device detected a valid output genlock.

`ML_EVENT_VIDEO_SYNC_LOST`

Device lost the output genlock sync signal.

`ML_EVENT_VIDEO_VERTICAL_RETRACE`

A video vertical retrace occurred.

If you ask for events, your application must read its receive queue frequently enough to prevent the device from running out of space for messages that you have asked it to enqueue. If the queue starts to fill up, then the device will enqueue an event message advising that it is stopping notification of exception events.

Note: The device never needs to allocate space in the data area for reply messages. It will automatically stop sending notifications of events if the output list starts to fill up. Space is reserved in the receive queue for a reply to every message your application enqueues. If there is insufficient space, attempts to send new messages will fail.

Processing In-Band Reply Messages

To receive a reply message from a device, use `mlReceiveMessage`(3dm) as follows:

```
MLstatus mlReceiveMessage(MLopenid openId,
                          MLint32* messageType,
                          MLpv** reply);
```

where:

- `openId` is a previously-opened digital media object.
- `messageType` indicates why this reply was generated. It could come from the following:
  - A call to `mlSendControls`, `mlQueryControls`, or `mlSendBuffers`
  - Generated spontaneously by the device as the result of an event
- `reply` is a pointer that is guaranteed to remain valid until you attempt to receive a subsequent message. This allows a small optimization — you can read the current message in place without first copying it off the queue. It is acceptable to overwrite a value in a reply message and then send that as a new message.

This call returns the earliest unread message sent from the device back to your application.

Beginning and Ending Transfers

Devices do not begin to process enqueued messages until explicitly instructed to by an application with the `mlBeginTransfer`(3dm) call:

```
MLstatus mlBeginTransfer( MLopenid openId);
```

This call frees the device to begin processing enqueued messages. It also commands the device to begin generating exception events. Typically, an application will open a device, enqueue several buffers (priming the input queue), and then call `mlBeginTransfer`. In this way, it avoids the underflow that could otherwise occur if the application were swapped out immediately after enqueueing the first buffer to the device.

To stop a transfer, call `mlEndTransfer`(3dm):

```
MLstatus mlEndTransfer( mlopenid openId);
```

This causes the device to do the following:

- Stop processing messages containing buffers
- Flush its input queue
- Stop notification of exception events

Closing a Logical Path

When your application has finished using an open path, it may close it by using the following:

```
MLstatus mlClose( MLopenId openId);
```

This causes an implicit `mlEndTransfer` on any device with an active transfer. It then frees any resources used by the device. If you wish to have pending messages processed prior to closing a device, you must identify a message (perhaps by adding a piece of user data or by remembering its MSC number) and make sure it is the last thing you enqueue. When it appears on the output queue, you will know all messages have been processed. At that point, you can close the device.

For more information, see the `mlClose`(3dm) man page.

Transcoders

A *transcoder* provides a means to process data in memory. Support for transcoders may be implemented entirely in software, or it may be performed with hardware assistance. In either case, the software interfaces are consistent.

Each ML transcoder device consists of the following:

- A transcoder engine that performs the actual processing
- A number of source pipes and destination pipes

The engine takes data from buffers in the source pipes, processes it, and stores the result in buffers in the destination pipes. Each pipe acts much like a path that provides two things:

- A way for your application to send buffers containing bits to be processed
- A way to send empty buffers to hold the results of that processing

This chapter discusses the following:

- "Finding a Suitable Transcoder" on page 54
- "Opening a Logical Transcoder" on page 54
- "Controlling the Transcoder" on page 54
- "Sending Buffers" on page 55
- "Starting a Transfer" on page 56
- "Changing Controls During a Transfer" on page 56
- "Receiving a Reply Message" on page 57
- "Ending Transfers" on page 57
- "Closing a Transcoder" on page 58
- "Work Functions" on page 58

Finding a Suitable Transcoder

Use `mlGetCapabilities`(3dm) to obtain details of all transcoders on the system:

```
MLstatus mlGetCapabilities(MLint64 objectId, MLpv** capabilities);
```

where:

- `objectId` is the 64-bit identifier for the object whose capabilities are being queried
- `capabilities` is a pointer to the head of the resulting capabilities list

Opening a Logical Transcoder

Open a transcoder in much the same way as a path, but using `mlOpen`(3dm):

```
MLstatus mlOpen (MLint64 xcodeId, MLpv* options, MLopenid* openid);
```

When a transcoder is opened, it creates any required source and destination pipes. Just as for a path, an open transcoder is a logical entity — as such, a single physical device may support several transcoders simultaneously.

Controlling the Transcoder

The transcoder engine is controlled indirectly through the source and destination pipes:

- Controls on the source pipe describe what you will be sending the transcoder for input
- Controls on the destination pipe describe the results you want

The difference between the source and destination controls dictates what operations the transcoder should perform.

For example, if `ML_IMAGE_COMPRESSION_INT32` is `ML_COMPRESSION_UNCOMPRESSED` on the source and `ML_COMPRESSION_DVCPRO50_525` on the destination, then you are requesting the transcoder to do the following:

- Take uncompressed data from the source pipe
- Apply a `ML_COMPRESSION_DVCPRO50_525` compression

- Write the results to the destination pipe

To set controls on a transcoder, construct a controls message as you would for a video path. The only difference is that you must explicitly direct controls to a particular pipe. This is done through the `ML_SELECT_ID` parameter, which directs all following controls to a particular ID (in this case, the ID of a pipe on the transcoder).

Example 5-1 shows code to set the image width and height on both the source and destinations pipes.

Example 5-1 Set Image Width/Height on Pipes

```
msg[0].param = ML_SELECT_ID_INT64;
msg[0].value.int64 = ML_XCODE_SRC_PIPE;
msg[1].param = ML_IMAGE_WIDTH_INT32;
msg[1].value.int32 = 1920;
msg[2].param = ML_IMAGE_HEIGHT_INT32;
msg[2].value.int32 = 1080;
msg[3].param = ML_SELECT_ID_INT64;
msg[3].value.int64 = ML_XCODE_DST_PIPE;
msg[4].param = ML_IMAGE_WIDTH_INT32;
msg[4].value.int32 = 1920;
msg[5].param = ML_IMAGE_HEIGHT_INT32;
msg[5].value.int32 = 1280;
msg[6].param = ML_END;

mlSetControls(someOpenXcode, msg);
```

Sending Buffers

After the controls on a pipe have been set, you may begin to send buffers to it for processing. Do this with the `mlSendBuffers`(3dm) call:

```
MLstatus mlSendBuffers(MLopenid openid, MLpv* buffers);
```

where:

- `openid` is a previously-opened digital media object
- `buffers` is a list of parameter/value (`MLpv`) pairs

Call `mlSendBuffers` once for all the buffers corresponding to a single instant in time. For example, if the transcoder expects both an image buffer and an audio buffer, you must send both in a single `sendBuffers` call.

Example 5-2 shows code to send a source buffer to the source pipe and a destination buffer to the destination pipe.

Example 5-2 Send Source/Destination Buffers to Source/Destination Pipes

```
msg[0].param = ML_SELECT_ID_INT64;
msg[0].value.int64 = ML_XCODE_SRC_PIPE;
msg[1].param = ML_IMAGE_BUFFER_POINTER;
msg[1].value.pByte = srcBuffer;
msg[1].length = srcImageSize;
msg[2].param = ML_SELECT_ID_INT64;
msg[2].value.int64 = ML_XCODE_DST_PIPE;
msg[3].param = ML_IMAGE_BUFFER_POINTER;
msg[3].value.pByte = dstBuffers;
msg[3].maxLength = dstImageSize;
msg[4].param = ML_END;

mlSendBuffers(someOpenXcode, msg);
```

Starting a Transfer

The `mlSendBuffers` call places buffer messages on a pipe queue to the device. You must then call `mlBeginTransfer`(3dm) to tell the transcoder engine to start processing messages.

Note: The `mlBeginTransfer` call may fail if the source and destination pipe settings are inconsistent.

Changing Controls During a Transfer

During a transfer, you could attempt to change controls by using `mlSetControls`(3dm), but this is often undesirable because the effect of the control change on buffers currently being processed is undefined. A better method is to send control changes in the same queue as the buffer messages. Do this with the same

`mlSendControls`(3dm) call as on a path, again using `ML_SELECT_ID` to direct particular controls to a particular pipe.

Note: Parameter changes sent with `mlSendControls` are guaranteed to only affect buffers sent with subsequent send calls.

Some hardware transcoders may be unable to accommodate control changes during a transfer. If in doubt, examine the capabilities of particular parameter to determine if it may be changed while a transfer is in progress.

In a transcoder, it is possible to get the following exception event:

`ML_EVENT_XCODE_FAILED`

Transcoder was unable to process data.

Receiving a Reply Message

Whenever you pass buffer pointers to the transcoder by calling `mlSendBuffers`, you give up all rights to that memory until the transcoder has finished using it. As the transcoder finishes processing each buffers message, it will enqueue a reply message back to your application. You may read these reply messages in exactly the same way as on a path by calling `mlReceiveMessage`(3dm).

The transcoder queue maintains a strict first-in, first-out ordering. If buffer A is sent before buffer B, then the reply to A will come before the reply to B. This is guaranteed even on transcoders that parallelize across multiple physical processors.

Ending Transfers

To end a transfer, call `mlEndTransfer`(3dm). This is a blocking call that does the following:

- Allows all buffers currently in the engine to complete
- Marks any remaining messages as `aborted`

By examining the reply to each message, your application can determine whether or not it was successfully processed.

It is also acceptable to call `mlEndTransfer` before `mlBeginTransfer` has been called. In that case, any messages in the queue are aborted and returned to the application.

Note: If you are not interested in the result of any pending buffers, you can simply close the transcoder without bothering to first end the transfer.

Closing a Transcoder

When your application has finished using a transcoder it may close it:

```
MLstatus mlClose( MLopenId openId);
```

This causes an implicit `mlEndTransfer`. It then frees any resources used by the device.

See the `mlClose`(3dm) man page.

Work Functions

In most cases, the difference between hardware and software transcoders is transparent to an application. Software transcoders may have more options and may run more slowly, but for many applications these differences are not significant.

One notable difference between hardware and software transcoders is that software transcoders will attempt to use as much of the available processor time as possible. This may be undesirable for some applications. To counter this, an application has the option to do the work of the transcoder itself, in its own thread. This is achieved with the `mlXcodeWork`(3dm) function.

If you open a software transcoder while setting the `ML_XCODE_MODE_SYNCHRONOUS` option, the transcoder will not spawn any threads and will not do any processing on its own. To perform a unit of transcoding work, your application must now call the `mlXcodeWork`(3dm) function.

Note: This only applies to software transcoders, and only if you set the `ML_XCODE_MODE_SYNCHRONOUS` option when opening the transcoder.

Video Parameters

The processing of a video input/output path is described by two sets of parameters:

- *Video parameters* describe how to interpret and generate the signal as it arrives and leaves, as discussed in this chapter
- *Image parameters* describe how to write/read the resulting bits to/from the device (see Chapter 7, "Image Buffer Parameters" on page 71)

Not all parameters may be supported on a particular video jack or path. Some parameters may be adjusted on both a path and a jack, or may be adjusted on just one or the other. Use `mlGetCapabilities` to obtain a list of parameters supported by a jack or path. In addition, not all values may be supported on a particular parameter. Use `mlPvGetCapabilities` to obtain a list of the values supported by the parameter.

This chapter contains the following sections:

- "Temporal Video Sampling" on page 59
- "Video Parameter Descriptions" on page 61
- "Video Example" on page 70

Note: This chapter assumes a working knowledge of digital video concepts. Readers unfamiliar with terms such as *video timing*, *422*, or *CbYCr* should consult a text devoted to this subject. A good resource is *A Technical Introduction to Digital Video* by Charles Poynton, published by John Wiley & Sons, 1996 (ISBN 0-471-12253-X, hardcover).

Temporal Video Sampling

There are two kinds of video sampling, spatial and temporal. Our concern here is with temporal sampling, of which there are two techniques:

- *Progressive sampling* is frame-based (for example, from film)
- *Interlaced sampling* is field-based

Progressive Sampling

In progressive, frame-based sampling, a picture at a specified resolution is sampled at a constant rate. Film is a progressive sampling source for video.

Imagine an automatic film advance camera that can take 60 pictures per second, with which you take a series of pictures of a moving ball. Figure 6-1 on page 60 shows 10 pictures from that sequence (different colors emphasize the different positions of the ball in time). The time delay between each picture is a 60th of a second, so this sequence lasts 1/6th of a second.

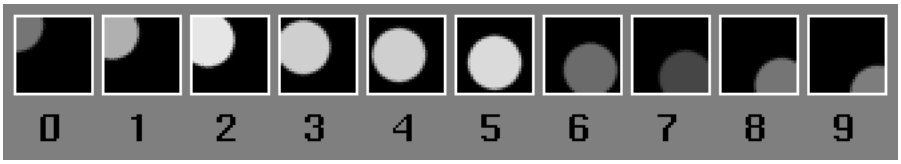

Figure 6-1 Progressive Sampling: Film at 60 Frames-per-Second

Interlaced Sampling

In interlaced sampling, the video is sampled periodically at two sample fields, F1 and F2, such that half of the display lines of the picture are scanned at a time.

Pairs of sample fields are superimposed on each other (*interlaced*) to create the *video frame*. In the video frame, the sample frames appear coincident to the eye even though they are consecutive. This effect is aided by the persistence of phosphors on the display screen that hold the impression of the first set of scanned lines as the second set displays. (For example, this sequence is made visible if you videotape a computer monitor display.)

Most video signals in use today, including several high-definition video formats, are field-based (interlaced) rather than frame-based (progressive). In ML, the value of the video timing parameter `ML_VIDEO_TIMING_INT32` defines the specific video standard, and each standard is defined as progressive or interlaced.

For example, suppose you shoot the moving ball with an NTSC video camera. NTSC video has 60 fields-per-second, so you might think that the video camera would record the same series of pictures as shown in Figure 6-1 on page 60, but it does not. The video camera does record 60 images per second, but each image consists of only

half of the scanned lines of the complete picture at a given time, as shown in Figure 6-2 on page 61, rather than a filmstrip of 10 complete images.

Note how the image lines alternate between odd- and even-numbered images.

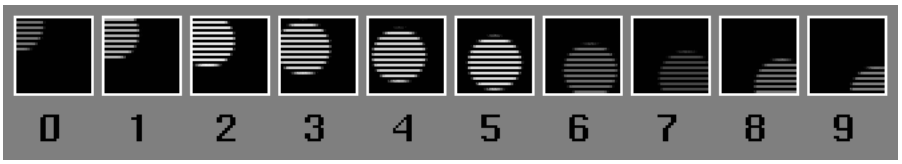

Figure 6-2 Interlaced Sampling: Video at 60 Frames-per-Second

Video Parameter Descriptions

This section describes the video parameters.

ML_VIDEO_ALPHA_SETUP_INT32

Sets or gets the video signal alpha channel setup.

ML_VIDEO_BLUE_SETUP_INT32

Sets or gets the video signal blue channel setup.

ML_VIDEO_BRIGHTNESS_INT32

Sets or gets the video signal brightness.

ML_VIDEO_COLORSPACE_INT32

Sets the colorspace at the video jack. For input paths, this is the colorspace you expect to receive at the jack. For output paths, it is the colorspace you desire at the jack.

The following colorspace values are supported:

`ML_COLORSPACE_RGB_601_FULL`

```
ML_COLORSPACE_RGB_601_HEAD
ML_COLORSPACE_CbYCr_601_FULL
ML_COLORSPACE_CbYCr_601_HEAD
ML_COLORSPACE_RGB_240M_FULL
ML_COLORSPACE_RGB_240M_HEAD
ML_COLORSPACE_CbYCr_240M_FULL
ML_COLORSPACE_CbYCr_240M_HEAD
ML_COLORSPACE_RGB_709_FULL
ML_COLORSPACE_RGB_709_HEAD
ML_COLORSPACE_CbYCr_709_FULL
ML_COLORSPACE_CbYCr_709_HEAD
```

See "ML_IMAGE_COLORSPACE_INT32" on page 74 for a detailed description of colorspace values.

ML_VIDEO_CONTRAST_INT32

Sets or gets the video signal contrast.

ML_VIDEO_DITHER_FILTER_INT32

Sets or gets the video signal dither filter.

ML_VIDEO_FILL_ALPHA_REAL32

Describes the alpha value for any pixel outside the clipping region. This is a real number: a value of `0.0` is the minimum (fully transparent), `1.0` is the maximum (fully opaque). Default is `1.0`.

ML_VIDEO_FILL_BLUE_REAL32

Describes the blue value for any pixel outside the clipping region. This is a real number: a value of `0.0` is the minimum (fully transparent), `1.0` is the maximum (fully opaque). Default is `1.0`.

ML_VIDEO_FILL_Cb_REAL32

Describes the Cb value for any pixel outside the clipping region. This is a real number: a value of `0.0` is the minimum legal value, `1.0` is the maximum legal value. Default is `0`.

ML_VIDEO_FILL_Cr_REAL32

Describes the Cr value for any pixel outside the clipping region. This is a real number: a value of `0.0` is the minimum legal value, `1.0` is the maximum legal value. Default is `0`.

ML_VIDEO_FILL_GREEN_REAL32

Describes the green value for any pixel outside the clipping region. This is a real number: a value of `0.0` is the minimum legal value, `1.0` is the maximum legal value. Default is `0`.

ML_VIDEO_FILL_RED_REAL32

Describes the red value for any pixel outside the clipping region. This is a real number: a value of `0.0` is the minimum legal value (black), `1.0` is the maximum legal value. Default is `0`.

ML_VIDEO_FILL_Y_REAL32

Describes the luminance value for any pixel outside the clipping region. This is a real number: a value of `0.0` is the minimum legal value (black), `1.0` is the maximum legal value. Default is `0`.

ML_VIDEO_FLICKER_FILTER_INT32

Sets or gets the video signal filter.

ML_VIDEO_GENLOCK_SIGNAL_PRESENT_INT32

Queries the incoming genlock signal for an output path. Not all devices may be able to sense genlock timing, but those that do will support this parameter. Common values match those for ML_VIDEO_TIMING listed in "Video Parameter Descriptions" on page 61, plus the following:

ML_TIMING_NONE

There is no signal present

ML_TIMING_UNKNOWN

The timing of the genlock cannot be determined

ML_VIDEO_GENLOCK_SOURCE_TIMING_INT32

Describes the genlock source timing. Only accepted on output paths. Each genlock source is specified as an output timing on the path and corresponds to the same timings as available with ML_VIDEO_TIMING_INT32.

ML_VIDEO_GENLOCK_TYPE_INT32

Describes the genlock signal type. Only accepted on output paths. Each genlock type is specified as either a 32-bit resource ID or ML_VIDEO_GENLOCK_TYPE_INTERNAL.

ML_VIDEO_GREEN_SETUP_INT32

Sets or gets the video signal green channel setup.

ML_VIDEO_H_PHASE_INT32

Sets or gets the video signal horizontal phase genlock offset.

ML_VIDEO_HEIGHT_F1_INT32

Sets the vertical height for each F1 field of the video signal. For progressive signals, it specifies the height of every frame.

ML_VIDEO_HEIGHT_F2_INT32

Sets the vertical height for each F2 field of the video signal. For progressive signals, it always has value `0`.

ML_VIDEO_HUE_INT32

Sets or gets the video signal hue.

ML_VIDEO_INPUT_DEFAULT_SIGNAL_INT64

Set or get the video signal default input signal.

ML_VIDEO_NOTCH_FILTER_INT32

Sets or gets the video signal notch filter.

ML_VIDEO_OUTPUT_DEFAULT_SIGNAL_INT64

Sets the default signal at the video jack when there is no active output. The following values are supported:

`ML_SIGNAL_BLACK`

Output will generate a black picture complete with legal synchronization values

`ML_SIGNAL_COLORBARS`

Output will use an internal colorbar generator

`ML_SIGNAL_INPUT_VIDEO`

Output will use the default input signal as a pass-through

`ML_SIGNAL_NOTHING`

No output signal with legal synchronization values will be generated

ML_VIDEO_OUTPUT_REPEAT_INT32

Determines the device behavior if the application is doing output and fails to provide buffers fast enough (that is, the queue to the device underflows). Allowable options are:

`ML_VIDEO_REPEAT_FIELD`

The device repeats the last field. For progressive signals or interleaved formats, this is the same as `ML_VIDEO_REPEAT_FRAME`.

`ML_VIDEO_REPEAT_FRAME`

The device repeats the last two fields. This output capability is device dependent and the allowable settings should be queried via the get capabilities of the `ML_VIDEO_OUTPUT_REPEAT_INT32` parameter.

`ML_VIDEO_REPEAT_NONE`

The device does nothing, usually resulting in black output.

ML_VIDEO_PRECISION_INT32

Sets the precision (number of bits of resolution) in the signal at the jack. This is an integer. Example values are as follows:

`8`	8-bit signal
`10`	10-bit signal

ML_VIDEO_RED_SETUP_INT32

Sets or gets video signal red channel setup.

ML_VIDEO_SAMPLING_INT32

Sets the sampling at the video jack. (See "ML_IMAGE_SAMPLING_INT32" on page 82 for a detailed description of sampling values.)

The following values are supported:

ML_SAMPLING_422
ML_SAMPLING_4224
ML_SAMPLING_444
ML_SAMPLING_4444

ML_VIDEO_SATURATION_INT32

Sets or gets the video signal color saturation.

ML_VIDEO_SIGNAL_PRESENT_INT32

Used to query the incoming signal on an input path. Not all devices may be able to sense timing, but those that do will support this parameter. Common values match those for ML_VIDEO_TIMING listed in "Video Parameter Descriptions" on page 61, plus the following:

ML_TIMING_NONE

There is no signal present

ML_TIMING_UNKNOWN

The timing of the input signal cannot be determined

ML_VIDEO_START_X_INT32

Sets the start horizontal location on each line of the video signal.

ML_VIDEO_START_Y_F1_INT32

Sets the start vertical location on F1 fields of the video signal. For progressive signals, it specifies the start of every frame.

ML_VIDEO_START_Y_F2_INT32

Sets the start vertical location on F2 fields of the video signal. Ignored for progressive timing signals.

ML_VIDEO_TIMING_INT32

Sets the timing on an input or output video path. Not all timings may be supported on all devices. On devices that can auto-detect, the timing may be read-only on input. (Details of supported timings may be obtained by calling `mlPvGetCapabilites` on this parameter). Figure B-1 on page 144 and Figure B-2 on page 145 illustrate details of the 601 standard.

Note: See Appendix B, "Common Video Standards" on page 143 for diagrams of common video standards.

The format is as follows:

`ML_TIMING_`*xxxx*`_`*yyyy*`x`*zzzz*`_`*nnn*`[i|p|PsF]`

where:

xxxx	Total number of lines.
yyyy `x` *zzzz*	Width by height of the active video region (high definition).
nnn`[i|p|PsF]`	The frame rate, followed by one of the following: • `i` (interlaced) • `p` (progressive) • `PsF` (segmented frame)

Standard Definition (SD) Timings

The following SD timings are supported:

`ML_TIMING_525` (NTSC)
`ML_TIMING_525_SQ_PIX`
`ML_TIMING_625` (PAL)
`ML_TIMING_625_SQ_PIX`

High Definition (HD) Timings

The following HD timings are supported:

```
ML_TIMING_1125_1920x1080_60p
ML_TIMING_1125_1920x1080_5994p
ML_TIMING_1125_1920x1080_50p
ML_TIMING_1125_1920x1080_60i
ML_TIMING_1125_1920x1080_5994i
ML_TIMING_1125_1920x1080_50i
ML_TIMING_1125_1920x1080_30p
ML_TIMING_1125_1920x1080_2997p
ML_TIMING_1125_1920x1080_25p
ML_TIMING_1125_1920x1080_24p
ML_TIMING_1125_1920x1080_2398p
ML_TIMING_1125_1920x1080_24PsF
ML_TIMING_1125_1920x1080_2398PsF
ML_TIMING_1125_1920x1080_30PsF
ML_TIMING_1125_1920x1080_2997PsF
ML_TIMING_1125_1920x1080_25PsF
ML_TIMING_1250_1920x1080_50p
ML_TIMING_1250_1920x1080_50i
ML_TIMING_1125_1920x1035_60i
ML_TIMING_1125_1920x1035_5994i
ML_TIMING_750_1280x720_60p
ML_TIMING_750_1280x720_5994p
```

ML_VIDEO_V_PHASE_INT32

Sets or gets the video signal vertical phase genlock offset.

ML_VIDEO_WIDTH_INT32

Sets the horizontal width of the clipping region on each line of the video signal.

Video Example

Following is an example that sets the video timing and colorspace for an HDTV signal:

```
MLpv message[3]
message[0].param = ML_VIDEO_TIMING_INT32
message[0].value.int32 = ML_TIMING_1125_1920x1080_5994p;
message[1].param = ML_VIDEO_COLORSPACE_INT32;
message[1].value.int32 = ML_COLORSPACE_CbYCr_709_HEAD;
message[2].param = ML_END;
mlSetControls( device, message);
```

Image Buffer Parameters

This chapter describes in detail the ML image buffer parameters and gives examples of the resulting in-memory pixel formats:

- "Image Buffer Layouts" on page 71
- "Image Buffer Parameters Summary" on page 73

Note: This chapter assumes a working knowledge of digital video concepts. Readers unfamiliar with terms such as *video timing*, *422*, or *CbYCr* should consult a text devoted to this subject. A good resource is *A Technical Introduction to Digital Video* by Charles Poynton, published by John Wiley & Sons, 1996 (ISBN 0-471-12253-X, hardcover).

Image Buffer Layouts

An *image buffer* is memory allocated for a frame or field of pixels. Because ML itself does not allocate memory for buffers, the application must do the allocation. This means that each buffer requires a dedicated memory allocation call (`malloc`, for example.)

Buffers must be in contiguous virtual memory and should be pinned in memory for optimum performance. Once a buffer has been created, the pointer to the buffer is passed to ML with the parameter `ML_IMAGE_BUFFER_POINTER`. The buffer pointer points to the first byte of the image in memory and must comply with the alignment constraints for buffers on the particular path or transcoder to which it is being sent. See the `mlGetCapabilities`(3dm) man page for details on determining alignment requirements with `ML_PATH_BUFFER_ALIGNMENT_INT32`.

For example, if `ML_PATH_BUFFER_ALIGNMENT_INT32` is `8`, this means that the value of the buffer pointer must be a multiple of 8 bytes. The same applies to `ML_PATH_COMPONENT_ALIGNMENT_INT32`, where the beginning of each line (the first pixel of each line) must be a multiple of the value of the `ML_PATH_COMPONENT_ALIGNMENT_INT32` parameter.

Figure 7-1 shows an image that is mapped into a image buffer in a very general form. Figure 7-2 shows the more common simple image buffer layout.

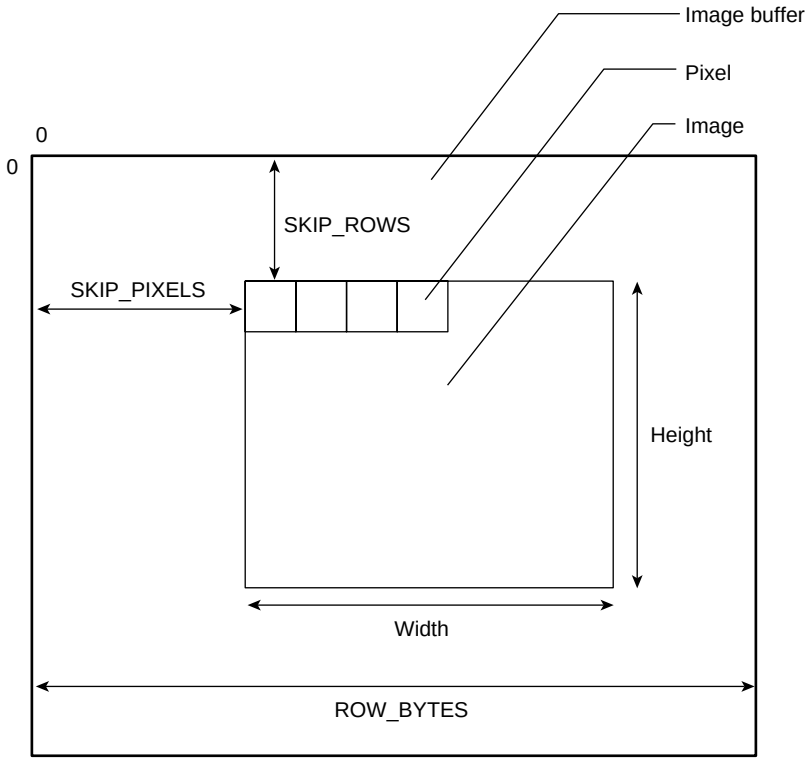

Figure 7-1 General Image Buffer Layout

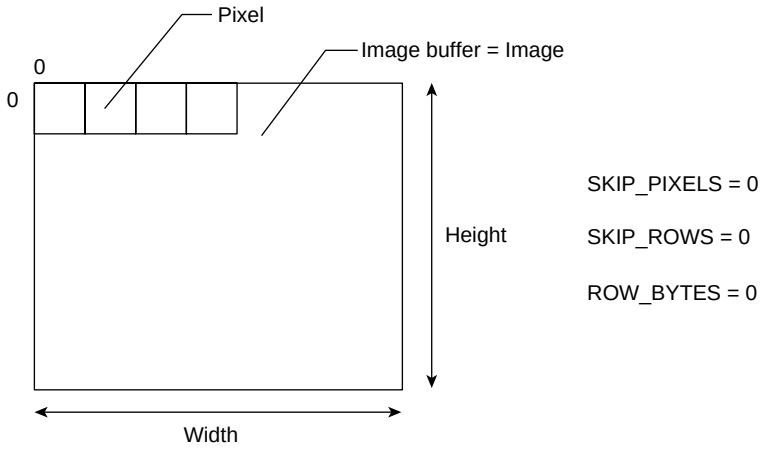

Figure 7-2 Simple Image Buffer Layout

Image Buffer Parameters Summary

This section describes the image buffer parameters.

ML_IMAGE_BUFFER_POINTER

Sets the pointer to the first byte of an image buffer in memory. The buffer address must comply with the alignment constraints for buffers on the particular path or transcoder to which it is being sent. See the `mlGetCapabilities`(3dm) man page for details on determining alignment requirements with `ML_PATH_BUFFER_ALIGNMENT_INT32`.

For example, if `ML_PATH_BUFFER_ALIGNMENT_INT32` is `8`, this means that the value of the buffer pointer must be a multiple of 8 bytes. The same applies to `ML_PATH_COMPONENT_ALIGNMENT_INT32`, where the beginning of each line (the first pixel of each line) must be a multiple of the value of the `ML_PATH_COMPONENT_ALIGNMENT_INT32` parameter.

ML_IMAGE_BUFFER_SIZE_INT32

Sets the size of the image buffer in bytes. This is a read-only parameter and is computed in the device using the current path control settings. This value represents the worst-case buffer size.

ML_IMAGE_COLORSPACE_INT32

Describes how to interpret each component. The full colorspace parameter is:

`ML_COLORSPACE_`*representation_standard_range*

where:

- *representation* is either `ML_REPRESENTATION_RGB` or `ML_REPRESENTATION_CbYCr`.

  This controls how to interpret each component. Table 7-1 shows this mapping (assuming that every component is sampled once per pixel).

Table 7-1 Mapping Colorspace *representation* Parameters

Colorspace Representation	Component 1	Component 2	Component 3	Component 4
RGB	Red	Green	Blue	Alpha
CbYCr	Cb	Y	Cr	Alpha

The packing dictates the size and order of the components in memory, while the colorspace describes what each component represents. For example, the following shows the effect of colorspace and packing combined (assuming a 4444 sampling, see "`ML_IMAGE_SAMPLING_INT32`" on page 82).

```
Color                    31            int             0
Standard  Packing        +-----------------------------+
RGB       10_10_10_2     RRRRRRRRRRGGGGGGGGGGBBBBBBBBBBAA
RGB       10_10_10_2_R   AABBBBBBBBBBGGGGGGGGGGRRRRRRRRRR
CbYCr     10_10_10_2     bbbbbbbbbbYYYYYYYYYYrrrrrrrrrrAA
CbYCr     10_10_10_2_R   AAbbbbbbbbbbYYYYYYYYYYrrrrrrrrrr
```

- *standard* indicates how to interpret particular values as actual colors. Choosing a different standard alters the way the system converts between different color representations. The current standards supported are Rec. 601, Rec. 709, and SMPTE 240M.
- *range* is one of the following:
  - `FULL`, where the smallest and largest values are limited only by the available packing size. This is common in computer graphics.
  - `HEAD`, where the smallest and largest values are somewhat less than the theoretical min/max values to allow some "headroom". This is common in video, particularly when sending video signals over a wire. For example, values outside the legal component range may be used to mark the start or end of a video frame.

In Rec. 601 video, the black level (blackest black) is 16 for 8–bit video and 64 for 10–bit video. In computer graphics, 0 is blackest black. If a picture with 16 for blackest black is displayed by a system that uses 0 as blackest black, the image colors are all grayed-out as a result of shifting the colors to this new scale. Similarly, the brightest level is 235 for 8–bit video and 940 for 10–bit video. The best results are obtained by choosing the correct colorspace.

Example 7-1 `ML_COLORSPACE_RGB_709_FULL`

`ML_COLORSPACE_RGB_709_FULL` is shorthand for the following:

```
ML_REPRESENTATION_RGB
+
ML_STANDARD_709
+
ML_RANGE_FULL
```

where:

- Representation is RGB
- The standard is 709
- Full-range data is used

ML_IMAGE_COMPRESSION_FACTOR_REAL32

Describes desired compression factor (for compressed images only). The values are as follows:

1	Disables the compressor or puts the compressor in pass-through mode.
x	Indicates that approximately *x* compressed buffers require the same space as 1 uncompressed buffer.

Note: The size of the uncompressed buffer depends on image width, height, packing, and sampling. The default value is implementation-dependent, but should represent a reasonable trade-off between compression time, quality, and bandwidth. *x* is a number larger than 1.

ML_IMAGE_COMPRESSION_INT32

Sets the input buffer or desired output buffer format on an image compressor/decompressor (codec). Possible values are as follows:

```
ML_COMPRESSION_UNCOMPRESSED
ML_COMPRESSION_BASELINE_JPEG
ML_COMPRESSION_DV_625
ML_COMPRESSION_DV_525
ML_COMPRESSION_MPEG2I
ML_COMPRESSION_DVCPRO_625
ML_COMPRESSION_DVCPRO_525
ML_COMPRESSION_DVCPRO50_625
ML_COMPRESSION_DVCPRO50_525
ML_COMPRESSION_MPEG2
```

Note: In case of a compressed bit stream, all parameters that describe the image data (such as height, width, and color space) might not be known. The only parameters that might be known are the compression type `ML_IMAGE_COMPRESSION_INT32` and the size of the bit stream `ML_IMAGE_BUFFER_SIZE_INT32`. The image buffer layout parameters (`ML_IMAGE_SKIP_ROWS`, `ML_IMAGE_SKIP_PIXELS`, and `ML_IMAGE_ROW_BYTES`) do not apply to compressed images.

For more infomation, see the following:

- JPEG: W. B. Pennebaker and J. L. Mitchell, *JPEG: Still Image Data Compression Standard*, New York, NY: Van Nostrand Reinhold, 1993.
- IEC 61834-1 (1997). *Recording - Helical-Scan Digital Video Cassette Recording System Using 6.35 mm Magnetic Tape for Consumer Use (525-60, 625-50, 1125-60, and 1250-50 Systems) - Part 1: General Specifications*
- IEC 61834-2 (1997). *Recording - Helical-Scan Digital Video Cassette Recording System Using 6.35 mm Magnetic Tape for Consumer Use (525-60, 625-50, 1125-60, and 1250-50 Systems) - Part 2: SD Format for 525-60 and 625-50 Systems*
- *SMPTE 314M Television - Data Structure for DV-Based Audio, Data and Compressed Video - 25 and 50 Mb/s* THE SOCIETY OF MOTION PICTURE AND TELEVISION ENGINEERS, 1997
- MPEG2: *ISO/IEC 13818-2 GENERIC CODING OF MOVING PICTURES AND ASSOCIATED AUDIO: SYSTEMS.*

ML_IMAGE_DOMINANCE_INT32

Sets the field dominance, which defines the order of fields in a frame. For the same sequence of fields, there are two valid interpretations about which of the two fields belong together:

- *F1–dominant* is an F1 field followed by an F2 field
- *F2-dominant* is an F2 field followed by an F1 field

Changing the field dominance is most significant when external devices (for example, a tape deck) can only operate on frame boundaries. Figure 7-3 describes field dominance.

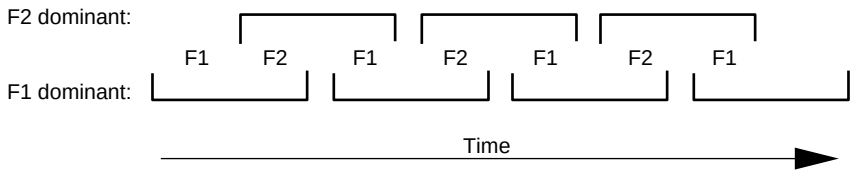

Figure 7-3 Field Dominance

The parameter can be one of the following:

`ML_DOMINANCE_F1`

Specifies that the video signal is F1–dominant. This is the default.

`ML_DOMINANCE_F2`

Specifies that the video signal is F2-dominant.

This parameter is ignored for progressive signals.

ML_IMAGE_HEIGHT_1_INT32

Represents one of the following:

- The height of each frame for progressive or interleaved buffers (depending on parameter `ML_IMAGE_INTERLEAVE_MODE_INT32`)
- The height of each F1 field, measured in pixels, for interlaced and non-interleaved signals.

Note: *Interlaced* pertains to video signals while *interleaved* pertains to images in memory. Progressive video signals are noninterlaced and noninterleaved. An interlaced signal could be either interleaved (a frame will have 2 fields in a buffer) or noninterleaved (a frame will have 2 buffers, with each field in a buffer).

For more information, see *A Technical Introduction to Digital Video* by Charles Poynton, published by John Wiley & Sons, 1996 (ISBN 0-471-12253-X, hardcover.

ML_IMAGE_HEIGHT_2_INT32

Sets the height of each F2 field in an interlaced non-interleaved signal. For progressive video signals and interlaced interleaved buffers, it must be set to value `0`.

ML_IMAGE_INTERLEAVE_MODE_INT32

Specifies whether the two fields have been interleaved into a single image (and reside in a single buffer) or are stored in two separate fields (hence in two separate buffers). This parameter is only used in interlaced images.

Possible values are:

`ML_INTERLEAVE_MODE_INTERLEAVED`

Each pair of fields is interleaved into a single buffer. In this case, the parameter `ML_IMAGE_HEIGHT_2_INT32` is set to `0`.

`ML_INTERLEAVE_MODE_SINGLE_FIELD`

The two fields are stored separately. This means that each field has its own image buffer. Use `ML_IMAGE_HEIGHT_1_INT32` for the F1 buffer and `ML_IMAGE_HEIGHT_2_INT32` for the F2 buffer.

This parameter is ignored for signals with progressive timing. Default is interleaved.

ML_IMAGE_ORIENTATION_INT32

Sets the orientation of the image:

`ML_ORIENTATION_TOP_TO_BOTTOM`

Natural video order pixel `[0,0]` is at the top left of the image

`ML_ORIENTATION_BOTTOM_TO_TOP`

Natural graphics order pixel `[0,0]` is at the bottom left of the image

ML_IMAGE_PACKING_INT32

Sets the image packing. The image packing parameter describes the pixel storage in detail as follows:

`ML_PACKING_`*type_size_order*

where:

- *type* is the base type of each component. Leave blank for an unsigned integer, use `S` for a signed integer.
- *size* defines the number of bits per component. The size may refer to simple, padded, or complex packings.

For the simplest formats, every component is the same size and there is no additional space between components. A single numeric value specifies the number of bits per component. The first component consumes the first *size* bits, the next consumes the next *size* bits, and so on. Within each component, the most significant bits always precede the least-significant bits. For example, a size of `12` means that the first byte in memory has the most significant 8 bits of the first component, the second byte holds the remainder of the first component and the most significant 4 bits of the second component, and so on.

Space is only allocated for components that are in use (this depends on the sampling mode, see "`ML_IMAGE_SAMPLING_INT32`" on page 82). For these formats, the data must always be interpreted as a sequence of bytes. For example, `ML_PACKING_8` describes a packing in which each component is an unsigned 8-bit quantity. `ML_PACKING_S8` describes the same packing except that each component is a signed 8-bit quantity.

For padded formats, each component is padded and may be treated as a short 2-byte integer. When this occurs, the *size* takes the form:

Bits`in`*SpaceAlignment*

where:

Bits	Specifies the number of bits of space per component
Space	Specifies the total size of each component
Alignment	Indicates whether the information is left–shifted (`L`) or right-shifted (`R`) in that space

In this case, each component in use consumes *space* bits and those bits must be interpreted as a short integer. (Unused components consume no space).

For example, following are some common packings:

```
          15  int  short 0
Packing   +--------------+
12in16R   0000iiiiiiiiiiii
S12in16R  ssssiiiiiiiiiiii
12in16L   iiiiiiiiiiiipppp
S12in16L  iiiiiiiiiiiipppp
S12in16L0 iiiiiiiiiiii0000
```

where:

– `s` indicates sign-extension.

- `i` indicates the actual component information.
- `p` indicates padding (replicated from the most significant bits of information).
- `S` indicates a signed number (for 12 bits: -2048 ... 2047). The range is: -2**(*nbits*-1) .. 2**(*nbits*-1) - 1.
- `0` indicates that unused bits are padded with zeros.

Note: These bit locations refer to the locations when the 16–bit component has been loaded into a register as a 16-bit integer quantity.

For the most complex formats, the size of every component is specified explicitly and the entire pixel must be treated as a single 4–byte integer. The *size* takes the form *size1_size2_size3_size4*, where *size1* is the size of component 1, *size2* is the size of component 2, and so on. In this case, the entire pixel is a single 4–byte integer of length equal to the sum of the component sizes. Any space allocated to unused components must be zero-filled. The most common complex packing occurs when 4 components are packed within a 4-byte integer. For example, `ML_PACKING_10_10_10_2` is:

```
            31             int             0
Packing     +-------------------------------+
10_10_10_2  11111111112222222222333333333344
```

where `1` is the first component, `2` is the second component, and so on. The bit locations refer to the locations when this 32–bit pixel is loaded into a register as a 32–bit integer quantity. If only three components were in use (determined from the sampling), then the space for the fourth component would be zero-filled.

- *order* is the order of the components in memory. Leave blank for natural ordering (1,2,3,4), use `R` for reversed ordering (4,3,2,1). For all other orderings, specify the component order explicitly. For example, `4123` indicates that the fourth component is stored first in memory, followed by the remaining three components. Here, we compare a normal, a reversed, and a 4123 packing:

```
             31             int             0
Packing      +-------------------------------+
10_10_10_2   11111111112222222222333333333344
10_10_10_2_R 44333333333322222222222111111
10_10_10_2_4123 44111111111122222222223333333333
```

where 1 is the first component, 2 is the second component, and so on. Because this is a complex packing, the bit locations refer to the locations when this entire pixel is loaded into a register as a single integer.

For recommendations on packing and component ordering, see Appendix A, "Pixels in Memory" on page 133.

ML_IMAGE_ROW_BYTES_INT32

Specifies the number of bytes along one row of the image buffer. If this value is 0, each row is exactly ML_IMAGE_WIDTH_INT32 pixels wide. Default is 0.

Note: In physical memory, there is no notion of two dimensions; the end of the first row continues directly at the start of the second row. An image buffer contains either one frame or one field. For interlaced image data, the two fields can be stored in two separate image buffers or they can be stored in interleaved form in one image buffer.

ML_IMAGE_SAMPLING_INT32

Specifies the sampling rate. The sampling parameters take their names from common terminology in the video industry. They describe how often each component is sampled for each pixel. In computer graphics, it is normal for every component to be sampled once per pixel, but in video that need not be the case.

For RGB colorspaces, the only legal values are:

RGB Value	Description
ML_SAMPLING_444	Indicates that the R, G, and B components are each sampled once per pixel, and only the first 3 channels are used. If used with an image packing that provides space for a fourth component, then those bits should have value 0 on an input path and will be ignored on an output path.
ML_SAMPLING_4444	Indicates that the R, G, B, and A components are sampled once per pixel.

For all CbYCr colorspaces, the legal values include the following:

CbYCr Value	Description
ML_SAMPLING_444	Indicates that Cb, Y, and Cr are each sampled once per pixel and only the first 3 channels are used. If a packing provides space for a fourth channel then those bits should have value 0.
ML_SAMPLING_4444	Indicates that Cb, Y, Cr, and Alpha are each sampled once per pixel.
ML_SAMPLING_422	Indicates that Y is sampled once per pixel and Cb/Cr are sampled once per pair of pixels. In this case, Cb and Cr are interleaved on component 1 (Cb is first, Cr is second) and the Y occupies component 2. If used with an image packing that provides space for a third or fourth component, those bits should have value 0 on an input path and will be ignored on an output path.
ML_SAMPLING_4224	Indicates that Y and Alpha are sampled once per pixel and Cb/Cr are sampled once per pair of pixels. In this case, Cb and Cr are interleaved on component 1, Y is on component 2, component 3 contains the Alpha channel, and component 4 is not used (and will have value 0 if space is allocated for it in the packing).
ML_SAMPLING_411	Indicates that Y is sampled once per pixel and Cb and Cr are sampled once per 4 pixels. In this case, Cb is component 1, Y is component 2, and Cr is component 3. If used with an image packing that provides space for a fourth component, those bits should have value 0 on an input path and will be ignored on an output path.
ML_SAMPLING_420	Indicates that Y is sampled once per pixel and Cb or Cr is sampled once per pair of pixels on alternate lines. In this case, Cb or Cr is interleaved on component 1 and the Y occupies component 2. If used with an image packing that provides space for a third or fourth component, those bits should have value 0 on an input path and will be ignored on an output path.
ML_SAMPLING_400	Indicates that only Y is sampled per pixel (a greyscale image). Y is stored on component 1, all other

components are unused. If used with an image packing that provides space for additional components, those bits should have value `0` on an input path and will be ignored on an output path.

`ML_SAMPLING_0004` — Indicates that only Alpha is sampled per pixel. If used with an image packing that provides space for additional components, those bits should have value `0` on an input path and will be ignored for an output path.

Table 7-2 shows the combined effect of sampling and colorspace on the component definitions.

Table 7-2 Effect of Sampling and Colorspace on Component Definitions

Sampling	Colorspace Representation	Component 1	Component 2	Component 3	Component 4
4444	RGB	Red	Green	Blue	Alpha
444	RGB	Red	Green	Blue	
0004	RGB	Alpha	Y	Cr	Alpha
444	CbYCr	Cb	Y	Cr	0
4224	CbYCr	Cb/Cr	Y	Alpha	0
422	CbYCr	Cb/Cr	Y		
400	CbYCr	Y			
420	CbYCr	Cb/Cr [1]	Y		
411	CbYCr	Y	Cr		
0004	CbYCr	Alpha			

ML_IMAGE_SKIP_PIXELS_INT32

Specifies the number of pixels to skip at the start of each line in the image buffer. Must be `0` if `ML_IMAGE_ROW_BYTES_INT32` is `0`. Default is `0`.

1 Cb and Cr components are multiplexed with Y on alternate lines (not pixels.)

ML_IMAGE_SKIP_ROWS_INT32

Specifies the number of rows to skip at the start of each image buffer. Default is `0`.

ML_IMAGE_TEMPORAL_SAMPLING_INT32

Specifies whether the image source is progressive or interlaced. Set to one of the following:

```
ML_TEMPORAL_SAMPLING_FIELD_BASED
ML_TEMPORAL_SAMPLING_PROGRESSIVE
```

Default is device-dependent.

If the image data is field based, the parameter `ML_IMAGE_INTERLEAVE_MODE_INT32` defines how the two fields are stored in an image buffer.

ML_IMAGE_WIDTH_INT32

Sets the width of the image in pixels.

ML_SWAP_BYTES_INT32

Note: Not available on all devices.

Sets whether or not byte reordering occurs:

`1`	Reorders bytes as a first step when reading data from memory and as a final step when writing data to memory. The exact reordering depends on the packing element size.
`0`	Does not reorder (default)

For simple and padded packing formats, the element size is the size of each component. For complex packing formats, the element size is the sum of the four component sizes. Table 7-3 describes how this parameter reorders bits.

Table 7-3 Bit Reordering

Element Size	Default Ordering	Modified Ordering
16-bit	[15..0]	[7..0][15..8]
32-bit	[31..0]	[7..0][15..8][23..16][31..24]
Other	[*n*..0]	[*n*..0] (no change)

Audio Parameters

This chapter describes the ML audio parameters and buffers:

- "Audio Buffer Layout" on page 87
- "Audio Parameters Summary" on page 89
- "Uncompressed Audio Buffer Size Computation" on page 93

Audio Buffer Layout

The digital representation of an audio signal is generated by periodically sampling the amplitude (voltage) of the audio signal. The samples represent periodic snapshots of the signal amplitude. The sampling rate specifies the number of samples per second. The audio buffer pointer points to the source or destination data in an audio buffer for processing a fragment of a media stream. For audio signals, a fragment typically corresponds to between 10 milliseconds and 1 second of audio data. An audio buffer is a collection of sample frames. A *sample frame* is a set of audio samples that are coincident in time. A sample frame for mono data is a single sample. A sample frame for stereo data consists of a left-right sample pair.

Stereo samples are interleaved; left-channel samples alternate with right-channel samples. Four-channel samples are also interleaved, with each frame usually having two left/right sample pairs, but there can be other arrangements.

Figure 8-1 shows the relationship between the number of channels and the frame size of audio sample data. Figure 8-2 shows the layout of an audio buffer in memory.

Figure 8-1 Different Audio Sample Frames

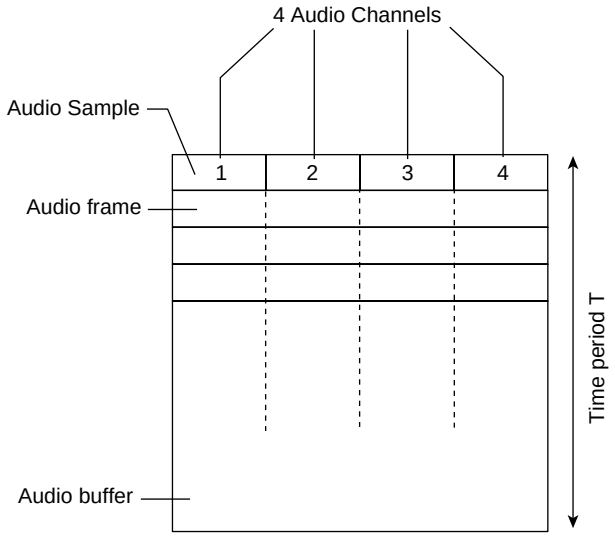

Figure 8-2 Layout of an Audio Buffer with 4 Channels

Audio Parameters Summary

This section discusses the audio parameters.

ML_AUDIO_BUFFER_POINTER

Sets a pointer to the first byte of an in-memory audio buffer. The buffer address must comply with the alignment constraints for buffers on the particular path to which it is being sent. See the `mlGetCapabilities`(3dm) man page for details of determining alignment requirements.

ML_AUDIO_CHANNELS_INT32

Sets the number of channels of audio data in the buffer. Multichannel audio data is always stored interleaved, with the samples for each consecutive audio channel

following one another in sequence. For example, a 4-channel audio stream will have the form:

```
123412341234...
```

where: `1` is the sample for the first audio channel, `2` is for the second, and so on.

Common values are:

```
ML_CHANNELS_MONO
ML_CHANNELS_STEREO
ML_CHANNELS_4
ML_CHANNELS_8
```

ML_AUDIO_COMPRESSION_INT32

Specifies the compression format if the audio data is in compressed form. The compression format may be an industry standard such as MPEG-1 audio, or it may be no compression at all.

Common values are:

```
ML_COMPRESSION_A_LAW
ML_COMPRESSION_AC3
ML_COMPRESSION_IMA_ADPCM
ML_COMPRESSION_MPEG1
ML_COMPRESSION_MPEG2
ML_COMPRESSION_MU_LAW
ML_COMPRESSION_UNCOMPRESSED
```

ML_AUDIO_FORMAT_INT32

Specifies the format in which audio samples are stored in memory. The interpretation of format values is as follows:

`ML_FORMAT_`*TypeBits*

- *Type* is `U` for unsigned integer samples, `S` for signed (2's compliment) integer samples, `R` for real (floating point) samples
- *Bits* is the number of significant bits per sample

For sample formats in which the number of significant bits is less than the number of bits in which the sample is stored, the format of the values is:

ML_FORMAT_*TypeBits*in*SizeAlignment*

- *Size* is the total size used for the sample in memory, in bits.
- *Alignment* is either R or L depending on whether the significant bits are right- or left-shifted within the sample. For example, following are three of the most common audio buffer formats:

  ML_AUDIO_FORMAT_U8

  ```
  7 char 0
  +------+
  iiiiiiii
  ```

  ML_AUDIO_FORMAT_S16

  ```
  15  short int  0
  +--------------+
  iiiiiiiiiiiiiiii
  ```

  ML_AUDIO_FORMAT_S24in32R

  ```
  31            int              0
  +------------------------------+
  ssssssssiiiiiiiiiiiiiiiiiiiiiiii
  ```

where:

- s indicates sign-extension
- i indicates the actual component information

The bit locations refer to the locations when the 8-, 16-, or 32-bit sample has been loaded into a register as an integer quantity. If the audio data compression parameter ML_AUDIO_COMPRESSION_INT32 indicates that the audio data is in compressed form, the ML_AUDIO_FORMAT_INT32 indicates the data type of the samples after decoding. Common formats are:

```
ML_FORMAT_U8
ML_FORMAT_S16
ML_FORMAT_S24in32R
ML_FORMAT_R32
```

Default is hardware-specific.

ML_AUDIO_FRAME_SIZE_INT32

Sets the size of an audio sample frame in bytes. This is a read-only parameter and is computed in the device using the current path control settings.

ML_AUDIO_GAINS_REAL64_ARRAY

Sets the gain factor in decibels (dB) on the given path. There will be a value for each audio channel. Negative values represent attenuation. Zero represents no change of the signal. Positive values amplify the signal. A gain of negative infinity indicates infinite attenuation (mute).

ML_AUDIO_PRECISION_INT32

Queries the maximum width in bits for an audio sample at the input or output jack. For example, a value of 16 indicates a 16-bit audio signal.
`ML_AUDIO_PRECISION_INT32` specifies the precision at the audio I/O jack, whereas `ML_AUDIO_FORMAT_INT32` specifies the packing of the audio samples in the audio buffer. If `ML_AUDIO_FORMAT_INT32` is different than `ML_AUDIO_PRECISION_INT32`, the system will convert between the two formats. Such a conversion might include padding and/or truncation.

ML_AUDIO_SAMPLE_RATE_REAL64

Sets the sample rate of the audio data in Hz. The sample rate is the frequency at which samples are taken from the analog signal. Sample rates are measured in hertz (Hz). A sample rate of 1 Hz is equal to one sample per second. For example, when a mono analog audio signal is digitized at a 44.1-kilohertz (kHz) sample rate, 44,100 digital samples are generated for every second of the signal. Values are dependent on the hardware, but are usually between 8,000.0 and 96,000.0. Default is hardware-specific. Common sample rates are:

8,000.0
16,000.0
32,000.0
44,100.0

48,000.0
96,000.0

The Nyquist theorem defines the minimum sampling frequency required to accurately represent the information of an analog signal with a given bandwidth. According to Nyquist, digital audio information is sampled at a frequency that is at least double the highest interesting analog audio frequency. The sample rate used for music-quality audio, such as the digital data stored on audio CDs, is 44.1 kHz. A 44.1-kHz digital signal can theoretically represent audio frequencies from 0 kHz to 22.05 kHz, which adequately represents sounds within the range of normal human hearing. Higher sample rates result in higher-quality digital signals; however, the higher the sample rate, the greater the signal storage requirement.

Uncompressed Audio Buffer Size Computation

The following equation shows how to calculate the number of bytes for an uncompressed audio buffer given the sample frame size, sampling rate, and the time period representing the audio buffer:

$N = F \cdot R \cdot T$

where:

N	Audio buffer size in bytes
F	The number of bytes per audio sample frame (`ML_AUDIO_FRAMESIZE_INT32`)
R	The sample rate in Hz (`ML_AUDIO_SAMPLE_RATE_REAL64`)
T	The time period the audio buffer represents in seconds

Example 8-1 Buffer Size Computation

If:

- *F* is 4 bytes (if *packing* is S16 and there are two channels)
- *R* (sample rate) is 44,100 Hz
- *T* = 40 ms = 0.04 s.

then the resulting buffer size (*N*) is 7056 bytes.

ML Processing

The ML library is concerned with two types of interfaces:

- Paths for digital media through jacks into and out of the machine
- Pipes for digital media to and from transcoders

Both types of interfaces share common control, buffer, and queueing mechanisms. These mechanisms are first described in the context of a complete program example. Subsequently, the individual functions are presented.

This chapter contains the following sections:

- "ML Program Structure" on page 96
- "`MLstatus` Return Value" on page 97
- "Device States" on page 99
- "Opening a Jack, Path, or Transcoder" on page 100
- "Set Controls" on page 102
- "Get Controls" on page 103
- "Send Controls" on page 104
- "Send Buffers" on page 106
- "Query Controls" on page 109
- "Get Wait Handle" on page 111
- "Begin Transfer" on page 112
- "Transcoder Work" on page 113
- "Get Message Count" on page 114
- "Receive Messsage" on page 115
- "End Transfer" on page 116
- "Close Processing" on page 116

- "Utility Functions" on page 117
- "Example: Printing the Interpretation of a Video Timing Parameter" on page 122

ML Program Structure

ML programs are composed of the following structure. Each of the functions are described later in this chapter (except where noted).

```
// Get list of available media devices:
mlGetCapabilities( systemId, &capabilities );

// Search the devices to find the desired jack, path, or transcoder to open (see Chapter 7):
mlGetCapabilities( deviceId, &capabilities );

// Query the jack, path or transcoder to discover allowable open options and parameters (see Chapter 7):
mlGetCapabilities( objectId, &capabilities );

// Query for individual parameter characteristics (see Chapter 7):
mlPvGetCapabilities( deviceId, paramId, &capabilities );

// Free memory associated with any of the above get-capabilities (see Chapter 7):
mlFreeCapabilities( capabilities );

// Open a logical connection to the desired object:
mlOpen( objectId, options, &openid );

// Get and set any necessary immediate controls:
mlGetControls( openId, controls );
mlSetControls( openId, controls );

// Send any synchronous controls:
mlSendControls( openId, controls );

// Send buffers to device:
mlSendBuffers( opendId, buffers );

// Prepare for asynchronous processing by getting wait handle:
mlGetWaitHandle( openId, &waitHandle );
```

```
// Start the path or transcoder transfer:
mlBeginTransfer( openId );

// Perform synchronous transcoder work (if applicable):
mlXcodeWork( openId );

// Check on the status of the queue:
mlGetSendMessageCount( openId, &sendMsgCount );
mlGetReceiveMessageCount( openId, &receiveMsgCount );

// Process return messages:
mlReceiveMessage( openId, &status, &replyMsg );

// Find specific returned parameters:
mlPvFind( replyMsg, param );

//Repeat mlSendControls, mlSendBuffers, mlXcodeWork, etc. as required, then
// stop the transfer:
mlEndTransfer( openId );

// Close the logical connection:
mlClose( openId );

// Other useful functions:
mlGetVersion( &major, &minor );
mlGetSystemUST( systemId, &UST );
statusName = mlStatusName( status );
msgName = mlMessageName( messageType );
```

`MLstatus` Return Value

Note: All ML API functions return an `MLstatus` value. This provides a consistent error-checking interface. (However, certain convenience functions do not adhere to this standard.)

`ML_STATUS_ACCESS_DENIED`

The requested open access conflicts with a previous access already established or the requested parameter cannot be modified during the current operation mode.

`ML_STATUS_DEVICE_UNAVAILABLE`

The requested device has become unavailable, possibly by being powered down or removed from the system.

`ML_STATUS_INSUFFICIENT_RESOURCES`

Not all the resources required to complete the operation are available.

`ML_STATUS_INTERNAL_ERROR`

An operation was aborted due to a system or device I/O error.

`ML_STATUS_INVALID_ARGUMENT`

One of the arguments in the function call is invalid.

`ML_STATUS_INVALID_CONFIGURATION`

Because control messages may be incomplete, and individual set or send controls may be valid, there exists a point in time where the processing of buffers must be accomplished using those aggregate controls. If the combination of controls is invalid, the processing is aborted and the `ML_STATUS_INVALID_CONFIGURATION` error (for `mlSetControls`) or the event (for `mlSendControls`) is returned.

`ML_STATUS_INVALID_ID`

One of the arguments representing an ID is invalid.

`ML_STATUS_INVALID_PARAMETER`

The specified parameter (`param` field) is invalid for the requested operation.

`ML_STATUS_INVALID_VALUE`

The value of a parameter is invalid.

`ML_STATUS_NO_ERROR`

The operation succeeded without error.

`ML_STATUS_NO_OPERATION`

The function resulted in no operation.

`ML_STATUS_OUT_OF_MEMORY`

The operation was aborted due to lack of memory resources.

`ML_STATUS_RECEIVE_QUEUE_EMPTY`

The receive queue was empty when an `mlReceiveMessage` function was processed.

`ML_STATUS_RECEIVE_QUEUE_OVERFLOW`

The receive queue will not accept the return message if the current message is enqueued on the send queue.

`ML_STATUS_SEND_QUEUE_OVERFLOW`

Too many `mlSendControls` and/or `mlSendBuffers` have been issued.

Device States

For audio and video paths and transcoders, the device transitions through well-known states, known as *device states*. These states are as follows:

`ML_DEVICE_STATE_ABORTING`

The device has terminated message processing, usually by accepting an `mlEndTransfer`. All messages remaining on the input queue will be flushed to the output queue with the message type indicating that the message was aborted. Values are:

```
ML_BUFFERS_ABORTED
ML_CONTROLS_ABORTED
ML_QUERY_CONTROLS_ABORTED
```

`ML_DEVICE_STATE_FINISHING`

The device is terminating the transfer, but will complete processing of the remaining messages in the input queue.

`ML_DEVICE_STATE_READY`

The device is in a quiescent state and can accept messages, but will not process them until it enters the `ML_DEVICE_STATE_TRANSFERRING` state.

`ML_DEVICE_STATE_TRANSFERRING`

The device has accepted an `mlBeginTransfer` and is now processing messages.

`ML_DEVICE_STATE_WAITING`

The device is currently waiting for an external event such as the `ML_WAIT_FOR_AUDIO_MSC_INT64` predicate control. Messages may still be enqueued, but will not be processed until the wait condition is removed.

Opening a Jack, Path, or Transcoder

In order to communicate with a jack, path, or transcoder, a connection must be opened. A physical device (such as a PCI card) may simultaneously support several such connections. These connections are done by calling `mlOpen`:

```
MLstatus mlOpen (const MLint64 objectId, MLpv* options,MLopenid* openid);
```

where:

- `objectId` is the 64-bit unique identifier for the object (jack, path or transcoder) to be opened
- `options` specify the initial configuration of the device to be opened

These parameters are described in Table 3-7 on page 38.

The status return for this function is one of the following:

`ML_STATUS_ACCESS_DENIED`

The requested open access mode is not available.

`ML_STATUS_DEVICE_UNAVAILABLE`

The requested device has gone off-line, possibly by being disconnected.

`ML_STATUS_NO_ERROR`

The call succeeded and the handle of the open instance of the object has been returned in `openid`.

`ML_STATUS_OUT_OF_MEMORY`

Insufficient memory is available to perform the operation, including the space needed to allocate the queues for messages between the application and the device.

`ML_STATUS_INSUFFICIENT_RESOURCES`

Some other required resource is not available, possibly by being already in use by this or another application.

`ML_STATUS_INTERNAL_ERROR`

An operating system error has occurred.

`ML_STATUS_INVALID_ARGUMENT`

One of the arguments is otherwise invalid.

`ML_STATUS_INVALID_ID`

The argument `objectId` is invalid.

`ML_STATUS_INVALID_PARAMETER`

One of the parameters in the options list is invalid.

`ML_STATUS_INVALID_VALUE`

One of the parameters in the options list has an invalid value.

Set Controls

Some controls on a logical connection are asynchronous in nature and do not affect an ongoing data transfer. These controls may be set in an out-of-band message using the `mlSetControls` operation:

```
MLstatus mlSetControls( MLopenid openid, MLpv* controls );
```

where:

- `openid` is the 64-bit unique identifier returned by the `mlOpen` function
- `controls` is a message containing various parameters as described elsewhere in Chapter 2, "Parameters" on page 17

This call blocks until the device has processed the message. To identify an invalid value specification, the device will set the length component of the erroneous `MLpv` to `-1`, otherwise the controls array will not be altered in any way and may be reused. The controls message is not enqueued on the send queue but instead is sent directly to the device. The device will attempt to process the message as soon as possible.

Enqueueing entails a copy operation, so the application is free to delete/alter the message array as soon as the call returns. Any error return value indicates the control change has not been enqueued and will thus have no effect.

Note: The `mlSetControls` call returns as soon as the control array has been processed. This does not mean that buffers have been affected by the parameter change. Rather, it means that the parameters have been validated and sent to the device (that is, in most cases this means that they reside in registers).

The status return is one of the following:

`ML_STATUS_INVALID_ID`

The specified `openid` is invalid.

`ML_STATUS_INVALID_PARAMETER`

At least one of the parameters in the controls array was not recognized. (The first such offending control will be marked with length `-1`, remaining controls will be skipped and the entire message will be ignored.)

`ML_STATUS_INVALID_VALUE`

At least one of the parameters in the controls array has a value that is invalid. This may be because the parameter value is outside the legal range, or it may be that parameter value is inconsistent. (The entire message will be ignored and the system will attempt to flag the first offending value by setting the length to `-1`.)

`ML_STATUS_NO_ERROR`

The control values were set successfully.

Get Controls

Control on a logical connection may be queried asynchronously to an ongoing transfer:

```
MLstatus mlGetControls (MLopenid openid, MLpv* controls);
```

where:

- `openid` is the identifier (returned by `mlOpen`) of the jack, path, or transcoder whose parameters are to be queried.
- `controls` is a message consisting of parameters to be queried. The device will place its reply in the controls array argument (overwriting existing values). Control values that were obtained successfully will have nonnegative lengths. `mlGetControls` returns the state of the controls at the time the call is made.

If `mlGetControls` is called before a control has been explicitly set, then generally the returned value is undefined. (Exceptions are noted in the definitions the controls; see Chapter 10, "Synchronization" on page 123 or `ML_UST` in `mlSynchronization`(3dm).)

The status return for this function is one of the following:

`ML_STATUS_INVALID_ID`

The specified open device ID is invalid.

`ML_STATUS_INVALID_PARAMETER`

At least one of the parameters in the controls array was not recognized. (The offending control will be marked with length `-1`; remaining controls will still be processed.)

`ML_STATUS_INVALID_VALUE`

At least one of the parameters in the controls array has a value that is invalid. (The offending control will be marked with length `-1`; remaining controls will still be processed.)

`ML_STATUS_NO_ERROR`

The control values were obtained successfully.

Send Controls

Other controls on a logical connection are synchronous in nature and affect the processing of subsequent data buffers. These controls should be set in an in-band message using the `mlSendControls` operation:

```
MLstatus mlSendControls( MLopenid openid, MLpv* controls );
```

where:

- `openid` is the 64-bit unique identifier returned by the `mlOpen` function
- `controls` is a message containing various parameters as described in "`param/value` Pairs in an `MLpv` Message" on page 17

The `mlSendControls` operation sends a message containing a list of control parameters to a previously-opened digital media device. These controls are enqueued on the send queue in sequence with any other messages to that device. Any control changes are thus guaranteed not to have any effect until all previously enqueued messages have been processed.

This call returns as soon as the control change has been enqueued to the device. It does not wait until the control change has actually taken effect.

All the control changes within a single message are considered to occur atomically. If any one control change in the message fails, then the entire message has no effect. A successful return does not guarantee that resources will be available to support the requested control change at the time it is processed by the device.

As each message is processed by the device, a reply message will be enqueued for return to the application. By examining that reply, the application may obtain the result of attempting to process the requested controls.

Note: A device may take an arbitrarily long time to generate a reply (it may, for example, wait for several messages before replying to the first). If an application requires an immediate response, consider using the `mlSetControls` operation instead.

Enqueueing entails a copy operation, so the application is free to delete/alter the message array as soon as the call returns. Any error return value indicates the control change has not been enqueued and will thus have no effect.

The status return is one of the following:

`ML_STATUS_INVALID_ID`

The specified `openid` is invalid.

`ML_STATUS_INVALID_PARAMETER`

At least one of the parameters in the controls array was not recognized. (The first such offending control will be marked with length `-1`, remaining controls will be skipped and the entire message will be ignored.)

`ML_STATUS_INVALID_VALUE`

At least one of the parameters in the controls array has a value that is invalid. This may be because the parameter value is outside the legal range, or it may be that parameter value is inconsistent. (The entire message will be ignored and the system will attempt to flag the first offending value by setting the length to `-1`.)

`ML_STATUS_NO_ERROR`

The control values were set successfully.

`ML_STATUS_RECEIVE_QUEUE_OVERFLOW`

There is not currently enough space on the receive queue to hold the reply to this message. Read some replies from the receive queue and then try to send again, or specify a larger receive-queue size on open.

`ML_STATUS_SEND_QUEUE_OVERFLOW`

There was not enough space on the path send queue for this latest message. Try again later after the device has had time to catch up, or specify a larger send-queue size on open.

The return event may be one of the following:

`ML_CONTROLS_ABORTED`

The processing of the controls were aborted due to another asynchronous event, such as the `mlEndTransfer` function was requested.

`ML_CONTROLS_COMPLETE`

The controls were processed without error.

`ML_CONTROLS_FAILED`

The processing of the controls failed because the values were not accepted at the time of processing.

Send Buffers

The `mlSendBuffers` function sends a message containing a list of buffers to a previously-opened digital media device. These buffers are enqueued on the send queue in sequence with any other messages to that device. All the buffers within a single message are considered to apply to the same point in time. For example, a single buffers message could contain image, audio, HANC, and VANC buffers, each specified with its own buffer parameter in the buffers message. The `mlSendBuffers` operation is as follows:

```
MLstatus mlSendBuffers( MLopenid openid, MLpv* buffers );
```

where:

- `openid` refers to a previously-opened logical connection as returned from `mlOpen`
- `buffers` is a message containing a list of buffer parameters

As each message is processed by the path, a reply message will be enqueued for return to the application. By examining that reply, the application may obtain the result of attempting to process the buffers.

A successful return value from `mlSendBuffers` guarantees only that the requested buffers have been enqueued to the device. Any error return value indicates the buffers have not been enqueued and will thus have no effect.

The memory for the buffers is designated by the `pointer` value, and is always owned by the application. However, after a buffer has been sent, it is on loan to the system and must not be touched by the application. After the buffer has been returned via `mlReceiveMessage`, then the application is again free to delete and/or modify it.

When sending a buffer to be output, the application must set the buffer `length` to indicate the number of valid bytes in the buffer. In this case, `maxLength` is ignored by the device. (It does not matter how much larger the buffer may be because the device will not read past the last valid byte.)

When sending a buffer to be filled (on input) the application must set the buffer `maxLength` to indicate the maximum number of bytes that may be written by the device to the buffer. As the device processes the buffer, it will write no more than the `maxLength` bytes and then set the returned length to indicate the last byte written. The `maxLength` is returned without change. It is acceptable to send the same buffer multiple times.

Enqueueing entails a copy operation, so the application is free to delete/alter the message array as soon as the call returns. Any error return value indicates that the buffer has not been enqueued and will thus have no effect.

The status return is one of the following:

`ML_STATUS_INVALID_ID`

The specified `openid` is invalid.

`ML_STATUS_INVALID_PARAMETER`

At least one of the parameters in the message was not recognized. (The first such offending control will be marked with length `-1`, remaining controls will be skipped and the entire message will be ignored.)

`ML_STATUS_INVALID_VALUE`

At least one of the parameters in the message has a value that is invalid. This may be because the parameter value is outside the legal range, or it may be that parameter value is inconsistent. (The entire message will be ignored and the system will attempt to flag the first offending value by setting the length to `-1`.)

`ML_STATUS_NO_ERROR`

The buffers message was enqueued successfully.

`ML_STATUS_RECEIVE_QUEUE_OVERFLOW`

There is not currently enough space on the receive queue to hold the reply to this message. Read some replies from the receive queue and then try to send again, or specify a larger receive-queue size on open.

`ML_STATUS_SEND_QUEUE_OVERFLOW`

There was not enough space on the path send queue for this latest message. Try again later after the device has had time to catch up, or specify a larger send-queue size on open.

The return event may be one of the following:

`ML_BUFFERS_ABORTED`

The processing of the buffers was aborted due to another asynchronous event, such as the `mlEndTransfer` function was requested.

`ML_BUFFERS_COMPLETE`

The buffers were processed without error.

`ML_BUFFERS_FAILED`

The processing of the buffers failed because the values were not accepted at the time of processing. This can occur both because parameters in the buffers message were invalid or because the current control settings at the time of processing (due to previous `mlSendControls` messages), the processing of buffers would be invalid. Because preceding control messages may be incomplete, and each of the individual set or send controls may be valid, there still exists a point in time where the processing of buffers must be accomplished using those aggregate controls. If the combination of

controls is invalid, the processing is aborted and the event `ML_BUFFERS_FAILED` is returned.

Query Controls

To obtain the control values on a logical connection that are synchronous via an in-band message, use the `mlQueryControls` operation:

```
MLstatus mlQueryControls( MLopenid openid, MLpv* controls );
```

where:

- `openid` is the 64-bit unique identifier returned by the `mlOpen` function.
- `controls` is a message containing various parameters as described in "`param/value` Pairs in an `MLpv` Message" on page 17.
- `mlQueryControls` sends a message containing a list of control parameters to a previously-opened digital media device. These controls are enqueued on the send queue in sequence with any other messages to that device. The control values returned are thus guaranteed to reflect all previously enqueued `mlSendControls` messages that have been processed.

This call returns as soon as the message has been enqueued to the device. It does not wait until the control value is available.

As each message is processed by the device, a reply message will be enqueued for return to the application. By examining that reply, the application may obtain the result of attempting to process the requested controls.

Note: A device may take an arbitrarily long time to generate a reply (it may, for example, wait for several messages before replying to the first). If an application requires an immediate response, consider using the `mlGetControls` operation instead.

Enqueueing entails a copy operation, so the application is free to delete/alter the message array as soon as the call returns. Any error return value indicates the control change has not been enqueued and will thus have no effect.

The status return is one of the following:

`ML_STATUS_INVALID_ID`

The specified `openid` is invalid.

`ML_STATUS_INVALID_PARAMETER`

At least one of the parameters in the controls array was not recognized. (The first such offending control will be marked with length `-1`, remaining controls will be skipped and the entire message will be ignored.)

`ML_STATUS_NO_ERROR`

The control values were set successfully.

`ML_STATUS_RECEIVE_QUEUE_OVERFLOW`

There is not currently enough space on the receive queue to hold the reply to this message. Read some replies from the receive queue and then try to send again, or specify a larger receive-queue size on open.

`ML_STATUS_SEND_QUEUE_OVERFLOW`

There was not enough space on the path send queue for this latest message. Try again later after the device has had time to catch up, or specify a larger send-queue size on open.

The return event may be one of the following:

`ML_QUERY_CONTROLS_ABORTED`

The processing of the query controls were aborted due to another asynchronous event, such as the `mlEndTransfer` function was requested.

`ML_QUERY_CONTROLS_COMPLETE`

The query controls were processed without error.

Get Wait Handle

When processing a number of digital media streams asynchronously, there exists a need for the application to know when processing is required on each individual stream. The `mlGetSendWaitHandle` and `mlGetReceiveWaitHandle` functions are provided to facilitate this processing:

```
MLstatus mlGetSendWaitHandle( MLopenid openid, MLwaitable* WaitHandle );
MLstatus mlGetReceiveWaitHandle( MLopenid openid, MLwaitable* WaitHandle );
```

where:

- `openid` is a previously-opened digital media object as returned by a `mlOpen` call.
- `MLwaitable` on IRIX, UNIX, and Linux is a file descriptor for use in `select()`. On Windows, `MLwaitable` is a handle that may be used in the `win32` functions `WaitForSingleObject` or `WaitForMultipleObjects`.
- `WaitHandle` is the requested returned wait handle. This function returns an event handle on which an application may wait.

The send queue handle is signaled whenever the device dequeues a message and the message count drops below a preset level (set by the parameter `ML_OPEN_SEND_SIGNAL_COUNT` specified when the object was opened). Thus, if the send queue is full, an application may wait on this handle for notification that space is available for additional messages.

The receive queue handle is signaled whenever the device enqueues a reply message. Thus, if the receive queue is empty, the application may wait on this handle for notification that additional reply messages are ready.

The returned handles were created when the device was opened and are automatically destroyed when the path is closed.

The status return is one of the following:

`ML_STATUS_INVALID_ID`

The specified open device handle is invalid

`ML_STATUS_NO_ERROR`

The wait handle was obtained successfully

Begin Transfer

`mlBeginTransfer` starts the actual transferring of buffers to the logical media connection:

```
MLstatus mlBeginTransfer (MLopen id openid);
```

where `openid` is a previously-opened digital media object as returned by an `mlOpen` call.

This function begins a continuous transfer on the specified path or transcoder. It is not used on a logical connection to a jack. This call advises the device to begin processing buffers and returning messages to the application. As stated earlier, sending a buffer to a device that has not yet begun transfers will cause the send queue to stall until the transfers have started. Typically, applications will open a device, send several buffers, and then call `mlBeginTransfer`. This call returns as soon as the device has begun processing transfers. It does not block until the first buffer has been processed. It is an error to call this function more than once without an intervening call to `mlEndTransfer`.

Note: The delay between a call to `mlBeginTransfer` and the transfer of the first buffer is implementation-dependent. To begin sending data at a particular time, an application should start the transfer early (enqueueing blank buffers) and use the UST/MSC mechanism to synchronize the start of real data.

The status return is one of the following:

`ML_STATUS_INVALID_ID`

The specified open device ID is invalid

`ML_STATUS_NO_ERROR`

The device agreed to begin transfer on the path

`ML_STATUS_NO_OPERATION`

The call had no effect (transfers have already been started)

Transcoder Work

For software-only transcoders opened with the `ML_XCODE_MODE_INT32` open option set to `ML_XCODE_MODE_SYNCHRONOUS`, this function allows an application to control exactly when (and in which thread) the processing for that codec takes place:

```
MLstatus mlXcodeWork( MLopenid openid );
```

where `openid` refers to a previously-opened digital media transcoder.

This function performs one unit of processing for the specified codec. The processing is done in the thread of the calling process and the call does not return until the processing is complete. For most codecs, a unit of work is the processing of a single buffer from the source queue and the writing of a single resulting buffer on the destination queue.

Note: The default behavior for all codecs is for processing to happen automatically as a side effect of enqueueing messages to the device. This function only applies to software codecs and only applies if they are opened with the `ML_XCODE_MODE_SYNCHRONOUS` open option.

The status return is one of the following:

`ML_STATUS_INVALID_ID`

The specified `openid` is invalid

`ML_STATUS_NO_ERROR`

The software transcoder performed one unit of work successfully

`ML_STATUS_NO_OPERATION`

There was no work to be done

Get Message Count

During the processing of messages, it is sometimes necessary to inquire about the current capacity (fullness) of the message queues. The following functions provide that capability:

```
MLstatus mlGetSendMessageCount ( MLopenid openid, MLint32* messageCount);
MLstatus mlGetReceiveMessageCount ( MLopenid openid, MLint32* messageCount );
```

where:

- `openid` is a previously-opened digital media object returned by `mlOpen`.
- `messageCount` is the resulting returned count.

These functions return a count of the number of messages in the send or receive queues of a device:

- The send queue contains messages queued by the application for processing by the device. A message resides in the send queue from the moment it is enqueued by the application until the device begins processing it.
- The receive queue holds messages that have been processed and are waiting to be read by the application. A message resides in the receive queue from the moment the device enqueues it until the application dequeues the corresponding reply message. (All messages in the receive queue are counted, regardless of whether or not they were successfully processed.)

The message counts are intended to aid load-balancing in sophisticated applications. They are not a reliable method for predicting UST/MSC pairs.

Some devices can begin processing one or more following messages before the first has been completed. Thus, the sum of the send and receive queue counts may be less than the difference between the number of messages that have enqueued and dequeued by the application. The time lag between a message being removed from the send queue and the time at which it affects data passing though a physical jack is implementation-dependent. The message counts are not a reliable method for timing or synchronizing media streams.

The status return is one of the following:

`ML_STATUS_INVALID_ID`

The specified open device ID is invalid

`ML_STATUS_NO_ERROR`

The message count was obtained successfully

Receive Messsage

To obtain the results of previous digital media requests, use the `mlReceiveMessage` function:

```
MLstatus mlReceiveMessage( MLopenid openid, MLint32* messageType, MLpv *reply );
```

where:

- `openid` is a previously-opened digital media object
- `messageType` is an integer to be filled in by the device, indicating the type of message received
- `reply` is a pointer to the head of the reply message

This function reads the oldest message from the receive queue. The receive queue holds reply messages sent from a digital media device back to an application.

Messages on the receive queue may be the result of the following:

- Processing a message sent with `mlSendControls` or `mlSendBuffers`
- Generated spontaneously by the device to advise the application of some exceptional event

Each message sent with an `mlSendBuffers` or `mlSendControls` generates a single reply message with `messageType`. This indicates whether or not the message was processed successfully and a pointer to a list of parameters holding the reply.

The contents of the reply array are guaranteed to remain valid until the next call to `mlReceiveMessage`. It is acceptable for an application to modify the reply and then send it to the same or to another device by calling `mlSendControls` or `mlSendBuffers`.

On some devices, triggering of the receive wait handle does not guarantee that a message is waiting on the receive queue. Thus applications must accept a status return of `ML_STATUS_RECEIVE_QUEUE_EMPTY` from an `mlReceiveMessage` function.

End Transfer

To invoke an orderly shutdown of a digital media stream, use the `mlEndTransfer` function:

```
mlEndTransfer( openId );
```

where `openid` is a previously-opened digital media object.

This function ends a continuous transfer on the specified path or transcoder. This call advises the device to stop processing buffers and aborts any remaining messages on its input queue. This is a blocking call. It does not return until transfers have stopped and any messages remaining on the device input queue have been aborted and flushed to the device output queue. Calling `mlEndTransfer` on a device that has not begun transfers is legal (it still causes the queue to be flushed). Any messages that are flushed will be marked to indicate they were aborted. Buffer messages are marked `ML_BUFFERS_ABORTED`; control messages are marked `ML_CONTROLS_ABORTED`.

The status return is one of the following:

`ML_STATUS_INVALID_ID`

The specified open device handle is invalid

`ML_STATUS_NO_ERROR`

The device agreed to end transfer on the path

Close Processing

After an application is finished with a digital media connection, it should terminate that connection. Use the `mlClose` function to close a previously opened digital media:

```
MLstatus mlClose(MLopenid openid);
```

where `openid` is the handle of the device to be closed.

When a digital media object is closed, all messages in the message queues of the device are discarded. The device handle `openid` becomes invalid; any subsequent attempt to use it to refer to the closed object will result in an error.

Note: An `mlClose` is implied if an application terminates (for any reason) before an `mlClose` function is called.

`mlClose` returns `ML_STATUS_INVALID_ID` if `openid` is invalid. Otherwise it returns `ML_STATUS_NO_ERROR` after the device has been closed and associated resources have been freed.

The pipes opened as a side-effect of opening a transcoder are also closed as a side-effect of closing a transcoder. Pipes should not be closed explicitly.

Utility Functions

This section describes the following:

- "Get Returned Parameters" on page 117
- "Get Version" on page 118
- "Status Name" on page 118
- "Message Name" on page 119
- "`MLpv` String Conversion Routines" on page 119

Get Returned Parameters

In returned messages, the application often wants to query specific parameters. The `mlPvFind` convenience function is provided for this use:

```
MLpv* mlPvFind( MLpv* msg, MLint64 param );
```

where:

- `msg` is a message for which the parameter being searched is to be found
- `param` argument is the parameter that is being searched

Get Version

The following function obtains the version number for the ML library:

```
MLstatus mlGetVersion( MLint32* majorVersion, MLint32* minorVersion );
```

where:

- `majorVersion` is the first digit in the major version
- `minorVersion` is the first digit in the minor version

For example, the 1.0 release will have a major number of 1 and a minor number of 0. Changes in major numbers indicate a potential incompatibility, while changes in minor numbers indicate small backward-compatible enhancements. Within a particular major version, all the minor version numbers will start at 0 and increase monotonically.

Note: This is the version number of the ML core library. The version numbers for device-dependent modules are available in the capabilities list for each physical device.

The status return is one of the following:

`ML_STATUS_INVALID_ARGUMENT`

At least one of the pointers passed in is invalid

`ML_STATUS_NO_ERROR`

The version numbers were obtained successfully

Status Name

Intended mainly as an aid in debugging, the following call converts the integer ML status value into a C string. The converted string is exactly the same as the status enumerated value:

```
const char *mlStatusName( MLstatus status );
```

where `status` is the return code from an ML function.

For example, the value `ML_STATUS_NO_ERROR` is converted to the string `"ML_STATUS_NO_ERROR"`.

This function returns a valid C string or NULL character if the status value is invalid.

Message Name

Intended mainly as an aid in debugging, the following call converts the integer ML message type into a C string:

```
const char *mlMessageName( MLint32 messageType );
```

where messageType is the the message type resulting from a call to mlReceiveMessage.

The converted string is exactly the same as the message enumerated values. For example, the value ML_CONTROLS_FAILED is converted to the string "ML_CONTROLS_FAILED".

This function returns a valid C string, or NULL character if the message value is invalid.

MLpv String Conversion Routines

You can use the following routines to convert parameters and values to strings or to convert strings to parameters and values:

```
MLstatus  mlPvValueToString(MLint64 objectId, MLpv* pv, char* buffer, MLint32* bufferSize);
MLstatus  mlPvParamToString(MLint64 objectId, MLpv* pv,char* buffer, MLint32* bufferSize);
MLstatus  mlPvToString(MLint64 objectId, MLpv* pv,  char* buffer, MLint32* bufferSize);
MLstatus  mlPvValueFromString(MLint64 objectId, const char* buffer,MLint32* bufferSize,
                              MLpv* pv, MLbyte* arrayData, MLint32 arraySize);
MLstatus  mlPvParamFromString(MLint64 objectId, const char* buffer, MLint32* size, MLpv* pv);
MLstatus  mlPvFromString(MLint64 objectId, const char* buffer, MLint32* bufferSize, MLpv* pv,
                         MLbyte* arrayData, MLint32 arraySize);
```

where:

- objectID is the 64-bit ID number for the digital media library on which the parameter is interpreted.
- pv is a pointer to the MLpv for use in the conversion.
- buffer is a pointer to a buffer to hold the string.

- `bufferSize` initially contains the size of the buffer (in bytes). Upon completion, this is overwritten with the actual number of bytes processed.
- `arrayData` is a pointer to a buffer to hold any array data resulting from the conversion.
- `arraySize` initially contains the size of the array buffer (in bytes).

These routines convert between `MLpv` message `param/value` pairs and strings. They are of benefit to applications writing lists of parameters to/from files, but are most commonly used as an aid to debugging.

These routines make use of the parameter capability data to generate and interpret human-readable ASCII strings. (See the `mlPvGetCapabilities`(3dm) man page.)

`mlPvParamToString` converts `pv->param` into a string. The resulting value for `bufferSize` is the length of the string (excluding the terminating `'\0'`.

`mlPvValueToString` converts `pv->value` into a string. The resulting value for `bufferSize` is the length of the string (excluding the terminating `'\0'`).

`mlPvToString` converts the `MLpv` into a string. It writes the parameter name and value separated by `=`. The resulting value for `bufferSize` is the length of the string (excluding the terminating `'\0'`).

`mlPvParamFromString` interprets a string as a parameter name and writes the result in `pv->param`. It expects the string was created by `mlPvParamToString`.

`mlPvValueFromString` interprets a string as the value of an `MLpv` and writes the result in `pv->value`. It expects the string was created by `mlPvValueToString`. For scalar parameters, the result is returned in the value field of the `MLpv` structure and the array arguments are not used. For array parameters, additional space is required for the result. In this case, the contents of the array are returned inside the `arrayData` buffer and `arraySize` is set to indicate the number of bytes written.

`mlPvFromString` interprets a string as an `MLpv`. It expects the string was created by `mlPvToString`.

The interpretation of a `param/value` pair depends on the parameter, its value, and the device on which it will be used. Thus, all these functions require both a `param/value` pair and a 64-bit device identifier. That identifier may be one of the following:

- A static ID (obtained from a call to `mlGetCapabilities`)
- The open ID of a jack, path or transcoder (obtained from a call to `mlOpen`)
- The ID of an open pipe (obtained by calling `mlXcodeGetOpenPipe`)

The status returns are:

`ML_STATUS_INVALID_ARGUMENT`

The arguments could not be interpreted correctly. Perhaps the `bufferSize` or `arraySize` is too small to hold the result of the operation.

`ML_STATUS_INVALID_ID`

The specified id is invalid.

`ML_STATUS_INVALID_PARAMETER`

The parameter name is invalid. When converting to a string, the parameter name was not recognized on this device. When converting from a string, the string could not be interpreted as a valid parameter for this device.

`ML_STATUS_INVALID_VALUE`

The parameter value is invalid. When converting to a string, the parameter value was not recognized on this device. When converting from a string, the string could not be interpreted as a valid parameter value for this device.

`ML_STATUS_NO_ERROR`

The conversion was performed successfully.

Example: Printing the Interpretation of a Video Timing Parameter

The following example prints the interpretation of a video timing parameter by a previously-opened video path. The calls could fail if that path did not accept the particular timing value we have chosen here. Because the interpretation is coming from the device, this example will work for device-specific parameters.

```
char buffer[200];
MLpv control;

  control.param = ML_VIDEO_TIMING_INT32;
  control.value = ML_TIMING_1125_1920x1080_5994i;

  mlPvParamToString(someOpenPath, &control, buffer, sizeof(buffer));
  printf("control.param is %s\n", buffer);
  mlPvValueToString(someOpenPath, &control, buffer, sizeof(buffer));
  printf("control.value is %s\n", buffer);
  mlPvToString(someOpenPath, &control, buffer, sizeof(buffer));
```

Synchronization

This chapter describes ML support for synchronizing digital media streams. The described techniques are designed to enable accurate synchronization even when there are large (and possibly unpredictable) processing delays.

This chapter contains the following sections:

- "Time Representation" on page 123
- "Get Unadjusted System Time (UST)" on page 124
- "UST/MSC/ASC Parameters" on page 124

Time Representation

To time-stamp each media stream, some convenient representation for time is needed. In ML, time is represented by the value of the unadjusted system time (UST) counter. That counter starts at 0 when the system is reset and increases continuously (without any adjustment) while the system is running.

Each process and/or piece of hardware may have its own view of the UST counter. That view is an approximation to the real UST counter. The difference between any two views is bounded for any implementation.

Each UST time stamp is a signed 64-bit integer value with units of nanoseconds representing a recent view of the UST counter. To obtain a current view of the UST, use the `mlGetSystemUST` function call:

```
MLstatus mlGetSystemUST(MLint64 systemId, MLint64* ust);
```

where:

- `systemId` is `ML_SYSTEM_LOCALHOST` (any other value results in the return of `ML_STATUS_INVALID_ID`)
- `ust` is a pointer to an `int64` that will hold the resulting UST value

The status return is one of the following:

`ML_STATUS_INVALID_ARGUMENT`

UST is invalid

`ML_STATUS_NO_ERROR`

Successful execution

Get Unadjusted System Time (UST)

To obtain the current UST on a particular system, use the following:

```
MLstatus mlGetSystemUST( systemId,  );
```

where `systemID` is `ML_SYSTEM_LOCALHOST`.

This function returns one of the following:

`ML_STATUS_INVALID_ARGUMENT`

The UST was not returned successfully (there may be an invalid pointer)

`ML_STATUS_INVALID_ID`

The specified system ID is invalid

`ML_STATUS_NO_ERROR`

The system UST was obtained successfully

UST/MSC/ASC Parameters

This section discusses the following:

- "Unadjusted System Time (UST) Parameters" on page 125
- "Media Stream Count (MSC) Parameters" on page 125
- "Application Stream Count (ASC) Parameters" on page 126
- "UST/MSC and Corresponding Messages" on page 126

- "UST/MSC Example" on page 127
- "Predicate Controls" on page 130

Basic support for synchronization requires that the application know exactly when video or audio buffers are passed through a jack. In ML, this is achieved with the UST and MSC buffer parameters.

Unadjusted System Time (UST) Parameters

The UST parameters are as follows:

```
ML_AUDIO_UST_INT64, ML_VIDEO_UST_INT64
```

The UST is the time stamp for the most recently processed slot in the audio/video stream:

- For video devices, the UST corresponds to the time at which the field/frame starts to pass through the jack
- For audio devices, the UST corresponds to the time at which the first sample in the buffer passed through the jack

Media Stream Count (MSC) Parameters

The MSC parameters are as follows:

```
ML_AUDIO_MSC_INT64, ML_VIDEO_MSC_INT64
```

The MSC is the most recently processed slot in the audio/video stream. This is snapped at the same instant as the UST described in "Unadjusted System Time (UST) Parameters" on page 125.

MSC increases by 1 for each potential slot in the media stream through the jack. For interlaced video timings, each slot contains one video field; for progressive timings, each slot contains one video frame. This means that when two fields are interlaced into one frame and sent as one buffer, then the MSC will increment by 2 (one for each field). Furthermore, the system guarantees that the least significant bit of the MSC will reflect the state of the field bit: `0` for Field 1 and `1` for Field 2. For audio, each slot contains one audio frame.

Application Stream Count (ASC) Parameters

The ASC parameters are as follows:

```
ML_AUDIO_ASC_INT64, ML_VIDEO_ASC_INT64
```

The ASC is provided to aid the developer in predicting when the audio or video data will pass through an output jack. See "UST/MSC Example" on page 127 for further information on the use of the ASC parameter.

UST/MSC and Corresponding Messages

Typically, an application will pass `mlSendBuffers` one of the following:

- A video message containing values for the following:
  - `ML_IMAGE_BUFFER_POINTER`
  - `ML_VIDEO_MSC_INT64`
  - `ML_VIDEO_UST_INT64` (and possibly for `ML_VIDEO_ASC_INT64`)
- An audio message containing values for the following:
  - `ML_AUDIO_BUFFER_POINTER`
  - `ML_AUDIO_UST_INT64`
  - `ML_AUDIO_MSC_INT64`

In some cases, a message can contain both audio and video parameters.

Each message is processed as a single unit and a reply is returned to the application via `mlReceiveMessage`. The reply will contain the completed buffer and the UST/MSC or UST/ASC corresponding to the time at which the data in the buffers passed in or out of the jack.

Note: Due to hardware buffering on some cards, it is possible to receive a reply message before the data has finished flowing through an output jack.

UST/MSC Example

The following example sends an audio buffer and video buffer to an I/O path and requests both UST and MSC stamps:

```
MLpv message[7];
message[0].param = ML_IMAGE_BUFFER_POINTER;
message[0].value.pByte = someImageBuffer;
message[0].length = sizeof(someImageBuffer);
message[0].maxLength = sizeof(someImageBuffer);
message[1].param = ML_VIDEO_UST_INT64;
message[2].param = ML_VIDEO_MSC_INT64;
message[3].param = ML_AUDIO_BUFFER_POINTER;
message[3].value.pByte = someAudioBuffer;
message[3].length = sizeof(someAudioBuffer);
message[3].maxLength = sizeof(someAudioBuffer);
message[4].param = ML_AUDIO_UST_INT64;
message[5].param = ML_AUDIO_MSC_INT64;
message[6].param = ML_END;
mlSendBuffers( device, message);
```

After the device has processed the buffers, it will enqueue a reply message back to the application. That reply will be an exact copy of the message sent in, with the exception that the MSC and UST values will be filled in. (For input, the buffer parameter length will also be set to the number of bytes written into it).

Note: An `mlSendBuffers` call can only have one `ML_IMAGE_BUFFER_POINTER.`

This section discusses the following:

- "UST/MSC for Input" on page 127
- "UST/MSC for Output" on page 128

UST/MSC for Input

On input, the application can detect if any data is missing by looking for breaks in the MSC sequence. This could happen if an application did not provide buffers fast enough to capture all of the signal that arrived at the jack. (An alternative to looking at the MSC numbers is to turn on the events `ML_AUDIO_SEQUENCE_LOST` or `ML_VIDEO_SEQUENCE_LOST`. Those will fire whenever the queue from application to device overflows.)

Given the UST/MSC stamps for two different buffers (UST1,MSC1) and (UST2,MSC2), the input sample rate in samples per nanosecond can be computed as follows:

$$sampleRate = \frac{(MSC2 - MSC1)}{UST2 - UST1}$$

Figure 10-1 Sample Input Rate

One common technique for synchronizing different input streams is to start recording early, stop recording late, and then use the UST/MSC stamps in the recorded data to find exact points for trimming the input data.

An alternative way to start recording several streams simultaneously is to use predicate controls (see "Predicate Controls" on page 130).

UST/MSC for Output

On output, the actual output sample rate can be computed in exactly the same way as the input sample rate:

$$sampleRate = \frac{(MSC2 - MSC1)}{(UST2 - UST1)}$$

Figure 10-2 Actual Sample Rate

Some applications must determine exactly when the next buffer sent to the device will actually go out the jack. Doing this requires the following steps:

1. Maintain the field/frame count for the application. This parameter is called the ASC. The ASC may start at any desired value and should increase by one for every audio frame or video field enqueued. (For convenience, the application may wish to associate the ASC with the buffer by embedding it in the same message. The parameters `ML_AUDIO_ASC_INT32` and `ML_VIDEO_ASC_INT32` are provided for this use.)
2. Detect if there was any underflow by comparing the number of slots the application thought it had output with the number of slots that the system

actually output. (This assumes that the application knows the UST/MSC/ASC for two previously-output buffers.) If the following equation is true, then all is well:

$$(ASC2 - ASC1) == (MSC2 - MSC1)$$

Figure 10-3 Determining Underflow

3. Predict that the next data the application enqueues has the following system sequence count:

$$currentMSC = currentASC + (MSC2 - ASC2)$$

Figure 10-4 System Sequence Count

(This assumes that the application knows the current ASC.)

Predict when the data will hit the output jack:

$$currentUST = UST2 + \frac{(currentASC - ASC2)}{sampleRate}$$

Figure 10-5 Predict when the Data will Hit the Output Jack

The application should periodically recompute the actual sample rate based on measured MSC/UST values. It is not sufficient to rely on a nominal sample rate because the actual rate may drift over time.

In summary: given the above mechanism, the application knows the UST/MSC pair for every processed buffer. Using the UST/MSC's for several processed buffers, you can compute the frame rate. Given a UST/MSC pair in the past, a prediction of the current MSC, and the frame rate, the application can predict the UST at which the next buffer to be enqueued will hit the jack.

Predicate Controls

Predicate controls allow an application to insert conditional commands into the queue to the device. Using these, you can preprogram actions, allowing the device to respond immediately, without needing to wait for a round-trip through the application.

Unlike the UST/MSC time stamps, predicate controls are not required to be supported on all audio/video devices. To see if they are supported on any particular device, look for the desired parameter in the list of supported parameters on each path; see the `mlGetCapabilities`(3dm) man page. The simplest predicate controls are as follows:

```
ML_WAIT_FOR_AUDIO_MSC_INT64
ML_WAIT_FOR_VIDEO_MSC_INT64
```

When the message containing these controls reaches the head of the queue, it causes the queue to stall until the specified MSC value has passed. Then that message, and subsequent messages, are processed as normal.

For example, following is code that uses `WAIT_FOR_AUDIO_MSC` to send a particular buffer out after a specified stream count:

```
MLpv message[3];
message[0].param = ML_WAIT_FOR_AUDIO_MSC_INT64;
message[0].value.int64 = someMSCInTheFuture;
message[1].param = ML_AUDIO_BUFFER_POINTER;
message[1].value.pByte = someBuffer;
message[1].value.length = sizeof(someBuffer);
message[2].param = ML_END;
mlSendBuffers( someOpenPath, message);
```

This places a message on the queue to the path and then immediately returns control to the application. As the device processes that message, it will pause until the specified media MSC value has passed before allowing the buffer to flow through the jack.

Using this technique an application can program several media streams to start in synchronization by simply choosing some MSC count to start in the future.

Note: If both `ML_IMAGE_DOMINANCE` and `ML_WAIT_FOR_VIDEO_MSC` controls are set and do not correspond to the same starting output field order, the `ML_WAIT_FOR_VIDEO_MSC_INT64` control will override `ML_IMAGE_DOMINANCE_INT32` control settings.

Another set of synchronization predicate controls are:

```
ML_WAIT_FOR_AUDIO_UST_INT64
ML_WAIT_FOR_VIDEO_UST_INT64
```

When the message containing these controls reaches the head of the queue it causes the queue to stall until the specified UST value has passed. Then that message, and subsequent messages, are processed as normal.

Note: The accuracy with which the system is able to implement the `WAIT_FOR_UST` command is device-dependent. For more information, see the device-specific documentation for limitations.

For example, the following code uses `WAIT_FOR_AUDIO_UST` to send a particular buffer out after a specified time:

```
MLpv message[3];
message[0].param = ML_WAIT_FOR_AUDIO_UST_INT64;
message[0].value.int64 = someUSTtimeInTheFuture;
message[1].param = ML_AUDIO_BUFFER_POINTER;
message[1].value.pByte = someBuffer;
message[1].value.length = sizeof(someBuffer);
message[2].param = ML_END;
mlSendBuffers( someOpenPath, message);
```

This places a message on the queue to the path and then immediately returns control to the application. As the device processes that message, it will pause until the specified video UST time has passed before allowing the buffer to flow through the jack.

Using this technique, an application can program several media streams to start in synchronization by simply choosing some UST time in the future and programming each to start at that time. The following predicates control processing up to a specified time:

```
ML_IF_VIDEO_UST_LT_INT64
ML_IF_AUDIO_UST_LT_INT64
```

When included in a message, these controls will cause the following logical test: if the UST is less than the specified time, then the entire message is processed as normal; otherwise, the entire message is simply skipped.

Regardless of the outcome, any following messages are processed as normal. Skipping over a message takes time, so there is a limit to how many messages a device can skip before the delay starts to become noticeable. All media devices will support skipping at least one message without noticeable delay.

Pixels in Memory

This appendix provides examples of the more common in-memory pixel formats and their corresponding ML parameters:

- "Greyscale Examples" on page 133
- "RGB Examples" on page 134
- "CbYCr Examples" on page 138
- "422*x* CbYCr Examples" on page 140

Greyscale Examples

This section discusses the following:

- "8-bit Greyscale (1 Byte Per Pixel)" on page 133
- "Padded 12-bit Greyscale (1 Short Per Pixel)" on page 134

8-bit Greyscale (1 Byte Per Pixel)

```
byte 0
7      0
+------+
YYYYYYYY
```

Parameters:

```
ML_PACKING_8
ML_COLORSPACE_CbYCr_*
ML_SAMPLING_400
```

Padded 12-bit Greyscale (1 Short Per Pixel)

```
short 0
15             0
+--------------+
ssssYYYYYYYYYYY
```

Parameters:

```
ML_PACKING_S12in16R
ML_COLORSPACE_CbYCr_*
ML_SAMPLING_400
```

RGB Examples

This section discusses the following:

- "8-bit RGB (3 Bytes Per Pixel)" on page 135
- "8-bit BGR (3 Bytes Per Pixel)" on page 135
- "8-bit RGBA (4 Bytes Per Pixel)" on page 135
- "8-bit ABGR (4 Bytes Per Pixel)" on page 136
- "10-bit RGB (One 32-bit Integer Per Pixel)" on page 136
- "10-bit RGBA (One 32-bit Integer Per Pixel)" on page 136
- "12-bit RGBA (6 Bytes Per Pixel)" on page 137
- "Padded 12-bit RGB (Three 16-bit shorts per pixel)" on page 137
- "Padded 12-bit RGBA (Four 16-bit Shorts Per Pixel)" on page 137

8-bit RGB (3 Bytes Per Pixel)

```
byte 0    byte 1    byte 2
7      0  7      0  7      0
+------+  +------+  +------+
RRRRRRRR  GGGGGGGG  BBBBBBBB
```

Parameters:

```
ML_PACKING_8
ML_COLORSPACE_RGB_*
ML_SAMPLING_444
```

8-bit BGR (3 Bytes Per Pixel)

```
byte 0    byte 1    byte 2
7      0  7      0  7      0
+------+  +------+  +------+
BBBBBBBB  GGGGGGGG  RRRRRRRR
```

Parameters:

```
ML_PACKING_8_R
ML_COLORSPACE_RGB_*
ML_SAMPLING_444
```

8-bit RGBA (4 Bytes Per Pixel)

```
byte 0    byte 1    byte 2    byte 3
7      0  7      0  7      0  7      0
+------+  +------+  +------+  +------+
RRRRRRRR  GGGGGGGG  BBBBBBBB  AAAAAAAA
```

Parameters:

```
ML_PACKING_8
ML_COLORSPACE_RGB_*
ML_SAMPLING_4444
```

8-bit ABGR (4 Bytes Per Pixel)

```
byte 0    byte 1    byte 2    byte3
7      0  7      0  7      0  7      0
+------+  +------+  +------+  +------+
AAAAAAAA  BBBBBBBB  GGGGGGGG  RRRRRRRR
```

Parameters:

```
ML_PACKING_8_R
ML_COLORSPACE_RGB_*
ML_SAMPLING_444
```

10-bit RGB (One 32-bit Integer Per Pixel)

```
31            int              0
+-----------------------------+
RRRRRRRRRRGGGGGGGGGGBBBBBBBBBB00
```

Parameters:

```
ML_PACKING_10_10_10_2
ML_COLORSPACE_RGB_*
ML_SAMPLING_444
```

10-bit RGBA (One 32-bit Integer Per Pixel)

```
31            int              0
+-----------------------------+
RRRRRRRRRRGGGGGGGGGGBBBBBBBBBBAA
```

Parameters:

```
ML_PACKING_10_10_10_2
ML_COLORSPACE_RGB_*
ML_SAMPLING_4444
```

12-bit RGBA (6 Bytes Per Pixel)

```
byte 0    byte 1    byte 2    byte 3    byte 4    byte 5
7      0  7      0  7      0  7      0  7      0  7      0
+------+  +------+  +------+  +------+  +------+  +------+
RRRRRRRR  RRRRGGGG  GGGGGGGG  BBBBBBBB  BBBBAAAA  AAAAAAAA
```

Parameters:

```
ML_PACKING_S12
ML_COLORSPACE_RGB_*
ML_SAMPLING_4444
```

Padded 12-bit RGB (Three 16-bit shorts per pixel)

```
short 0           short 1           short 2
15             0  15             0  15             0
+--------------+  +--------------+  +--------------+
ssssRRRRRRRRRRRR  ssssGGGGGGGGGGGG  ssssBBBBBBBBBBBB
```

Parameters:

```
ML_PACKING_S12in16R
ML_COLORSPACE_RGB_*
ML_SAMPLING_444
```

Padded 12-bit RGBA (Four 16-bit Shorts Per Pixel)

```
short 0           short 1           short 2           short 3
15             0  15             0  15             0  15             0
+--------------+  +--------------+  +--------------+  +--------------+
ssssRRRRRRRRRRRR  ssssGGGGGGGGGGGG  ssssBBBBBBBBBBBB  ssssAAAAAAAAAAAA
```

Parameters:

```
ML_PACKING_S12in16R
ML_COLORSPACE_RGB_*
ML_SAMPLING_4444
```

CbYCr Examples

This section discusses the following:

- "8-bit CbYCr (3 Bytes Per Pixel)" on page 138
- "8-bit CbYCrA (4 Bytes Per Pixel)" on page 138
- "10-bit CbYCr (One 32-bit Integer Per Pixel)" on page 139
- "10-bit CbYCrA (One 32-bit Integer Per Pixel)" on page 139
- "Padded 12-bit CbYCrA (Four 16-bit Shorts Per Pixel)" on page 139

8-bit CbYCr (3 Bytes Per Pixel)

```
byte 0    byte 1    byte 2
7      0  7      0  7      0
+------+  +------+  +------+
bbbbbbbb  YYYYYYYY  rrrrrrrr
```

Parameters:

```
ML_PACKING_8
ML_COLORSPACE_CbYCr_*
ML_SAMPLING_444
```

8-bit CbYCrA (4 Bytes Per Pixel)

```
byte 0    byte 1    byte 2    byte 3
7      0  7      0  7      0  7      0
+------+  +------+  +------+  +------+
bbbbbbbb  YYYYYYYY  rrrrrrrr  AAAAAAAA
```

Parameters:

```
ML_PACKING_8
ML_COLORSPACE_CbYCr_*
ML_SAMPLING_4444
```

10-bit CbYCr (One 32-bit Integer Per Pixel)

```
31            int              0
+------------------------------+
bbbbbbbbbbYYYYYYYYYYrrrrrrrrrr00
```

Parameters:

```
ML_PACKING_10_10_10_2
ML_COLORSPACE_CbYCr_*
ML_SAMPLING_444
```

10-bit CbYCrA (One 32-bit Integer Per Pixel)

```
31             int             0
+------------------------------+
bbbbbbbbbbYYYYYYYYYYrrrrrrrrrrAA
```

Parameters:

```
ML_PACKING_10_10_10_2
ML_COLORSPACE_CbYCr_*
ML_SAMPLING_4444
```

Padded 12-bit CbYCrA (Four 16-bit Shorts Per Pixel)

```
short 0           short 1           short 2           short 3
15             0  15             0  15             0  15             0
+--------------+  +--------------+  +--------------+  +--------------+
ssssbbbbbbbbbbbb  ssssYYYYYYYYYYYY  ssssrrrrrrrrrrrr  ssssAAAAAAAAAAAA
```

Parameters:

```
ML_PACKING_S12in16R
ML_COLORSPACE_CbYCr_*
ML_SAMPLING_4444
```

422*x* CbYCr Examples

This section discusses the following:

- "10-bit 422 CbYCr (5 Bytes Per 2 Pixels)" on page 140
- "10-bit 422 CbYCr (5 Bytes Per 2 Pixels)" on page 141
- "Padded 12-bit 422 CbYCr (Four 16-bit Shorts Per 2 Pixels)" on page 141
- "10-bit 4224 CbYCrA (Two 32-bit Integers Per 2 Pixels)" on page 142

10-bit 422 CbYCr (5 Bytes Per 2 Pixels)

```
      byte 0    byte 1    byte 2    byte 3    byte 4
      7      0  7      0  7      0  7      0  7      0
      +------+  +------+  +------+  +------+  +------+
      bbbbbbbb  bbYYYYYY  YYYYrrrr  rrrrrrYY  YYYYYYYY
```

pixel 1

```
      ++++++++  ++++++++  ++++++++  ++++++
```

pixel 2

```
      ++++++++  ++        ++++      ++++++++  ++++++++
```

Parameters:

```
ML_PACKING_10
ML_COLORSPACE_CbYCr_*
ML_SAMPLING_422
```

10-bit 422 CbYCr (5 Bytes Per 2 Pixels)

```
      byte 0    byte 1    byte 2    byte 3    byte 4
      7      0  7      0  7      0  7      0  7      0
      +------+  +------+  +------+  +------+  +------+
      bbbbbbbb  bbYYYYYY  YYYYrrrr  rrrrrrYY  YYYYYYYY
```

pixel 1

```
     ++++++++  ++++++++  ++++++++  ++++++
```

pixel 2

```
     ++++++++  ++        ++++      ++++++++  ++++++++
```

Parameters:

```
ML_PACKING_10_R
ML_COLORSPACE_CbYCr_*
ML_SAMPLING_422
```

Padded 12-bit 422 CbYCr (Four 16-bit Shorts Per 2 Pixels)

```
   short 0           short 1           short 2           short 3
   15             0  15             0  15             0  15             0
   +--------------+  +--------------+  +--------------+  +--------------+
   ssssbbbbbbbbbbbb  ssssYYYYYYYYYYYY  ssssrrrrrrrrrrrr  ssssYYYYYYYYYYYY
```

pixel 1

```
   ++++++++++++++++  ++++++++++++++++  ++++++++++++++++
```

pixel 2

```
   ++++++++++++++++                    ++++++++++++++++  ++++++++++++++++
```

Parameters:

```
ML_PACKING_S12in16R
ML_COLORSPACE_CbYCr_*
ML_SAMPLING_422
```

10-bit 4224 CbYCrA (Two 32-bit Integers Per 2 Pixels)

```
   31          int 0              0  31          int 1              0
   +------------------------------+  +------------------------------+
   bbbbbbbbbbYYYYYYYYYYAAAAAAAAAA00  rrrrrrrrrrYYYYYYYYYYAAAAAAAAAA00
```

pixel 1

```
   ++++++++++++++++++++++++++++++    ++++++++++
```

pixel 2

```
   ++++++++++                        ++++++++++++++++++++++++++++++
```

Parameters:

```
ML_PACKING_10_10_10_2
ML_COLORSPACE_CbYCr_*
ML_SAMPLING_4224
```

Common Video Standards

This section diagrams common video standards:

- 525/60 timing (NTSC): see Figure B-1 on page 144
- 625/50 timing (PAL): see Figure B-2 on page 145
- 080i timing (high definition): see Figure B-3 on page 146
- 720p timing (high definition): see Figure B-4 on page 147

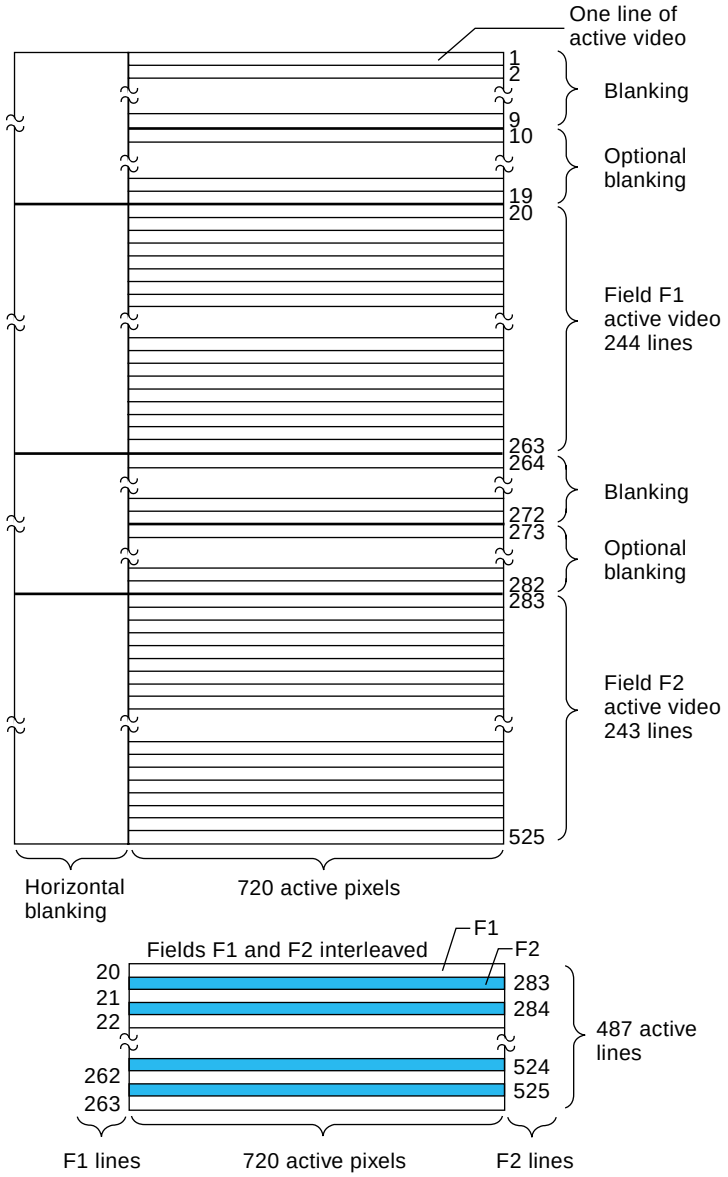

Figure B-1 525/60 Timing (NTSC)

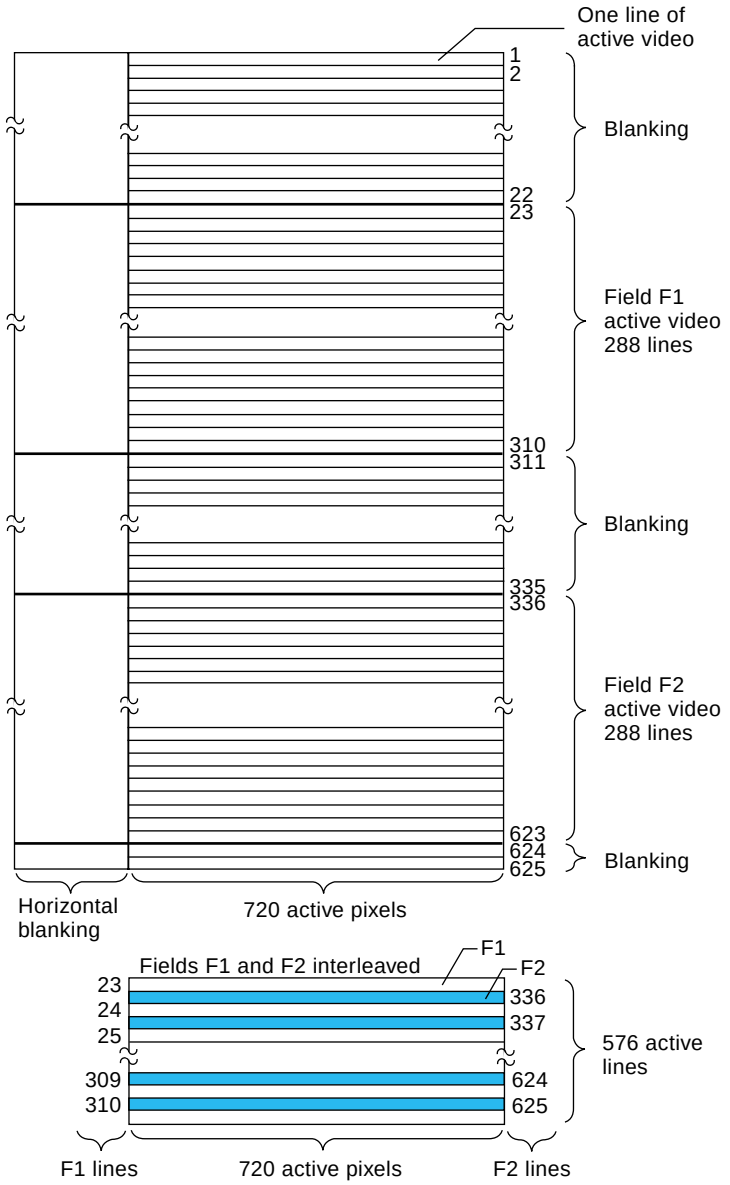

Figure B-2 625/50 Timing (PAL)

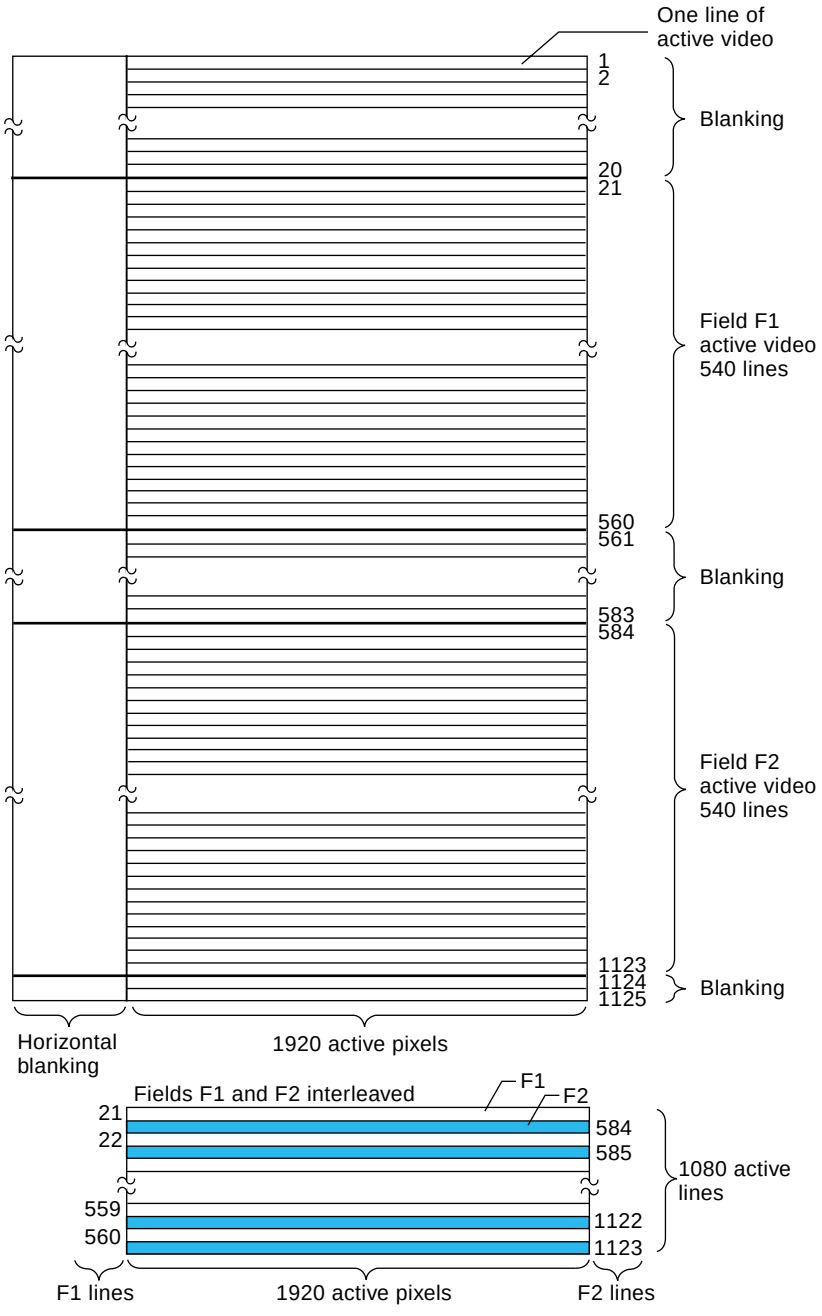

Figure B-3 1080i Timing (High Definition)

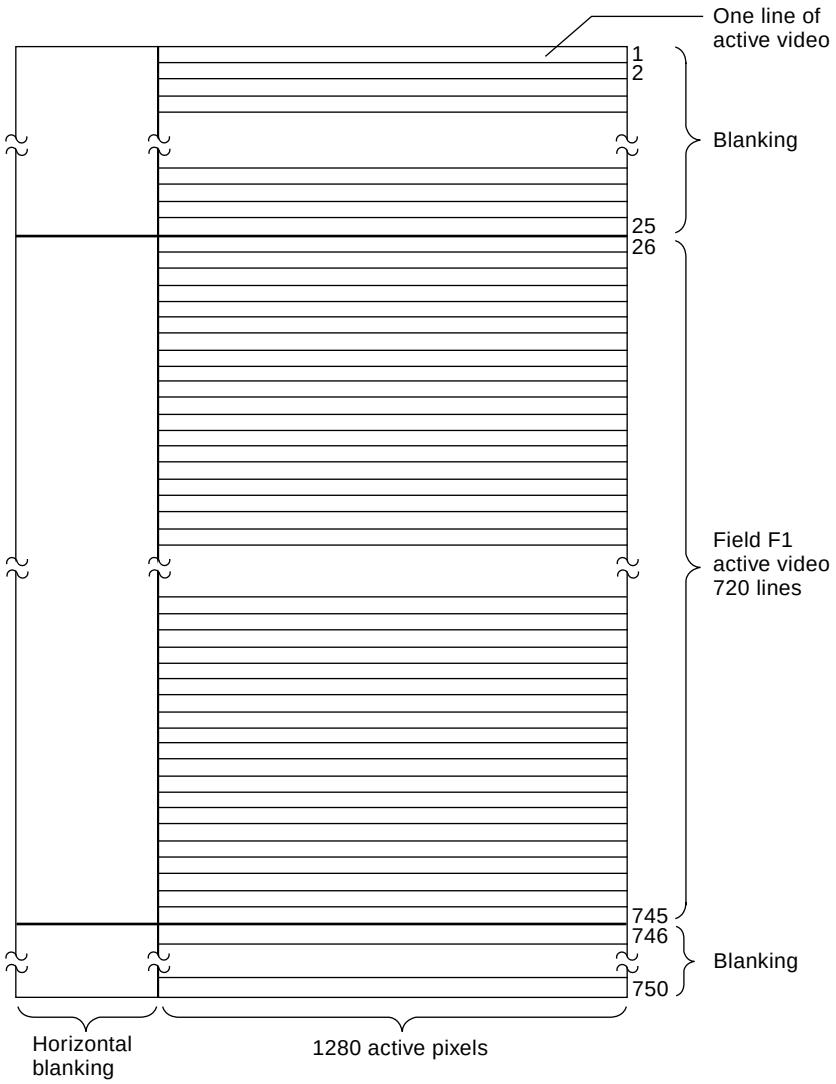

Figure B-4 720p Timing (High Definition)

Glossary

ASC (application stream control)

A measure used to predict when the audio or video will pass through an output jack.

capabilities tree

The hierarchy of all ML devices in the system. The capabilities tree contains information about each ML device. An application may search a capabilities tree to find suitable media devices for operations you wish to perform.

codec

An audio, image or video compressor/decompressor.

graphics

The graphical display used for the user interface on a computer; compare to *video*.

in-band messages

When an application waits for all previously enqueued messages to be processed before influencing a device. See also *out-of-band message*.

interlaced video

Video that is divided into two fields (upper and lower). Video standards that use interlacing are NTSC, PAL, and SECAM. In interlaced video, the first field is transmitted first (odd lines) then the second field of the frame is transmitted (even lines).

interleave

The method of laying out the lines of interlaced video data in memory. In interleaved mode, the lines of the frame are stored as follows in one buffer:

`f1-line1,f2-line1,f1-line2,f2-line2,f1-line3,f2-line3,...f1-lineN,f2-line`*N*

In noninterleave mode, the lines of the fields as stored as follows:

- In one buffer:

  `f1-line1,f2-line2,f1-line3 ... f1-line`*N*

- In another buffer:

  `f2-line1,f2-line2,f2-line3 ... f2-line`*N*

jack
A logical device that is an interface in/out of the system. Examples of jacks are composite video connectors and microphones. Jacks often, but not necessarily, correspond to a physical connector — it is possible for a single ML jack to refer to several such connectors. It is also possible for a single physical connector to appear as several logical jacks.

MSC (media stream count)
A measure of the number of media samples that have passed though a jack. This is useful to synchronize media streams.

logical device
A jack, path, or transcoder. See also *physical device*.

out-of-band message
When an application wishes to influence a device without first waiting for all previously enqueued messages to be processed. See also *in-band message*.

path
A logical device that provides logical connections between memory and jacks. For example, a video output path transports data from buffers to a video output jack. Paths are logical entities. Depending on the device, it is possible for more than one instance of a path to be open and in use at the same time.

pipe
The connections from memory to the transcoder, and from the transcoder to memory.

physical device

A device that corresponds to device-dependent modules in ML. Typically, each device-dependent module supports a set of software transcoders or a single piece of hardware. Examples of devices are audio cards on a PCI bus, DV camcorders on the 1394 bus, or software DV modules. Each device-dependent module may expose a number of logical devices.

system

The highest level in the capability tree hierarchy. The system is the machine on which your application is running. This machine is given the name `ML_SYSTEM_LOCALHOST`. Each system contains one or more devices.

transcoder

A logical device that takes data from buffers via one or more input pipes, performs an operation on the data, and returns the data to another buffer via an output pipe. Example transcoders are DV compression and JPEG decompression.

UST (unadjusted system time)

A special system clock that runs continuously without adjustment. This clock is used to synchronize media streams.

video

The type of signal sent to a video cassette recorder or received from a camcorder; compare to *graphics*.

Index

422x CbYCr Examples, 140
720p HD timing chart, 143
1080i HD timing chart, 143

A

array values, 20
ASC parameters, 126
audio buffer layout, 87
audio buffer size computation, 93
audio parameters, 89
  ML_AUDIO_BUFFER_POINTER, 89
  ML_AUDIO_CHANNELS_INT32, 89
  ML_AUDIO_COMPRESSION_INT32, 90
  ML_AUDIO_FORMAT_INT32, 90
  ML_AUDIO_FRAMESIZE_INT32, 92
  ML_AUDIO_GAINS_REAL64_ARRAY, 92
  ML_AUDIO_PRECISION_INT32, 92
  ML_AUDIO_SAMPLE_RATE_REAL64, 92
audio/video jacks, 13
audio/visual paths, 45

B

bit reordering, 85
buffer
  allocation, 9
  how to send to device for processing, 7
  refill, 12
  sending, 106

C

capabilities access, 41
capabilities access via function calls, 26
capabilities list, 41
capabilities tree, 1, 25
capabilities utility functions, 26
CbYCr examples, 138
clock, 3
colorspace parameter format, 74
colorspace values, 61
common video standards, 143
constant identification numbers, 30
control query, 103
controls query, 109
controls sending, 104
controls setting, 102

D

destination pipes, 53
device location, 4
device open, 9
device open options, 38
device output path, 5
device path controls, 6
device path set up, 5
device states, 99

E

example programs online, 1
exception events, 49

F

FI-dominant vs F2–dominant, 77

field dominance, 77
freeing capabilities, 44

G

get version, 118
graphics/video distinction between, 1
greyscale examples, 133

H

high definition (HD) timings, 69

I

identification numbers, 30
image buffer, 71
image buffer parameters
  ML_IMAGE_BUFFER_POINTER, 73
  ML_IMAGE_BUFFER_SIZE_INT32, 74
  ML_IMAGE_COLORSPACE_INT32, 74
  ML_IMAGE_COMPRESSION_FACTOR_REAL32, 76
  ML_IMAGE_COMPRESSION_INT32, 76
  ML_IMAGE_DOMINANCE_INT32, 77
  ML_IMAGE_HEIGHT_1_INT32, 78
  ML_IMAGE_HEIGHT_2_INT32, 78
  ML_IMAGE_INTERLEAVE_MODE_INT32, 78
  ML_IMAGE_ORIENTATION_INT32, 79
  ML_IMAGE_PACKING_INT32, 79
  ML_IMAGE_ROW_BYTES_INT32, 82
  ML_IMAGE_SAMPLING_INT32, 82
  ML_IMAGE_SKIP_PIXELS_INT32, 84
  ML_IMAGE_SKIP_ROWS_INT32, 85
  ML_IMAGE_TEMPORAL_SAMPLING_INT32, 85
  ML_IMAGE_WIDTH_INT32, 85
  ML_SWAP_BYTES_INT32, 85
image parameters, 59, 71
in-band messages, 47, 48
in-band reply messages, 50
individual parameters of logical devices, 29
interlaced, 78
interlaced sampling, 60
interleaved, 78
interpretation of a video timing parameter, 122
introduction to ML, 1

J

jack, 2
jack closing, 15
jack controls, 14
jack direction, 34
jack logical device capabilities, 33
jack open parameters, 38
jack opening, 13

L

local system capabilities, 27
logical device, 2
logical device capabilities, 33
logical device individual parameters, 29
logical devices, 28
logical path closing, 51
logical path opening, 45

M

media stream count, 3
message construction, 14, 46
message name, 119
message processing, 7
message reception, 8
messages, 18
messages, description, 18
ML processing, 95
ML program structure, 96

ML.h, 4
ML_BUFFERS_xxx, 108
ML_CHANNELS_xxx, 90
ML_COMPRESSION_xxx, 90
ML_DOMINANCE_xxx, 78
ML_EVENT_xxx, 49
ML_FORMAT_xxx, 90
ML_IMAGE_BUFFER_POINTER, 71
ML_INTERLEAVE_MODE_XXX, 79
ML_JACK_TYPE_xxxx, 33
ML_MODE_xxx, 38
ML_PATH_BUFFER_ALIGNMENT_INT32, 71
ML_PATH_COMPONENT_ALIGNMENT_INT32, 71
ML_SAMPLING_xxx, 67
ML_SIGNAL_xxx, 65
ML_SYSTEM_LOCALHOST, 2
ML_TIMING_xxx, 64, 67
ML_VIDEO, 14
ML_VIDEO_xxx, 61
mlBeginTransfer, 10, 51, 56, 112
mlBeginTransfer call, 7
mlClose, 15, 51
mlclose, 12
mlClose call, 8
mlEndTransfer, 12, 51, 57, 58
mlFreeCapabilities, 27
mlGetCapabilites, 54
mlGetCapabilities, 26, 28
mlGetControls, 14, 19, 21, 103
mlGetControls call, 21
mlGetSystemUST, 123
mlOpen, 9, 13, 100
MLpv
  and scalar parameters, 18
MLpv string conversion routines, 119
mlPvGetCapabilities, 27
mlquery
  system inventory tool, 3
mlReceiveMessage, 47, 50
mlSendBuffers, 13, 23, 47, 55
mlSendBuffers call, 7
mlSendControls, 13, 20, 104
mlSendControls call, 47
mlSetControls, 13, 14, 102
mlSetControls call, 19
MLstatus return Value, 97
mlu.h, 4
mlXcodeWork call, 58
MSC, 3
MSC parameters, 125
MSC/UST and corresponding messages, 126

N

NTSC timing chart, 143

O

online ML example programs, 1
open identification numbers, 30
open options, 38
open path identifier, 5
out-of-band messages, 46

P

PAL timing chart, 143
param/value pairs, 17
parameter syntax and semantics, 17
parameters that describe parameters, 29
path, 2
path closing, 8
path logical device capabilities, 34
path open parameters, 39
physical device, 2
physical device capabilities, 32
physical devices, 28
pipe, 2
pipe logical device capabilities, 37
pixels in memory, 133

422x examples, 140
CbYCr examples, 138
greyscale examples, 133
RGB examples, 135
pointer values, 22
predicate controls, 130
program examples
realistic audio output program, 8
simple audio output program, 3
progressive sampling, 60

R

realistic audio output program, 8
Rec 709, 75
RED 601, 75
refill the buffer, 12
reply message, 57
RGB examples, 134

S

sample frame, 87
sampling of video, 59
sampling parameter format, 82
sampling values, 67
scalar values, 18
select, 11
simple audio output program, 3
SMPTE 240, 75
source pipes, 53
spatial sampling, 59
standard definition (SD) timings, 68
standards, 143
static identification numbers, 30
status name, 118
string conversion routines, 119
supported timings, 68
synchronization, 123
synchronize media streams, 3
system capabilities, 31
system clock, 3
system level, 2

T

temporal sampling, 59
terminology, 1
time representation, 123
time stamp, 11
timing charts
1080i, 143
525/60 (NTSC), 143
625/50 (PAL), 143
720p, 143
timings, 68, 69
tools
mlquery system inventory, 3
transcoder, 2
transcoder logical device capabilities, 36
transcoder open parameters, 40
transcoders, 53
transfer, 10, 12
transfers, 51, 112

U

unadjusted system time, 3
UST, 3, 10, 124
UST parameters, 125
UST synchronization, 123
UST/MSC and corresponding messages, 126
UST/MSC example, 127

V

versions, 118
video example, 70

video field dominance, 78
video frame, 60
video parameters, 61
ML_VIDEO_ALPHA_SETUP_INT32, 61
ML_VIDEO_BLUE_SETUP_INT32, 61
ML_VIDEO_BRIGHTNESS_INT32, 61
ML_VIDEO_COLORSPACE_INT32, 61
ML_VIDEO_CONTRAST_INT32, 62
ML_VIDEO_DITHER_FILTER_INT32, 62
ML_VIDEO_FILL_ALPHA_REAL32, 62
ML_VIDEO_FILL_BLUE_REAL32, 62
ML_VIDEO_FILL_Cb_REAL32, 63
ML_VIDEO_FILL_Cr_REAL32, 63
ML_VIDEO_FILL_GREEN_REAL32, 63
ML_VIDEO_FILL_RED_REAL32, 63
ML_VIDEO_FILL_Y_REAL32, 63
ML_VIDEO_FLICKER_FILTER_INT32, 63
ML_VIDEO_GENLOCK_SIGNAL_PRESENT_INT32, 64
ML_VIDEO_GENLOCK_SOURCE_TIMING_INT32, 64
ML_VIDEO_GENLOCK_TYPE_INT32, 64
ML_VIDEO_GREEN_SETUP_INT32, 64
ML_VIDEO_H_PHASE_INT32, 64
ML_VIDEO_HEIGHT_F1_INT32, 64
ML_VIDEO_HEIGHT_F2_INT32, 65
ML_VIDEO_HUE_INT32, 65
ML_VIDEO_INPUT_DEFAULT_SIGNAL_INT64, 65
ML_VIDEO_NOTCH_FILTER_INT32, 65
ML_VIDEO_OUTPUT_DEFAULT_SIGNAL_INT64, 65
ML_VIDEO_OUTPUT_REPEAT_INT32, 66
ML_VIDEO_PRECISION_INT32, 66
ML_VIDEO_RED_SETUP_INT32, 66
ML_VIDEO_SAMPLING_INT32, 67
ML_VIDEO_SATURATION_INT32, 67
ML_VIDEO_SIGNAL_PRESENT_INT32, 67
ML_VIDEO_START_X_INT32, 67
ML_VIDEO_START_Y_F1_INT32, 67
ML_VIDEO_START_Y_F2_INT32, 68
ML_VIDEO_TIMING_INT32, 68
ML_VIDEO_V_PHASE_INT32, 69
ML_VIDEO_WIDTH_INT32, 69
video sampling, 59
video standards, 143
video/graphics distinction , 1

W

wait handle, 10, 111
WaitForSingleObject, 11
work functions for transcoders, 58