# Digital Media Programming Guide

Document Number 007-1799-050

CONTRIBUTORS

Written by Patricia Creek and Don Moccia
Illustrated by Dany Galgani and Martha Levine
Engineering author contributions by Bent Hagemark, Chris Pirazzi, Angela Lai, Scott Porter, Doug Scott, Mike Travis, Mark Segal, Bryan James, Doug Cook, Nelson Bolyard, Candace Obert, Eric Bloch, Brian Beach, Dan Kinney, and Mike Portuesi



Digital Media Programming Guide
Document Number 007-1799-050

# Contents

# List of Examples

# List of Figures

# List of Tables

# About This Guide

The *Digital Media Programming Guide* describes the Silicon Graphics® digital media development environment (DMdev) software. The DMdev is a family of libraries that provides application program interfaces (APIs) for digital media I/O, file operations, playback, and conversions. This guide describes the libraries and gives technical information on their design and proper use. A companion guide, *Digital Media Programmer's Examples*, contains code samples based on the DMdev to assist your development efforts. It can viewed online using the IRIS InSight™ viewer.

Silicon Graphics also supplies end user desktop media tools, which use the DMdev. Control panels, such as the Video Panel and the Audio Control Panel, are described in the *Media Control Panels User's Guide*. The *Media Tools User's Guide* describes a suite of end user tools for capturing, editing, recording, playing, compressing, and converting audio data and images. You can view both documents from the InSight viewer.

## What This Guide Contains

The *Digital Media Programming Guide* comprises the following sections:

Chapter 1, "Introduction to the Digital Media Libraries" gives an overview of the digital media development environment.

Chapter 2, "Digital Media Essentials" provides a foundation for understanding digital media data characteristics. It reviews how data is represented digitally and then explains how to express data attributes in the digital media libraries.

Chapter 3, "Digital Media Data Types and Parameter Lists" explains how to use the digital media data structures that facilitate data specification and setting, getting, and passing parameters.

Chapter 4, "Digital Media I/O" describes using the digital media library routines that facilitate real-time input and output between live media devices.

Chapter 5, “Digital Media Buffers” explains the Digital Media buffers (DMbuffers) real-time visual data transport facility. The facility establishes a unified approach to providing data flow between live video devices.

Chapter 6, “Digital Media Data Conversion” tells how to use the digital media conversion libraries to implement data format conversion in your application.

Appendix A, “Digital Media Conversion Libraries” contains the APIs of the individual image and audio conversion libraries. These libraries are not discussed in detail, but reference pages to the member functions are cited.

## How to Use This Guide

This guide is written for C language programmers that have some knowledge of digital media concepts. Readers unfamiliar with the basic concepts can refer to Chapter 2, “Digital Media Essentials,” or to the “Suggestions for Further Reading” listed below.

### Where to Start

If you’re not sure which library to use for a certain application, read Chapter 1, “Introduction to the Digital Media Libraries,” to get a brief overview of the uses and features of each library.

If you want to find some code that does what you want your application to do, browse through the *Digital Media Programmer’s Examples* online book to locate a sample program that performs a particular task.

### Style Conventions

These style conventions are used in this guide:

| | |
|---|---|
| **Bold** | functions, routines |
| *Italics* | arguments, variables, commands, program and file names, book titles, and emphasis |
| `Courier` | function prototypes, sample code |
| **`Courier Bold`** | user input entered from the keyboard |

## How to Use the Sample Programs

Code fragments and complete sample programs are used throughout this guide to demonstrate programming concepts. Source code for the sample programs is provided in the */usr/share/src/dmedia* directory, which is further organized in directories according to topic.

*README* files in each directory provide descriptions of the sample programs and instructions for compiling and running them. You must have the IRIS Development Option, *dev*, and the C language software, *c*, loaded before you can compile the sample programs. Use the *versions* command to find out which software is loaded on your system. See the release notes for each library for additional system software requirements for those libraries.

You should copy any program that you intend to modify to your home directory before making any changes.

## Suggestions for Further Reading

This section lists references containing information on programming topics beyond the scope of this guide, which you may find helpful for developing your digital media application. Additional reference materials are listed in the introductory chapters for each library.

### References for Using Digital Media with Other Libraries

If you are planning to integrate your digital media application with calls from the OpenGL™, IRIS Graphics Library™ (GL) or X Window System™ application, you may want to consult the following manuals:

- *OpenGL Programming Guide*, by Jackie Neider, Tom Davis, and Mason Woo, Addison-Wesley, 1994
- *OpenGL Reference Manual*, by Jackie Neider, Tom Davis, and Mason Woo, Addison-Wesley, 1994
- *Graphics Library Programming Guide*, by Patricia McLendon Creek, Silicon Graphics, 1992

- *Graphics Library Programming Tools and Techniques*, by Patricia McLendon Creek and Ken Jones, Silicon Graphics, 1993
- *IRIS IM Programming Notes*, by Patricia McLendon Creek and Ken Jones, Silicon Graphics, 1993
- *The X Window System, Volume 1: Xlib Programming Manual*, O'Reilly and Associates, 1990
- *The X Window System, Volume 4: X Toolkit Intrinsics, Motif Edition*, O'Reilly and Associates, 1990
- *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD*, Third Edition, by Robert W. Scheifler and James Gettys, Digital Press, 1992
- *X Window System Toolkit: The Complete Programmer's Guide and Specification*, Paul J. Asente and Ralph R. Swick, Digital Press, 1992

## References for Adding a User Interface to Your Program

The IRIS Digital Media don't impose any particular user interface (UI), so you can use any graphical UI toolkit, such as IRIS IM™ to build your interface. IRIS IM is Silicon Graphics' port of the industry-standard OSF / Motif™ software. Consult these OSF / Motif manuals for more information:

- *OSF/Motif Programmer's Guide*, Revision 1.2, Prentice-Hall, 1993
- *OSF/Motif Programmer's Reference*, Revision 1.2, Prentice-Hall, 1992
- *OSF/Motif Style Guide*, Revision 1.2, Prentice-Hall, 1992

## Technical References and Standards

The references listed below are some of the more important standards mentioned throughout this book. For more complete listings, you can check the Web sites of organizations such as the Society of Motion Picture & Television Engineers at *http://www.smpte.org/*, and the International Telecommunication Union at *http://www.itu.ch/index.html*.

*SMPTE Standard for Television—Composite Analog Video Signal—NTSC for Studio Applications*, SMPTE 170M-1994, The Society of Motion Picture and Television Engineers

*SMPTE Standard for Television—10-Bit 4:2:2 Component and 4fsc NTSC Composite Digital Signals—Serial Digital Interface*, SMPTE 259M-1993, The Society of Motion Picture and Television Engineers

*SMPTE Standard for Television—Component Video Signal 4:2:2 - Bit-Parallel Digital Interface*, SMPTE 125M-1995, The Society of Motion Picture and Television Engineers

*4:2:2 Digital Video: Background and Implementation Revised Edition*, The Society of Motion Picture and Television Engineers, l995

*Recommendation ITU-R BT.601-5—Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios*, The International Telecommunication Union, 1995

*JPEG Still Image Data Compression Standard*, by William B. Pennebaker and Joan L. Mitchell, Van Nostrand Reinhold, 1993

# Introduction to the Digital Media Libraries

The digital media development environment (DMdev) is a family of libraries that provides application program interfaces (APIs) for digital media I/O, file operations, playback, and conversions.

Figure 1-1 shows how the libraries in the digital media development environment are related to each other and to other development libraries. Lines in Figure 1-1 indicate where some libraries make internal calls to other libraries.

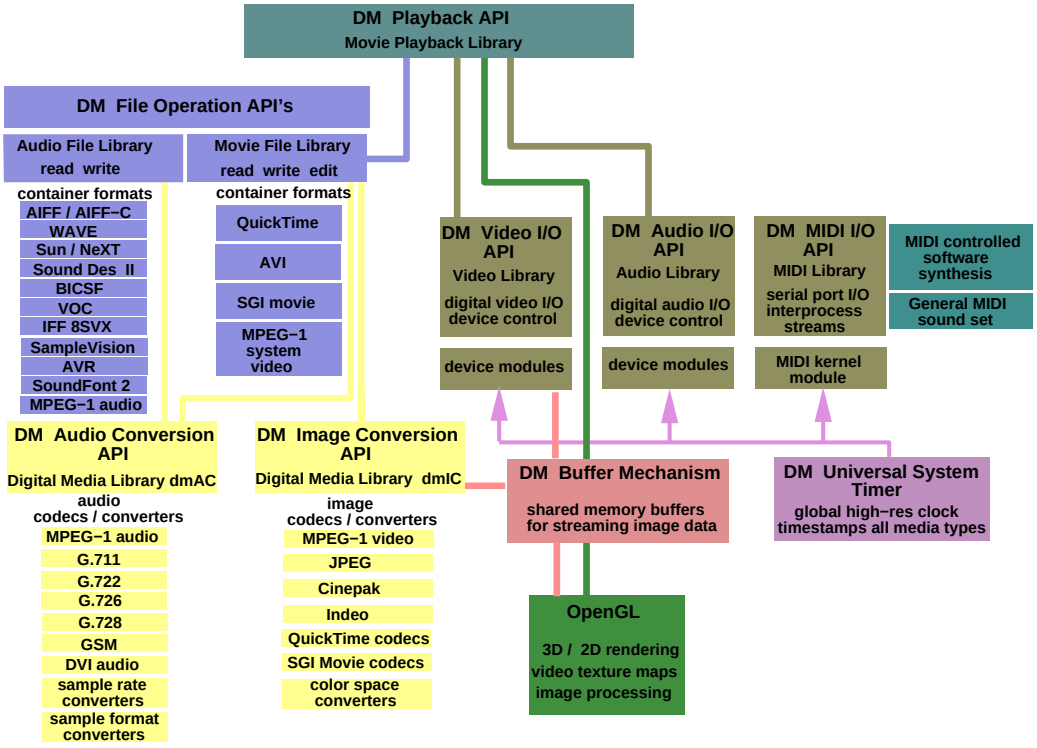



**Figure 1-1** Silicon Graphics Digital Media Programming Environment

Together, the family of Digital Media Libraries encompass facilities for data format description, live audio and video I/O with built-in conversion capabilities, file operations such as reading, writing, and editing multimedia files, data conversion and compression, and playback. The following sections describe these facilities.

## Digital Media Data Specification Facilities

What distinguishes the Digital Media Libraries from other multimedia developer software is the ability to accept any data regardless of format. You aren't limited to working with a particular format or the constraints it imposes. One of the things that makes this possible are the extensive data specification facilities offered by the DMdev.

The DM Library, *libdmedia.so*, provides global data type and parameter definitions for specifying digital media data attributes. You can use DM parameters to describe data held in memory, passed among the Digital Media Libraries, and imported and exported externally. File formats for on-disk data are described in "Digital Media File Operation and Conversion Facilities."

The DM Library features

- type definitions for digital media
- routines for creating and configuring digital media parameters
- routines for creating and configuring digital media buffers
- a debugging version of the library that lets you check for proper usage

See Chapter 3, "Digital Media Data Types and Parameter Lists," for a complete explanation of using the DM data types and parameters.

## Digital Media I/O Facilities

The Audio Library (AL), the Video Library (VL), and the MIDI Library enable real-time I/O by providing the interface between your application program, the workstation CPU, and external devices. An overview of the digital media I/O capabilities follows, and Chapter 4, "Digital Media I/O," provides a complete explanation of using the digital media I/O APIs.

## Audio I/O

The AL provides a device-independent C language API for programming audio I/O on all Silicon Graphics workstations. It provides routines for configuring the audio hardware, managing audio I/O between the application program and the audio hardware, specifying attributes of digital audio data, and facilitating real-time programming.

Use the AL for

- capturing audio from your workstation's microphone, line-level inputs, or a digital audio input source
- playing audio to your workstation's internal speaker, line-level outputs, headphones, or a digital output
- managing audio I/O between multiple audio devices
- adding audio to any application program

## Video I/O

The VL provides an API for transporting live video on Silicon Graphics workstations equipped with on-board video and video options. The VL enables live video flow into a program.

Use the VL for

- displaying live video input in an onscreen window
- capturing video from your workstation's camera input, S-video input, composite analog inputs, or a digital input port into program memory
- playing video from program memory to your workstation's analog or digital video output
- combining video with computer graphics

**Note:** The range of video and VL capabilities you can use depends on the capabilities of your workstation and the video options installed in it.

### MIDI I/O

The MIDI Library provides an API for sending, receiving, processing, and synthesizing musical instrument digital interface (MIDI) messages through the serial interface of Silicon Graphics workstations.

The MIDI Library enables

- content creation through an API to sound synthesizer and sequencer tools
- live interaction through a virtual 3D keyboard

## Digital Media Live Data Transport Facilities

The digital media buffers live data transport system provides data types and operations for sharing and exchanging time-sensitive visual data in real time between video I/O devices, compression devices, graphics rendering and texturing operations, and the host processor(s). It includes

- Digital media buffers (DMbuffers) for carrying digital representations of images
- Digital media buffer pools (DMbufferpools) for reserving, apportioning, and allocating dedicated system and physical memory under direct application control for live visual data processing/transport devices

See Chapter 4, “Digital Media I/O,” for a complete explanation of using DMbuffers.

## Digital Media File Operation and Conversion Facilities

Both the Movie Library and the Audio File Library provide file operations (identifying, opening, reading, writing, and editing files). These routines are capable of supporting a variety of file formats. Note the distinction between a file or container format, which is associated with media stored on disk (or tape), and the data format, which is a collection of attributes that describes the data. File format information is typically contained in a header that immediately precedes the data.

## Audio Files

The Audio File Library, *libaudiofile*, provides a uniform C language API for indentifying, opening, reading, writing, and converting digital audio files of a variety of storage formats.

Use the Audio File Library for

- identifying audio files (multimedia file recognition)
- opening and creating audio files
- seeking, reading, and writing audio files
- setting and retrieving information in audio file headers
- setting and retrieving characteristics of the audio file or the data it contains
- converting audio file formats

## Movie Files

The Movie File Library, *libmoviefile*, provides a file format-independent C language API for reading, writing, editing, and playing movies on Silicon Graphics workstations.

Use the Movie Library for:

- reading, writing, and editing movie files
- converting movie files from one container format to another
- compressing and decompressing movie files
- supporting movies embedded in applications programs

The Movie File Library provides a uniform interface to movies of various formats and lets you convert movies from one format to another. Currently, the Movie Library supports the following file formats:

- Apple® Computer QuickTime™ movie format, including uncompressed data, and JPEG, Indeo®, Cinepak®, Apple Animation, and Apple Video, compression
- Microsoft® audio-video interleaved (AVI) format, including uncompressed data, and JPEG, Indeo, and Cinepak compression
- MPEG-1 systems and video bitstreams

- Silicon Graphics movie format

**Note:** The Digital Media Libraries do not provide a QuickTime API, rather, they provide a file format-independent API that supports QuickTime and several other formats.

**Note:** Some QuickTime track types are not supported by the Movie File Library.

## Digital Media Data Conversion Facilities

The Digital Media Library, *libdmedia*, provides data conversion support for real-time I/O and file operations. Many of the file operations and real-time I/O routines perform automatic data format conversions, by calling these lower level converter APIs, which are also accessible directly from your application:

- dmAC, an audio conversion API that performs audio compression and decompression and audio sample rate conversion
- dmIC, an image conversion API that performs image compression and decompression and color space conversion.

## Digital Media Playback Facilities

The Digital Media Libraries provide a playback API, capable of playing audio, video, movies, and MIDI.

### Movies

The Movie Playback Library, *libmovieplay*, is implemented using calls from the OpenGL and Digital Media Libraries. It provides a scheduler and modules to communicate with output devices. You can take advantage of its built-in playback support in your application.

The movie library playback engine provides:

- asynchronous playback support
- flexible playback control (start, stop, speed, looping)

- the ability to properly combine and blend multiple image and audio tracks
- software scaling of audio

The main advantage of the built-in playback support is that it performs audio and video synchronization for you. Otherwise you would have to calculate the rate for each track and determine the proper display timing. You can still take advantage of this synchronization capability even if you want to use your own display method, by turning off the movie library display and using your own event loop to respond to events. You may also choose to create your own playback using routines from the other libraries.

*Chapter 2*

# Digital Media Essentials

Before writing a digital media application, it's essential to understand the basic attributes of digital image, video, and audio data. This chapter provides a foundation for understanding digital media data characteristics, and how to realize those qualities when creating your application. It begins by reviewing how data is represented digitally and then explains how to express data attributes in the Digital Media Libraries.

## About Digital Media

Data input from analog devices must be digitized in order to store to, retrieve from, and manipulate within a computer. Two of the most important concepts in digitizing are sampling and quantization.

### Sampling and Quantization

Sampling involves partitioning a continuous flow of information, whether with respect to time or space (or both) into discrete pieces. Quantization involves representing the contents of such a sample as an integer value. Both operations are performed to obtain a digital representation.

The topic of exactly how many integers to use for quantizing and how many samples to take (and when or where to take them) in representing a given continuum has received much study because these choices affect the accuracy of the digital representation. Mathematical formulas exist for determining the correct amount of sampling and quantization needed to accurately recreate a continuous flow of data from its constituent pieces. A treatise on sampling theory is beyond the scope of this book, but you should be familiar with concepts such as the Nyquist theorem, pulse code modulation (PCM), and so on. For more information about digitization and related topics, see:

- Poynton, Charles A. *A Technical Introduction to Digital Video*. New York: John Wiley & Sons, 1995 (ISBN 0-471-12253-X)
- Watkinson, John. *An Introduction to Digital Video*, New York: Focal Press, 1994.

Quantities such as the sampling rate and number of quantization bits are called *attributes*; they describe a defining characteristic of the data which has a certain physical meaning. An important point about data attributes is that while they thoroughly describe data characteristics, they do not impose nor imply a particular file format. In fact, one file format might encompass several types of data with several changeable attributes.

*File format*, also called container format, applies to data stored on disk or removable media. Data stored in a particular file format usually has a file header that contains information identifying the file format and auxiliary information about the data that follows it. Applications must be able to parse the header in order to recognize the file at a minimum, and to optionally open, read, or write the file. Similarly, data exported using a particular file format usually has a header prepended to it when output.

In contrast, *data format*, which is described by a collection of attributes, is meaningful for data I/O and exchange, and for data resident in memory. Because the Digital Media Libraries provide extensive data type and attribute specification facilities, they offer lots of flexibility for recognizing, processing, storing, and retrieving a variety of data formats.

## Parameters for Specifying Data Attributes

Parameters in the DM Library include files (*dmedia/dm_*.h)* provide a common language for specifying data attributes for the Digital Media Libraries. Not all of the libraries require or use all of the DM parameters.

Most of the Digital Media Libraries provide their own library-specific parameters for describing data attributes which are often only meaningful for the routines contained within each particular library. These library-specific parameters are prefaced with the initials of their parent library, rather than the initials DM. For example, the Video Library defines its own image parameters in *vl.h*, which are prefaced with the initials VL.

Many of the parameters defined in the DM Library have clones in the other libraries (except for the prefix initials). This makes it easy to write applications that use only one library. Some libraries provide convenience routines for converting a list of DM parameters to a list of library-specific parameters.

It's essential to understand the physical meaning of the attributes defined by each parameter. Knowing the meanings of the attributes enables you to get the intended results from your application and helps you recognize and be able to use DM parameters and their clones throughout the family of Digital Media Libraries.

The sections that follow describe the essential attributes of digital image and audio data, their physical meanings, and the DM parameters that define them.

## Digital Image Essentials

This section presents essential image concepts about color and video.

### Color Concepts

Important color concepts discussed in this section are:

- Colorspace
- Intensity
- Gamma
- Luma and Luminance
- Chroma and Chrominance

#### Colorspace

Two dimensional digital images are composed of a number of individual picture elements (pixels) obtained by sampling an image in 2D space. Each pixel contains the intensity and, for color images, the color information for the region of space it occupies.

Color data is usually stored on a per component basis, with a number of bits representing each component. A color expressed in RGB values is said to exist in the RGB colorspace. To be more precise, a pixel is actually a vector in colorspace. There are other ways to encode color data, so RGB is just one type of colorspace.

There are four colorspaces to know about for the Digital Media Libraries:

- full-range RGB with the following properties:
  - component R, G, B
  - each channel ranges [0.0 - 1.0]; mapped on to [0..2^n-1]
  - alpha is [0.0 - 1.0]; mapped on to [0..2^n-1]

- compressed-range RGB with the following properties:
  - component R, G, B
  - each channel ranges [0.0 - 1.0]; mapped on to [64..940] (10 bit mode) and [16..235] (8 bit mode)
  - alpha is [0.0 - 1.0]; mapped on to [64..940] (10 bit mode) and [16..235] (8 bit mode)
- full-range YUV with the following properties:
  - component Y, Cr, Cb
  - Y (luma) channel ranges [0.0 - 1.0] mapped on to [0..2^n-1]
  - Cb and Cr (chroma) channels range [-0.5 - +0.5] mapped on to [0..2^n-1]
  - alpha is [0.0 - 1.0] mapped on to [0..2^n-1]
  - color space is as defined in Rec 601 specification
- compressed-range YUV with the following properties:
  - component Y, Cr, Cb
  - Y (luma) channel ranges [0.0 - 1.0] mapped on to [64..940] (10 bit mode) and [16..235] (8 bit mode)
  - Cb and Cr (chroma) channels range [-0.5 - +0.5] mapped on to [64..960] (10 bit mode) and [16..240] (8 bit mode)
  - alpha is [0.0 - 1.0] mapped on to [64..940] (10 bit mode) and [16..235] (8 bit mode)
  - color space is as defined in Rec. 601 specification

On the display screen, each pixel is actually a group of three phosphors (red, green, and blue) located in close enough proximity that they are perceived as a single color. There are some issues with anomalies related to the physical properties of screens and phosphors that are of interest for programmers.

### Intensity

Humans perceive brightness in such a way that certain colors appear to be brighter or more intense than others. For example, red appears to be brighter than blue, and green appears to be brighter than either blue or red. Secondary colors (cyan, magenta, and yellow), each formed by combining two primary colors, appear to be even brighter still. This phenomenon is simulated in Figure 2-1.

Figure 2-1 is a plot that simulates the human eye's response to the light intensity of different wavelengths (colors) of light. Pure colors (red, green, blue, yellow, cyan, and magenta) are plotted in YCrCb colorspace with brightness plotted along the horizontal axis. The rightmost colors are perceived as brighter than the leftmost colors even though all the colors are of equal brightness.

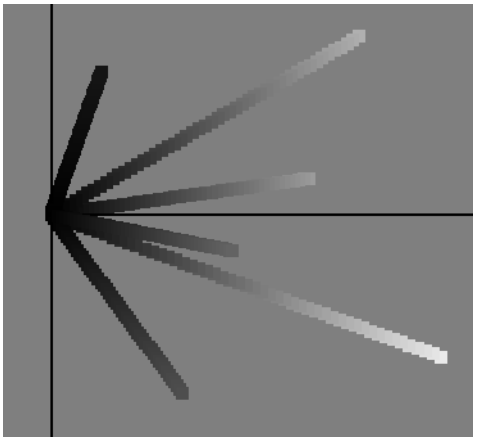


**Figure 2-1** Plot Simulating Human Visual Perception of Brightness vs. Color

The brightness of a color is measured by a quantity called *saturation*. Figure 2-2 connects the maximum saturation points of each color. The lines tracing the path of maximum saturation from the values at each corner are called hue lines.

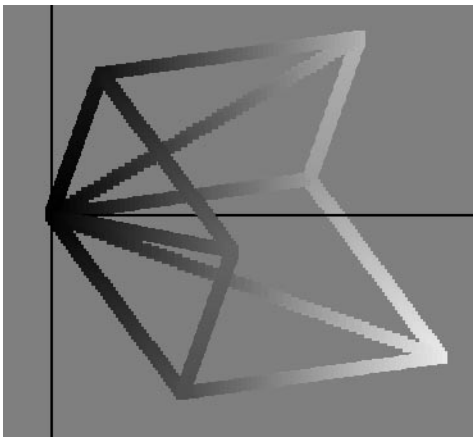


**Figure 2-2** Hue and Saturation

The program that generated Figure 2-1 and Figure 2-2, *colorvu*, uses the Colorspace API described in "The Color Space Library" in Appendix A.

Both the human visual perception of light intensity and the physical equipment used to convey the sensation of brightness in a computer display are inherently nonlinear, but in different ways. As it happens, the differences actually complement each other, so that the nonlinearity inherent in the way a computer display translates voltage to brightness almost exactly compensates for the way the human eye perceives brightness, but some correction is still necessary, as explained in the next section, "Gamma."

### Gamma

Cameras, computer displays, and video equipment must account for both the human visual response and the physical realities of equipment in order to create a believable image. Typically a *gamma* factor is applied to color values in order to correctly represent the visual perception of color, but the way in which the gamma correction is applied, the reason for doing so, and the actual gamma value used depends on the situation.

It is helpful to understand when in the image processing path gamma is applied. Computers apply a gamma correction to output data in order to correctly reproduce intensity when displaying visual data. Cameras compensate for the nonlinearity of human vision by applying a gamma correction to the source image data as it is captured.

The key points to know when working with image data are whether the data has been (or should be) gamma-corrected, and what is the value of the gamma factor. Silicon Graphics monitors apply a default gamma correction of 1.7 when displaying RGB images.

You can customize the gamma function by specifying gamma coefficients for image converters as described in "The Digital Media Color Space Library" in Chapter 6.

### Luma and Luminance

Video uses a *nonlinear* quantity, referred to as *luma*, to convey brightness. Luma is computed as a weighted sum of gamma-corrected RGB components. Color science theory represents the sensation of brightness as a *linear* quantity called *luminance*, which is computed by adding the red, green, and blue components, each weighted by a linear gamma factor that mimics human visual response.

In video and software documentation, the terms luma and luminance are often used interchangeably, and the letter Y can represent either quantity. Some authors (Poynton and others) use a prime symbol to denote a nonlinear quantity, and so represent luma as Y', and luminance as Y. This type of notation emphasizes the difference between a nonlinear and linear quantity, but it is not common practice, so it is important to realize that there are differences. Unless otherwise noted, in this document and in the Digital Media Libraries, Y refers to the nonlinear luma.

### Chroma and Chrominance

In order to supply the color information for video signal encoding, the luma value (Y) is subtracted from the red and blue color components, giving two color difference signals: B–Y (B minus Y) and R–Y (R minus Y), which together are called chroma. As in the case of luma and luminance, the term chrominance is sometimes mistakenly used to refer to chroma, but the two terms signify different quantities. In the strictest sense, chrominance refers to a representation of a color value expressed independently of luminance, usually in terms of chromaticity.

### Chromaticity

Color science uses *chromaticity* values to express absolute color in the absence of brightness. Chromaticity is a mathematical abstraction that is not represented in the physical world, but is useful for computation. Chrominance is often expressed in terms of chromaticity. A CIE chromaticity diagram is a (*x*, *y*) plot of colors in the wavelengths

of visible light (400 nm to 700 nm). Color matching and similar applications require an understanding of chromaticity, but you probably won't need to use it for most applications written using the Digital Media Libraries.

## Video Concepts

Important video concepts to be familiar with include the distinction between digital and analog video, video formats, black level, fields and interlacing. This section contains these topics, which highlight key video concepts:

- YCrCb and Component Digital Video
- YUV and Composite Analog Video
- Black Level
- Video Fields
- Field Dominance

### YCrCb and Component Digital Video

Because the human perception of brightness varies depending on color, some image encoding formats can take advantage of that difference by separating image data into separate components for brightness and color. One such method is the component digital video standard established by ITU-R BT.601 (also formerly known as CCIR Recommendation 601, or often simply Rec. 601).

For the digital video formats, Rec. 601 defines some basic properties common to digital component video, such as pixel sampling rate and color space, regardless of how it is transmitted. Then, the more specific documents (SMPTE 125M, SMPT259M, and ITU-R BT.656) define how the data format defined by Rec. 601 is to be transmitted over various kinds of links (serial, parallel) with various numbers of lines (525,or 625).

Component digital video uses scaled chroma values, called Cb and Cr, which are combined with luma into a signal format called YCrCb. This YCrCb refers to a signal format that is transmitted over a wire. It is related to, but separate from the YCrCb colorspace used to store samples in computer memory.

Recommendation 601 defines methods for subsampling chroma values. The most common subsampling method is 4:2:2, where there is one pair of Cb, Cr samples for

every other Y sample. In 4:4:4 subsampling, there is a luma sample for every chroma sample.

### YUV and Composite Analog Video

There are also composite analog video encoding signals, YUV and YIQ, which are based on color difference signals. In analog composite video, the two color difference signals (U,V or I,Q) are combined into a chroma signal which is then combined with the luma for transmission. NTSC and PAL are the two main standards for encoding and transmitting analog composite video. See SMPTE 170M for more information about analog video broadcast standards.

The important point to realize about YUV is that, like YCrCb, it is calculated by scaling color difference values, but different scale factors are used to obtain YUV than those used for YCrCb. The YUV and YCrCb color spaces are extremely similar to each other, but differ primarily in the ranges of acceptable values for the three components when represented as digital integers. The values of Y, U and V are in the range 0..255 (the SMPTE YUV ranges), while the range for Rec. 601 YCrCb is 16..235 / 240.

It is important to keep these differences in mind when selecting the colorspace for storing data in memory. While the terms YUV and YCrCb are used interchangeably and describe both video signals and a colorspace for encoding data in computer memory, they are separate concepts. Knowing and being able to specify precisely the colorspace of input data and the memory format you want are the keys to obtaining satisfactory results.

Analog video input to your workstation through an analog video connector is digitized and often converted to YCrCb in memory. YCrCb is also the colorspace used in many compression schemes (for example, MPEG and JPEG).On some Silicon Graphics video devices and connectors, component analog such as BetaSP and MII formats are digitized into a full-range YUV representation.

When working with analog video, the main points to be aware of are:

- bandwidth limitations (composite analog video uses a method devised to cope with bandwidth restrictions of early transmission methods for broadcast color television)
- chroma / luma crosstalk
- chroma aliasing

Refer to the video references listed in the introduction of this guide for more information.

### Black Level

A common problem when importing video data for computer graphics display (or outputting synthesized computer graphics to video), is that pictures can look darker than expected or can look somewhat hazy because video and computer graphics use a different color scale. In Rec. 601 video, the *black level* (blackest black) is 16, but in computer graphics, 0 is blackest black. If a picture whose blackest black is 16 is displayed by a system that uses 0 as the blackest black, the image colors are all grayed out as a result of shifting the colors to this new scale. The best results are obtained by choosing the correct colorspace. The black level is related to bias, which sets the reference level for a color scale.

### Video Fields

Video is sampled both spatially and temporally. Video is sampled and displayed such that only half the lines needed to create a picture are scanned at a particular instant in time. This is a result of the historical bandwidth limitations of broadcast video, but it is an important video concept.

A video field is set of image samples, practically coincident in time, that is composed of every other line of an image. Each field in a video sequence is sampled at a different time, determined by the video signal's field rate. Upon display, two consecutive fields are *interlaced*, a technique whereby a video display scans every other line of a video image at a rate fast enough to be undetectable to the human eye. The persistence of the phosphors on the display screen holds the impression of the first set of scan lines just long enough for them to be perceived as being shown simultaneously to the second set of scan lines.[1]

The human eye cannot detect and resolve the two fields in a moving image displayed in this manner, but they are detectable in a still image, such as that shown when you pause a videotape. When you attempt to photograph or videotape a computer monitor using a camera, this effect is visible.

Most video signals in use today, including the major video signal formats you are likely to encounter and work with on a Silicon Graphics computer (NTSC, PAL, and 525- and 625-line Rec. 601 digital video), are field-based rather than frame based. Correctly

[1] Actually, this is only strictly true of tube-based display devices whose electron beams take a whole field time to scan each line across the screen (from left-to-right then top-to-bottom). Array-based display devices change the state of all the pixels on the screen (or all the pixels on a given line) simultaneously.

dealing with fields in software involves understanding the effects of temporal and spatial sampling.

Suppose you have a automatic film advance camera that can take 60 pictures per second, with which you take a series of pictures of a moving ball. Figure 2-3 shows 10 pictures from that sequence (different colors are used to 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.

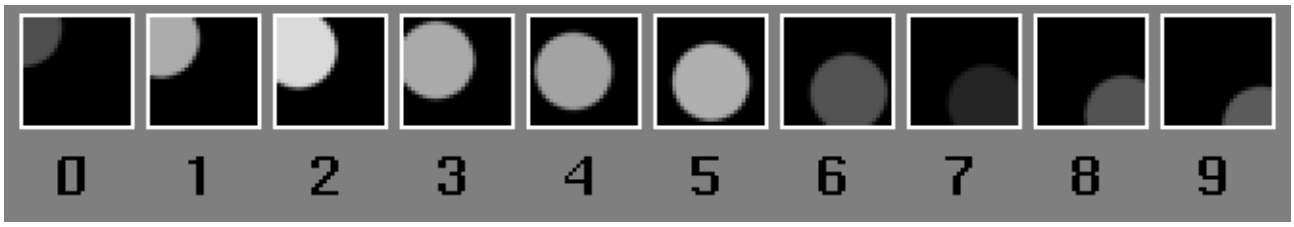



**Figure 2-3** 10 Pictures from a Film Camera Taken at 60 Pictures Per Second

Now suppose you take a modern NTSC video camera and shoot the same sequence. NTSC video has 60 fields per second, so you might think that the video camera would record the same series of pictures as Figure 2-3, but it does not. The video camera does record 60 images per second, but each image consists of only half of the scanlines of the complete picture at a given time, as shown in Figure 2-4, rather than a filmstrip of 10 complete images.

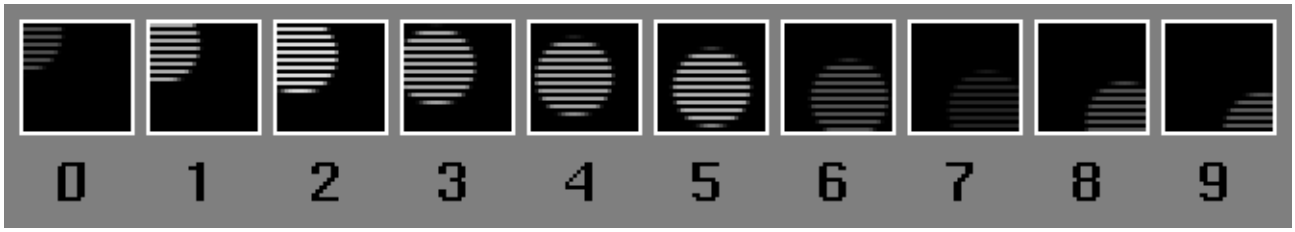



**Figure 2-4** 10 Fields from a 60 Field Per Second Video

Notice how the odd-numbered images contain one set of lines, and the even-numbered images contain the other set of lines (if you can't see this, click on the figure to bring up an expanded view).

Video data does not contain one complete image stored in every other frame, as shown in Figure 2-5.


**Figure 2-5** One Common Misinterpretation of Video Fields

Nor does video data contain two consecutive fields, each containing every other line of an identical image, as shown in Figure 2-6.

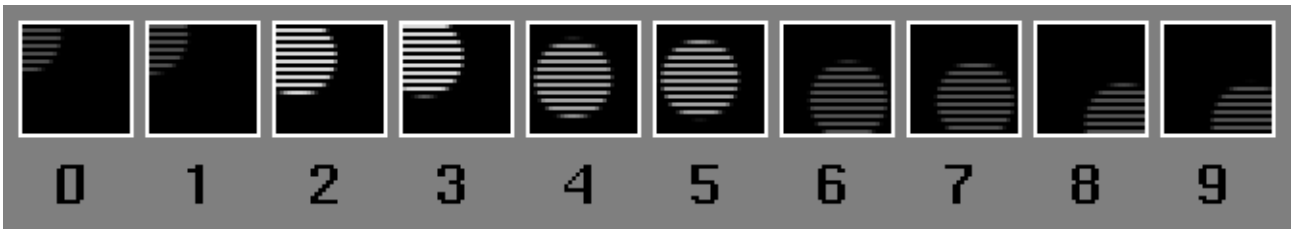



**Figure 2-6** Video is not Pairs of Fields of Identical Images with Alternate Scanlines

Data in video fields are temporally and spatially distinct. In any video sequence, half of the spatial information is omitted for every temporal instant. This is why you cannot treat video data as a sequence of intact image frames. See “Freezing Video” in Chapter 4 for methods of displaying still frames of motion video.

Other video formats, many of which are used for computer monitors, have only one field per frame (often the term field is not used at all in these cases), which is called *noninterlaced* or *progressive scan*. Sometimes, video signals have fields, but the fields are not spatially distinct. Instead, the fields each contain the information for one color basis vector (R, G, and B for example); such signals are called *field sequential*.

It is important to use precise terminology when writing software or communicating with others regarding fields. Some terminology for describing fields is presented next.

Interlaced video signals have a natural two-field periodicity. F1 and F2 are the names given to each field in the sequence. When viewing the waveform of a video field on an

oscilloscope, you can tell whether it is an F1 field or an F2 field by the shape of its sync pulses.

ANSI / SMPTE 170M-1994 defines Field 1, Field 2, Field 3, and Field 4 for NTSC.

ANSI / SMPTE 125M-1992 defines the 525-line version of the bit-parallel digital Rec.-601 signal, using an NTSC waveform for reference. ANSI / SMPTE 259M-1993 defines the 525-line version of the bit-serial digital Rec.-601 signal in terms of the bit-parallel signal. 125M defines Field 1 and Field 2 for the digital signal.

Rec. 624-1-1978 defines Field 1 and Field 2 for 625-line PAL.

Rec. 656 Describes a 625-line version of the bit-serial and bit-parallel Rec.-601 digital video signal. It defines Field 1 and Field 2 for that signal.

We define F1 as an instance of Field 1 or Field 3 and F2 as an instance of Field 2 or Field 4.

### Field Dominance

Field dominance is relevant when transferring data in such a way that frame boundaries must be known and preserved, such as:

- GPI / VLAN / LTC triggered capture or playback of video data
- edits on a VTR
- interpretation of fields in a VLBuffer for the purposes of interlacing or de-interlacing.

Field dominance defines the order of fields in a frame and can be either F1 dominant or F2 dominant.

F1 dominant specifies a frame as an F1 field followed by an F2 field. This is the protocol recommended by all of the above specifications.

F2 dominant specifies a frame as an F2 field followed by an F1 field. This is the protocol followed by several New York production houses for the 525-line formats only.

Most older VTRs cannot make edits on any granularity finer than the frame. The latest generation of VTRs are able to make edits on arbitrary field boundaries, but can (and most often are) configured only to make edits on frame boundaries. Video capture or playback on a computer, when triggered, must begin on a frame boundary. Software

must interlace two fields from the same frame to produce a picture. When software deinterlaces a picture, the two resulting fields are in the same frame.

Regardless of the field dominance, if there are two contiguous fields in a VLBuffer, the first field is always temporally earlier than the second one: under no circumstances should the temporal ordering of fields in memory be violated.

The terms even and odd could refer to whether a field's active lines end up as the even scanlines of a picture or the odd scanlines of a picture. In this case, one needs to additionally specify how the scanlines of the picture are numbered (beginning with 0 or beginning with 1), and one may need to also specify 525 vs. 625 depending on the context.

Even and odd could also refer to the number 1 or 2 in F1 and F2, which is of course a totally different concept that only sometimes maps to the above. This definition seems somewhat more popular.

For example:

- VL_CAPTURE_ODD_FIELDS captures F1 fields
- VL_CAPTURE_EVEN_FIELDS captures F2 fields

The way in which two consecutive fields of video should be interlaced together to produce a picture depends on

- which field is an F1 field and which field is an F2 field
- whether the fields are from a 525- or 625-line signal.

It does not depend on

- the relative order of the fields, that is, which one is first
- anything relating to field dominance

Line numbering in memory does not necessarily correspond to the line numbers in a video specification. Software line numbering can begin with either a 0 or 1. Picture line numbering scheme in software is shown both 0-based (like the Movie Library) and 1-based.

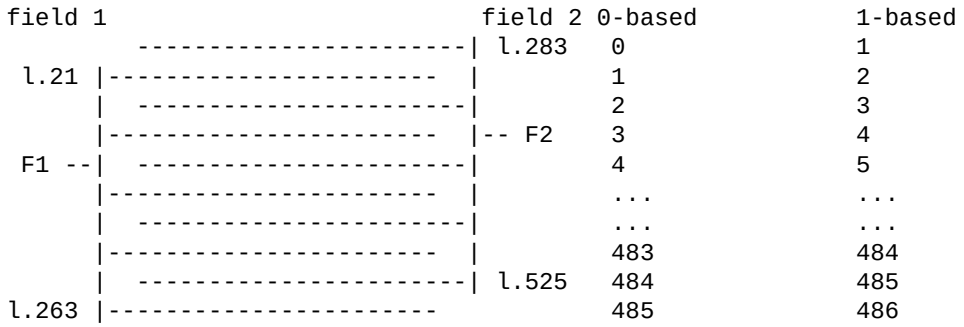
For 525-line analog signals, the picture should be produced in this manner: (F1 has 243 active lines, F2 has 243 active lines, totalling 486 active lines)

```
field 1                          field 2  0-based          1-based
         (second half only)-----| l.283   0               1
 l.21 |-----------------------  |         1               2
      |  -----------------------|         2               3
      |-----------------------  |-- F2    3               4
 F1 --|  -----------------------|         4               5
      |-----------------------  |         ...             ...
      |  -----------------------|         ...             ...
      |-----------------------  |         483             484
      |  -----------------------| l.525   484             485
l.263 |------(first half only)            485             486
```

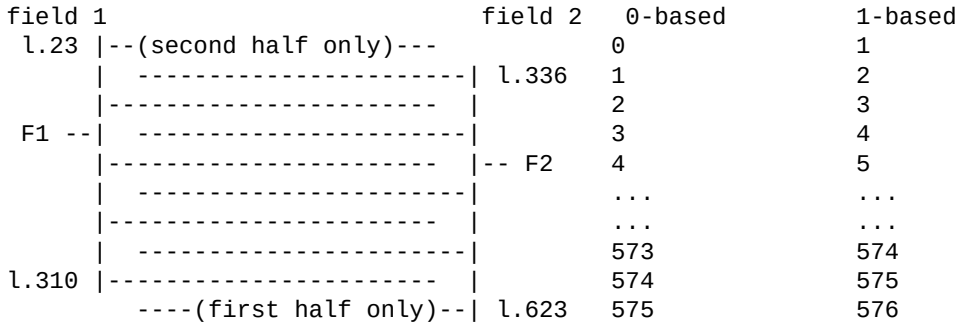
For official 525-line digital signals, the picture should be produced in this manner: (F1 has 244 active lines, F2 has 243 active lines, totalling 487 active lines)

```
field 1                          field 2  0-based          1-based
 l.20 |-----------------------             0               1
      |  -----------------------| l.283   1               2
 l.21 |-----------------------  |         2               3
      |  -----------------------|         3               4
      |-----------------------  |-- F2    4               5
 F1 --|  -----------------------|         5               6
      |-----------------------  |         ...             ...
      |  -----------------------|         ...             ...
      |-----------------------  |         483             486
      |  -----------------------| l.525   484             486
l.263 |-----------------------             486             487
```

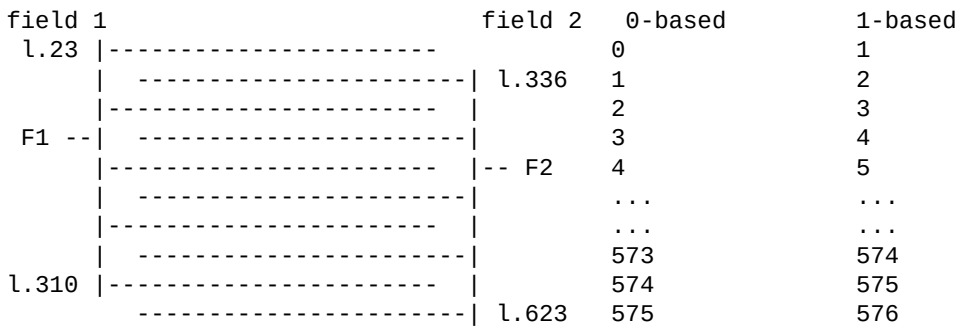
For practical 525-line digital signals, all current Silicon Graphics video hardware skips line 20 of the signal and pretends that the signal has 486 active lines. As a result, you can think of the digital signal as having exactly the same interlacing characteristics and line numbers as the analog signal: (F1 has 243 active lines and F2 has 243 active lines, totalling 486 active lines)




For 625-line analog signals, the picture should be produced in this manner: (F1 has 288 active lines, F2 has 288 active lines)




For 625-line digital signals, the picture should be produced in this manner: (F1 has 288 active lines, F2 has 288 active lines)




All Field 1 and Field 2 line numbers match those in SMPTE 170M and Rec. 624. Both of the digital specs use identical line numbering to their analog counterparts. However, *Video Demystified* and many chip specifications use nonstandard line numbers in some

(not all) of their diagrams. Warning: 125M draws fictitious half-lines in figure 3 in very strange places that do not correspond to where the half-lines fall in the analog signal.

## Digital Image Attributes

This section describes digital image data attributes and how to use them.Image attributes can apply to the image as a whole, to each pixel, or to a pixel component.

Parameters in *dmedia/dm_image.h* provide a common language for describing image attributes for the digital media libraries. Not all of the libraries require or use all of the DM image parameters. Clones of some DM image parameters can be found in *vl.h*.

Digital image attributes described in this section are:

- Image Dimensions
- Pixel Aspect Ratio
- Image Rate
- Image Compression
- Image Quality
- Bitrate
- Keyframe / Reference Frame Distance
- Image Orientation
- Image Interlacing
- Image Pixel Attributes, including
  - Pixel Packing
  - Pixel Component Data Type
  - Pixel Component Data Type
  - Pixel Component Order and Interleaving
- Image Layout
- Image Sample Rate

These attributes and the parameters that represent them are discussed in detail in the sections that follow.

## Image Dimensions

Image size is measured in pixels: DM_IMAGE_WIDTH is the number of pixels in the x (horizontal) dimension, and DM_IMAGE_HEIGHT is the number of pixels in the y (vertical) dimension.

Video streams and movie files contain a number of individual images of uniform size. The image size of a video stream or a movie file refers to the height and width of the individual images contained within it, and is also often referred to as *frame size*.

Some image formats require that the image dimensions be integral multiples of some factor, necessitating either cropping or padding of images that don't conform to those requirements.

**Note:** To determine the size of an image in bytes, use **dmImageGetSize(3dm)**.

## Pixel Aspect Ratio

Pixels aren't always perfectly square, in fact they often aren't.The shape of the pixel is defined by the *pixel aspect ratio*. The pixel aspect ratio is obtained by dividing the pixel height by the pixel width and is represented by DM_IMAGE_PIXEL_ASPECT.

Square pixels have a pixel aspect ratio of 1.0. Some video formats use nonsquare pixels, but computer display monitors typically have square pixels, so a square / nonsquare pixel conversion is needed for the image to look correct when displaying digital video images on the graphics monitor.

In general graphics rendering and display devices typically generate / accept only square pixels, but video I / O devices can typically generate / accept either square or nonsquare formats. It is probably preferable to use / retain a nonsquare format for an application whose purpose is to produce video, while it is probably preferable for an application whose ultimate intent is producing computer graphics to use / retain a square format. Whether a conversion is necessary or optimal depends on the original image source, the final destination, and, to a certain extent, the hardware path transporting the signal.

For example, the digital sampling of analog video in accordance to Rec. 601 yields a nonsquare pixel, while, on the other hand, graphics displays render each pixel as square. This means that a Rec. 601 nonsquare or video input stream sent directly (without filtering) to the workstation's video output displays correctly on an external video monitor, but does not display correctly when sent directly (without filtering) to an onscreen graphics window.

Conversely, computer-originated digital video (640x480 and 768x576) displays incorrectly when sent to video out in nonsquare mode, but displays correctly when sent to an onscreen graphics window or to video out in square mode.

Some Silicon Graphics video devices sample natively using only one format, ether square or nonsquare, and some Silicon Graphics video devices filter signals on certain connectors. See the video device reference pages for details.

Some video options for Silicon Graphics workstations perform square / nonsquare filtering in hardware; refer to your owner's manual to determine whether your video option supports this feature. Software filtering is also possible.

## Image Rate

DM_IMAGE_RATE is the native display rate of a movie file in frames per second.

## Image Compression

Compression is a method of encoding data more efficiently than raw data without changing its content significantly.

A codec (compressor / decompressor) defines a compressed data format. In some cases such as MPEG, the codec also defines a standard file format in which to contain data of that format. Otherwise, there is a set of file formats which can hold data of that format.

A "stateful" algorithm works by encoding the differences between multiple frames, as opposed to encoding each frame independently of the others. Stateful codecs are hard to use in an editing environment but generally produce better compression results because they get access to more redundancy in the data.

A "tile-based" algorithm (such as MPEG) divides the image up into (what is usually) a grid of fixed sections, usually called blocks, macroblocks, or macrocells. The algorithm

then compresses each region independently. Tile-based algorithms are notorious for producing output with visible blocking artifacts at the tile boundaries. Some algorithms specify that the output is to be blurred to help hide the artifacts.

A “transform-based” algorithm (such as JPEG) takes the pixels of the image (which constitute the spatial domain) and transforms them into another domain—one in which data is more easily compressed using traditional techniques (such as RLE, Lempel-Ziv, or Huffman) than the spatial domain. Such algorithms generally do a very good job at compressing images. The computational cost of the transformation is generally high, so:

- Transform-based algorithms are typically more expensive than spatial domain algorithms.
- Transform-based algorithms are typically also tile-based algorithms (since the computation is easier on small tiles), and thus suffer the artifacts of tile-based algorithms.

For most compression algorithms, the compressed data stream is designed so that the video can be played forward or backward, but some compression schemes, such as MPEG, are predictive and so are more efficient for forward playback.

**Note:** In general, JPEG, MPEG, Cinepak, Apple Video and other video compression algorithms are better for compressing camera-generated images; RLE, Apple Animation and other color-cell techniques are better for compressing synthetic (computer-generated) images.

### JPEG Still Video Compression

Although any algorithm can be used for still video images, the JPEG *(Joint Photographic Experts Group*)-baseline algorithm, which is referred to simply as JPEG for the remainder of this guide, is the best for most applications. JPEG is denoted by the DM parameter DM_IMAGE_JPEG.

JPEG is a compression standard for compressing full-color or grayscale digital images.JPEG is a lossy algorithm, meaning that the compressed image is not a perfect representation of the original image, but you may not be able to detect the differences with the naked eye.

JPEG is the preferred standard for compressed digital nonlinear editing because each image is coded separately (intra-coded).

JPEG is based on psychovisual studies of human perception: image information that is generally not noticeable is dropped out, reducing the storage requirement anywhere from 2 to 100 times. JPEG is most useful for still images; it is usable, but slow when performed in software, for video. (Silicon Graphics hardware JPEG accelerators are available for compressing video to and decompressing video from memory or for compressing to and decompressing from a special video connection to a video board. These JPEG hardware accelerators implement a subset of the JPEG standard (baseline JPEG, interleaved YCrCb 8-bit components) especially for video-originated images on Silicon Graphics workstations.

JPEG is typically used to compress each still frame during the writing or editing process, with the intention being to apply another type of compression to the final version of the movie or to leave it uncompressed. JPEG works better on high-resolution, continuous-tone images such as photographs, than on crisp-edged, high-contrast images like line drawings.

The amount of compression and the quality of the resulting image are independent of the image data. The quality depends on the compression ratio. You can select the compression ratio that best suits your application needs.

See also jpeg(4).

See also Pennebaker, William B. and Joan L. Mitchell, *JPEG Still Image Data Compression Standard*, New York: Van Nostrand Reinhold, 1993 (ISBN 0-442-01272-1).

### MPEG-1

MPEG-1 (ISO / IEC 11172) is the *Moving Pictures Expert Group* standard for compressing audio, video, and systems bitstreams. Each bitstream type has its own syntax, as defined by the standard.

The MPEG-1 systems specification defines multiplexing for compressed audio and video bitstreams without performing additional compression. An MPEG-1 encoded systems bitstream contains compressed audio and video data which has been packetized and interleaved along with timestamp and decoder buffering requirements. MPEG-1 allows for multiplexing of up to 32 compressed audio and 16 compressed video bitstreams.

MPEG-1 Video (ISO / IEC 11172-2) is a motion video compression standard that minimizes temporal and spatial data redundancies in order to achieve good image quality at higher compression ratios than either JPEG or MVC1.

MPEG-1 video uses a technique called *motion estimation* or *motion search* that compresses a video stream by comparing image data in nearby image frames. For example, if a video shows the same subject moving against a background, it's likely that the same foreground image appears in adjacent frames, offset by a few pixels. Compression is achieved by storing one complete image frame, which is called a *keyframe* or *I* frame, then comparing an *n*×*n* block of pixels to nearby pixels in proximal frames, searching for the same (or very similar) block of pixels, and then storing only the offset for the frames where a match is located. Images from the intervening frames can then be reconstructed by combining the offset data with the keyframe data.

There are two types of intervening frames:

- P (predictive) frames, which require information from previous P or I frames in order to be decoded. P frames are also sometimes considered as forward reference frames because they contain information that is needed to decode other P frames later in the video bitstream.
- B (between) frames, which require information from both the previous and next P or I frame.

Figure 2-7 shows the relationships between I, P, and B frames.

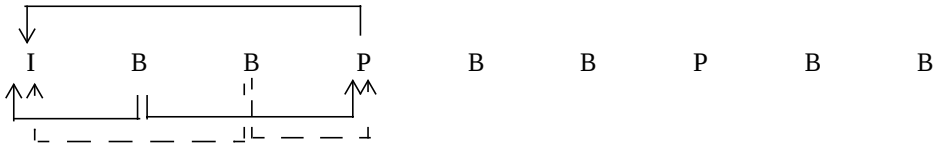



**Figure 2-7** MPEG I, P, and B frames

For example, suppose an MPEG-1 video bitstream contains the sequence $I_0\ P_3\ B_1\ B_2\ P_6\ B_4\ B_5\ P_9\ B_7\ B_8$, where the subscripts indicate the order in which the frames are to be displayed. You must first display $I_0$ and retain its information in order to decode $P_3$, but you cannot yet display $P_3$ because you must first decode and display the two between frames ($B_1$ and $B_2$), which also require information from $P_3$, as well as from each other, in order to be decoded. Once $B_1$ and $B_2$ have been decoded and displayed, you can then display $P_3$, but you must retain its information in order to decode $P_6$, and so on.

MPEG is an asymmetric coding technique—compression requires considerably more processing power than decompression because MPEG examines the sequence of frames and compresses it in a optimized way, including compressing the difference between

frames using motion estimation.This makes MPEG well suited for video publishing, where a video is compressed once and decompressed many times for playback. Because MPEG is a predictive scheme, it is tuned for random access (editing) due to its inter-coding or for forward playback rather than backward. MPEG is used on Video CD, DVD, Direct TV, and is the proposed future standard for digital broadcast TV.

See also mpeg(4).

#### Run Length Encoding

Run-length encoding (RLE) compresses images by replacing pixel values that are repeated for several pixels in a row with a single pixel at the first occurrence of a particular value, followed by a run-length (a count of the number of subsequent pixels of the same value) every time the color changes. Although this algorithm is lossless, it doesn't save as much space as the other compression algorithms—typically less than 2:1 compression is achieved. It is a good technique for animations where there are large areas with identical colors. The Digital Media Libraries have two RLE methods:

DM_IMAGE_RLE
: specifies lossless RLE encoding for 8-bit RGB data. It is the only algorithm currently available to directly compress 8-bit RGB data.

DM_IMAGE_RLE24
: specifies lossless RLE encoding for 24-bit RGB data.

#### Silicon Graphics Motion Video Compressor

Motion Video Compressor (MVC) is a Silicon Graphics proprietary algorithm that is a good general-purpose compression scheme for movies. MVC is a color-cell compression technique that works well for video, but can cause fuzzy edges in high-contrast animation. There are 2 versions:

DM_IMAGE_MVC1
: is a fairly lossy algorithm that does not produce compression ratios as high as JPEG, but it is well suited to movies.

DM_IMAGE_MVC2
: provides results similar to MVC1 in terms of image quality. MVC2 compresses the data more than MVC1, but takes longer to perform the compression. Playback is faster for MVC2, because there is less data to read in, and decompression is faster than for MVC1.

### QuickTime Compression

QuickTime is an Apple Macintosh® system software extension that can be installed in the Macintosh to extend its capabilities so as to allow time-based (audio, video, and animation) data for multimedia applications.

QuickTime movies store and play picture tracks and soundtracks independently of each other, analogous to the way the Movie Library stores separate image and audio tracks. You can't work with pictures and sound as separate entities using the QuickTime Starter Kit utilities on the Macintosh, but you can use the Silicon Graphics Movie Library to work with the individual image and audio tracks in a QuickTime movie.

QuickTime movie soundtracks are playable on both Macintosh and Silicon Graphics computers, but each has its own unique audio data format, so audio playback is most efficient when using the native data format and rate for the computer on which the movie is playing.

The Macintosh QuickTime system software extension includes five codecs:

- Apple None (uncompressed)
- Apple Photo (JPEG standard)
- Apple Animation
- Apple Video
- Apple Compact Video

#### Apple None

Apple None creates an uncompressed movie and can be used to change the number of colors in the images and/or the recording quality. Both the number of colors and the recording quality can affect the size of the movie.

To create an uncompressed QuickTime movie on the Macintosh, click on the "Apple None" choice in the QuickTime Compression Settings dialog box.

**Note:** Because the Macintosh compresses QuickTime movies by default, you must set the compression to Apple None and save the movie again to create an uncompressed movie.

### Apple Photo

Apple Photo uses the JPEG standard. JPEG is best suited for compressing individual still frames, because decompressing a JPEG image can be a time-consuming task, especially if the decompression is performed in software. JPEG is typically used to compress each still frame during the writing or editing process, with the intention to apply another type of compression to the final version of the movie or to leave it uncompressed.

### Apple Animation

Apple Animation uses a lossy run-length encoding (RLE) method, which compresses images by storing a color and its run-length (the number of pixels of that color) every time the color changes. Apple Animation is not a true lossless RLE method because it stores colors that are close to the same value as one color. This method is most appropriate for compressing images such as line drawings that have highly contrasting color transitions and few color variations.

### Apple Video

Apple Video uses a method whose objective is to decompress and display movie frames as fast as possible. It compresses individual frames and works better on movies recorded from a video source than on animations.

**Note:** Both Apple Animation and Apple Video compression have a restriction that the image width and height be a multiple of 4. Before transferring a movie from a Macintosh to a Silicon Graphics computer, make sure that the image size is a multiple of 4.

### Cinepak

Cinepak (developed by Radius, Inc.), otherwise known as “Compact Video,” is a compressed data format which can be stored inside Quicktime movies. It achieves better compression ratios than Quicktime but takes much more CPU time to compress.

Cinepak is designed to control its own bitrate, and thus it is extremely common on the world wide web and is also used in CD authoring.

Cinepak is not a transform-based algorithm. It uses techniques derived from “vector quantization” (which technically is also what color-cell compression techniques like MVC1 and MVC2 use) to represent small tiles of pixels using a small set of scalars. Cinepak builds and constantly maintains a “codebook” which it uses to map the

compressed scalars back into pixel tiles. The codebook evolves over time as the image changes, thus this algorithm is quite stateful.

#### Indeo

Indeo (developed by Intel Corporation) is a compressed data format that can be used in QuickTime and AVI movies.

## Image Quality

Compressed data isn't always a perfect representation of the original data. Information can be lost in the compression process. A *lossless* compression method retains all of the information present in the original data. Algorithms can be either numerically lossless or mathematically lossless. Numerically lossless means that the data is left intact. Mathematically lossless means that the compressed data is acceptably close to the original data.

A *lossy* compression method does not preserve 100% of the information in the original method.

Image quality is a measure of how true the compression is to the original image. Image quality is one of the conversion controls that you can specify for an image converter. Image quality is specified in both the spatial and temporal domains.

In a spatial approximation, pixels from a single image are compared to each other and identical (or similar) pixels are noted as repeat occurrences of a stored representative pixel. Spatial quality, denoted by DM_IMAGE_QUALITY_SPATIAL, conveys the exactness of a spatial approximation.

In a temporal approximation, pixels from an image stream are compared across time and identical (or similar) pixels are noted as repeat occurrences of a stored representative pixel, but offset in time. Temporal quality, denoted by DM_IMAGE_QUALITY_TEMPORAL, conveys the exactness of a temporal approximation.

Some lossless algorithms may require a quality factor, so specify DM_IMAGE_QUALITY_LOSSLESS.

Quality values range from 0 to 1.0, where 0 represents complete loss of the image fidelity and 1.0 represents an lossless image fidelity. You can set both quality factors numerically, or you can use the following rule-of-thumb factors to set quality informally:

| | |
|---|---|
| DM_IMAGE_QUALITY_MIN | approximately equal to 0 quality factor |
| DM_IMAGE_QUALITY_LOW | approximately equal to 0.25 quality factor |
| DM_IMAGE_QUALITY_NORMAL | approximately equal to 0.5 quality factor |
| DM_IMAGE_QUALITY_HIGH | approximately equal to 0.75 quality factor |
| DM_IMAGE_QUALITY_MAX | approximately equal to 0.99 quality factor |

Using these “fuzzy” quality factors can be useful if you application uses a thumbwheel or slider to let the end user indicate quality. These quality factors can be assigned to intermediate steps in the slider or thumbwheel to give the impression of infinitely adjustable quality.

## Bitrate

The compression ratio is a tradeoff between the quality and the bitrate. Adjusting either one of these parameters effects the other, and, if both are set, bitrate usually takes precedence in the Silicon Graphics Digital Media Libraries.

For applications that require a constant bitrate, such as applications that send data over fixed data rate carriers or playback image streams at a minimum threshold rate, set DM_IMAGE_BITRATE. The picture quality is then adjusted to achieve the stated rate. Some Silicon Graphics algorithms guarantee the bitrate, some try to achieve the stated rate, and some do not support a bitrate parameter.

## Keyframe/Reference Frame Distance

Certain compression algorithms such as MPEG use a technique called *motion estimation*, which compresses an image stream by storing a complete keyframe and then encoding related image data in nearby image frames, as described in “MPEG-1.” Images from the encoded frames are decoded based on the keyframes or other encoded frames that precede or follow the frame being decoded.

The Digital Media Libraries have their own terminology to define the 3 different types of frames possible in a motion estimation compression method:

| | |
|---|---|
| Intra | depends only on itself; contains all data needed to construct a complete image. Also called I frame or keyframe. |
| Inter | depends on a previous inter or intra frame. Also called reference frame, P (predictive) frame, or delta frame. |
| Between | depends on previous *and* next inter or intra frame; cannot be reconstructed using another between frame. Also called B frame. |

There are two parameters for setting the distance between keyframes and reference frames:

| | |
|---|---|
| DM_IMAGE_KEYFRAME_DISTANCE | specifies the distance between keyframes |
| DM_IMAGE_REFFRAME_DISTANCE | specifies the distance between reference frames |

## Image Orientation

Image orientation refers to the relative ordering of the horizontal scan lines within an image. The scanning order depends on the image source and can be either top-to-bottom or bottom-to-top, but it is important to know which. The default DM_IMAGE_ORIENTATION for images created on a Silicon Graphics workstation is bottom-to-top, denoted by DM_IMAGE_BOTTOM_TO_TOP. Video and compressed video is typically oriented top-to-bottom.

## Image Interlacing

Interlacing is a video display technique that minimizes the amount of video data necessary to display an image by exploiting human visual acuity limitations. Interlacing weaves alternate lines of two separate fields of video at half the scan rate. For an explanation of interlacing, see “Video Fields.”

Generally, interlacing refers to a technique for signal encoding or display, and interleaving refers to a method of laying out the lines of video data in memory.

Interleaving can also refer to how the samples of an image's different color basis vectors are arranged in memory, or how audio and video are arranged together in memory. Interleaving image pixel data is described in "Pixel Component Order and Interleaving."

A movie file encodes pairs of fields into what it calls frames, and all data transfers are on frame boundaries. A 2-field image in a movie file does not always represent a complete video frame because it could be clipped, or not derived from video. This is further complicated by that fact that both top-to-bottom and bottom-to-top ordering of video lines in images is supported.

DM_IMAGE_INTERLACING describes the original interlacing characteristics of the signal that produced this image (or lack of interlacing characteristics).

In a zero-based picture line numbering scheme for noninterlaced images:

- In a DM_IMAGE_INTERLACED_ODD image, the scanlines of the first field occupy the odd-numbered lines (1, 3, 5, 7, and so on).
- In a DM_IMAGE_INTERLACED_EVEN image, the scanlines of the first field occupy the even-numbered lines (0, 2, 4, 8, and so on).

In this sense, first field means the image that is first temporally and in memory.

**Note:** If the DM_IMAGE _ORIENTATION is DM_BOTTOM_TO_TOP instead of DM_TOP_TO_BOTTOM, then all temporal ordering and memory ordering rules are reversed.

For an example of how DM_IMAGE_INTERLACING relates to video, consider a top-to-bottom buffer containing unclipped video data (a buffer containing all the video lines described for analog 525, practical digital 525, analog 625, and digital 625-line signals). The buffer's DM_IMAGE_INTERLACING depends on many factors.

For a signal with F1 dominance, a frame consists of an F1 field followed by an F2 field (temporally and in memory). The DM_IMAGE_INTERLACING parameter determines which picture lines contain the first field's data:

- for an analog or practical digital 525-line image, DM_IMAGE_INTERLACED_ODD
- for an analog or digital 625-line image, DM_IMAGE_INTERLACED_EVEN

However, if the signal has F2 dominance, where a frame consists of F2 followed by F1, the first field is now an F2 field so we have:

- for an analog or practical digital 525-line image, DM_IMAGE_INTERLACED_EVEN
- for an analog or digital 625-line image, DM_IMAGE_INTERLACED_ODD

## Image Layout

DM_IMAGE_LAYOUT describes how pixels are arranged in an image buffer. In the DM_IMAGE_LAYOUT_LINEAR layout, lines of pixels are arranged sequentially. This is the typical image layout for most image data.

DM_IMAGE_LAYOUT_GRAPHICS and DM_IMAGE_LAYOUT_MIPMAP are two special layouts optimized for presentation to Silicon Graphics hardware. Both are passthrough formats; they are intended for use with image data that is passed untouched from a Silicon Graphics graphics or video input source directly to hardware. Use DM_IMAGE_LAYOUT_GRAPHICS to format image data sent to graphics display hardware. Use DM_IMAGE_LAYOUT_MIPMAP to format image data which represents a texture mipmap that is sent to texture memory, such as a video texture.

## Image Pixel Attributes

This section describes image attributes that are specified on a per pixel or per pixel component basis. Understanding these attributes requires some familiarity with the color concepts described in “Digital Image Essentials.”

### Pixel Packing

Pixel packing formats define the bit ordering used for packing image pixels in memory. Native packings are those packings which are supported directly in hardware. In other words, native packings don’t require a software conversion.

DM_IMAGE_PACKING parameters describe pixel packings recognized by the dmIC and Movie Library APIs. In addition to the DM_IMAGE_PACKING formats, there is also a set of VL_PACKING parameters in *vl.h* that describe image packings. There are some VL_PACKINGS that have no corresponding DM_IMAGE_PACKINGS.

For some packings, the DM_IMAGE_DATATYPE parameter controls how data is packed within the pixel. For example, 10-bit per pixel data can be left or right-justified in a 16-bit word.

The most common ways of packing data into memory are YCrCb and 32-bit RGBA.

**YCrCb (4:2:2) Video Pixel Packing**

Rec. 601 component digital video (4:2:2 subsampled) is composed of one 8-bit Y (luma) component per pixel, and two chroma samples coincident with alternate luma samples, supplying one 8-bit Cr component per two pixels, and one 8-bit Cb sample per two pixels. This results in 2 bytes per pixel. This is the Silicon Graphics native format for storing video image data in memory, which is represented by the DM_IMAGE_PACKING parameter DM_IMAGE_PACKING_CbYCrY, and the VL_PACKING parameter VL_PACKING_YVYU_422_8.

**Note:** The SMPTE 259M (specification for transmitting Rec. 601 over a link) digital video stream contains 10 bits in each component. An 8-bit packing format such as VL_PACKING_YVYU_422_8 uses only 8 of the 10 bits. This often generates acceptable results for strictly video data, but in order to parse some forms of ancillary data (such as embedded audio data) from a video stream, it is necessary to input all 10 bits. Because 10 bits is an atypical quantity for computers, the most common technique is to left-shift each 10-bit quantity to a 16-bit value, resulting in a 4-byte per component format called VL_PACKING_YVYU_422_10, where the extra bits are zero-padded on input and ignored on output. Storing data in this format takes more memory space, but may be preferable to the cost of manipulating 10-bit packed data on the CPU.

The pixel packing is independent of the color space. The use of a packing named “YUV” or “YVYU” does not imply that the data packed is YUV data, as opposed to YCrCb data. When YCrCb data is being packed with a YUV packing, the Cr component is packed as U, and the Cb component is packed as V. The VL_PACKING_YVYU_422_8 packing is the only packing that is natively supported in hardware (requiring no software conversion) on all VL video devices.

The 422 designation in the packing name means that the pixels are packed so that each horizontally-adjacent pair of pixels share one common set of chroma (for example, UV, or alternatively, CrCb) data. Each pixel has its own value of luma (Y) data. So, data is packed in pairs of two pixels, two Y values, and one U and one V (or alternatively, one Cr and one Cb) value pair, in each pixel pair. This pixel packing always has the number of pixels in each row will always be even.

The YUV and YCrCb color spaces are similar, but they differ primarily in the ranges of acceptable values for the three components when represented as digital integers. The values of Y, U and V are in the range 0..255 (the SMPTE YUV ranges), while the range for Rec. 601 YCrCb is 16..235 / 240.

The set of VL packings presently defined does not enable the application to choose between the YUV and Rec.-601 YCrCb color spaces. When an application specifies VL_PACKING_YVYU_422_8, the resultant color space is either YUV or YCrCb, depending on the device and the source node from which the data is coming. Most external digital sources produce YCrCb data. IndyCam produces Rec. 601-compliant YCrCb. There is no way to tell, from the VL_PACKING control, which of those two spaces (YUV or YCrCb) is used.

Each of the different VL video devices has a different set of color spaces and packings implemented in hardware. Any other color spaces and / or packings are implemented by means of a software conversion. The table below shows which color-space and packing combinations are implemented in hardware, or software, or not at all, for each device.

The chipset used in VINO and EV1 to convert analog input to digital pixels produces YUV output, not YCrCb output. That is, the values of Y, U and V are in the range 0..255 (the SMPTE YUV ranges), not the smaller 16..235 / 240 range specified for Rec. 601 YCrCb. For some devices that can't convert color space in hardware, e.g. EV1, the VL converts from YUV to RGBX / RGBA in software.

The VL routines used for this purpose assume the input is Rec. 601 YCrCb, not YUV, regardless of what the hardware actually produces. Therefore, if the hardware doesn't support the desired color space, and you require an accurate color space conversion, then specify pixels in a color space supported by the hardware, and do the color space conversion using dmIC or similar software converter, rather than relying on an automatic software colorspace conversion.

With Sirius Video, color space and packing are independent. Color space is chosen by the settings of the VL_FORMAT on the memory drain node, according to table below, and any packing can be applied to any color space, whether it makes sense or not. Color space conversion occurs when the VL_FORMAT of the video source node and the VL_FORMAT of the memory drain node imply different color spaces.

### 32-bit RGBA Graphics Pixel Packing

In 32-bit RGBA, the A may be a don't care or it may be an alpha channel, synthesized on the computer. This results in 4 bytes per pixel. In the VL, this is called VL_PACKING_RGBA_8, VL_PACKING_RGB_8, and VL_PACKING_ABGR_8.

Table 2-1 shows the results in memory of reading pixels (or the source for writing pixels) in various formats. Pixel 0 is the leftmost pixel read or written. An `x' means don't care (this bit is not used).

Memory layout is presented in 32-bit words, with the MSB on the left and the LSB on the right (read the bit numbers vertically).

**Table 2-1** Pixel Packing Formats

| MSB | | | LSB | Packing Format |
|---|---|---|---|---|
| **33222222** | **22221111** | **111111** | | |
| **10987654** | **32109876** | **54321098** | **76543210** | <----Bit numbers |
| **bbgggrrr** | **bbgggrrr** | **bbgggrrr** | **bbgggrrr** | DM_IMAGE_PACKING_8BGR<br>VL_PACKING_RGB_332_P |
| **aaaaaaaa** | **bbbbbbbb** | **gggggggg** | **rrrrrrrr** | DM_IMAGE_PACKING_ABGR<br>VL_PACKING_RGBA_8 |
| **xxxxxxxx** | **bbbbbbbb** | **gggggggg** | **rrrrrrrr** | DM_IMAGE_PACKING_XBGR<br>VL_PACKING_RGB_8 |
| **uuuuuuuu** | **yyyyyyyy** | **vvvvvvvv** | **yyyyyyyy** | DM_IMAGE_PACKING_CbYCrY |
| **uuuuuuuu** | **yyyyyyyy** | **vvvvvvvv** | **yyyyyyyy** | VL_PACKING_YVYU_422_8 |
| | | | | DM_IMAGE_PACKING_RBG323<br>VL_PACKING_RBG_323 |
| **xxxxxxxx** | **xxxxxxxx** | **xxxxxxxx** | **bbgggrrr** | DM_IMAGE_PACKING_BGR233<br>VL_PACKING_RGB_332 |
| **xxxxxxxx** | **xxxxxxxx** | **xxxxxxxx** | **rrrgggbb** | VL_PACKING_BGR_332 |
| **bbgggrrr** | **bbgggrrr** | **bbgggrrr** | **bbgggrrr** | VL_PACKING_RGB_332_IP |
| **bbgggrrr** | **bbgggrrr** | **bbgggrrr** | **bbgggrrr** | |
| **rrrgggbb** | **rrrgggbb** | **rrrgggbb** | **rrrgggbb** | VL_PACKING_BGR332_P |
| **rrrgggbb** | **rrrgggbb** | **rrrgggbb** | **rrrgggbb** | |

**Table 2-1 (continued)** Pixel Packing Formats

| MSB | | | LSB | Packing Format |
|---|---|---|---|---|
| xxxxxxx | xxxxxxx | bbbbbggg | gggrrrrr | VL_PACKING_RGB-565 |
| bbbbbggg | gggrrrrr | bbbbbggg | gggrrrrr | VL_PACKING_RGB_565_P |
| rrrrrrrr<br>gggggggg | bbbbbbbb<br>rrrrrrrr | gggggggg<br>bbbbbbbb | rrrrrrrr<br>gggggggg | VL_PACKING_RGB_565_IP |
| xxbbbbbb | bbbbgggg | gggggrr | rrrrrrrr | VL_PACKING_RGB_10 |
| yyyyyyyy | yyyyyyyy | yyyyyyyy | yyyyyyyy | DM_IMAGE_PACKING_LUMINANCE<br>VL_PACKING_Y_8_IP |
| xxxxxxxx | uuuuuuuu | yyyyyyyy | vvvvvvvv | DM_IMAGE_PACKING_CbYCr<br>VL_PACKING_YUV_444_8 |
| aaaaaaaa | uuuuuuuu | yyyyyyyy | vvvvvvvv | VL_PACKING_YUV_4444_8 |
| xxuuuuuu | uuuuyyyy | yyyyyyvv | vvvvvvvv | VL_PACKING_YUV_444_10 |
| rrrrrrrr | gggggggg | bbbbbbbb | aaaaaaaa | DM_IMAGE_PACKING_RGBA<br>VL_PACKING_ABGR_8 |
| vvvvvvvv | yyyyyyyy | uuuuuuuu | aaaaaaaa | DM_IMAGE_PACKING_CbYCrA<br>VL_PACKING_AUYV_8 |
| rrrrrrrr | rrgggggg | ggggbbbb | bbbbbbaa | VL_PACKING_A_2_BGR_10 |
| vvvvvvvv | vvyyyyyy | yyyyuuuu | uuuuuuaa | VL_PACKING_A_2_UYV_10 |
| uuuuuuuu | uuyyyyyy | yyyyaaaa | aaaaaaxx | VL_PACKING_AYU_AYV_10 |

Table 2-2 lists DM_IMAGE_PACKING formats.

**Table 2-2** DM Pixel Packing Formats

| Pixel Packing Format |
|---|
| DM_IMAGE_PACKING_RGB |
| DM_IMAGE_PACKING_BGR |
| DM_IMAGE_PACKING_RGBX |
| DM_IMAGE_PACKING_RGBA |
| DM_IMAGE_PACKING_XRGB |
| DM_IMAGE_PACKING_ARGB |
| DM_IMAGE_PACKING_XBGR |
| DM_IMAGE_PACKING_ABGR |
| DM_IMAGE_PACKING_RBG323 |
| DM_IMAGE_PACKING_BGR233 |
| DM_IMAGE_PACKING_XRGB1555 |
| DM_IMAGE_PACKING_CbYCr |
| DM_IMAGE_PACKING_CbYCrA |
| DM_IMAGE_PACKING_CbYCrY |
| DM_IMAGE_PACKING_CbYCrYYY |
| DM_IMAGE_PACKING_LUMINANCE |
| DM_IMAGE_PACKING_LUMINANCE_ALPHA |

### Pixel Component Data Type

DM_IMAGE_DATATYPE describes the number of bits per component and the alignment of the bits within the pixel.Table 2-3 lists the data type parameters and the attributes they describe.

**Table 2-3** Image Data Types

| Image Data Type Parameter | Attributes |
|---|---|
| DM_IMAGE_DATATYPE_BIT | Nonuniform number of bits per component |
| DM_IMAGE_DATATYPE_CHAR | 8 bits per component |
| DM_IMAGE_DATATYPE_SHORT10L | 10 bits per component, left aligned |
| DM_IMAGE_DATATYPE_SHORT10R | 10 bits per component, right aligned |
| DM_IMAGE_DATATYPE_SHORT12L | 12 bits per component, left aligned |
| DM_IMAGE_DATATYPE_SHORT12R | 12 bits per component, right aligned |

### Pixel Component Order and Interleaving

DM_IMAGE_ORDER describes the order of pixel components or blocks of components within an image and has one of the following formats:

- DM_IMAGE_ORDER_INTERLEAVED orders pixels component-by-component
- DM_IMAGE_ORDER_SEQUENTIAL groups like components together line-by-line
- DM_IMAGE_ORDER_SEPARATE groups like components together per image

Table 2-4 shows the resultant pixel component order for each interleaving method for some example image formats.

**Table 2-4** Pixel Interleaving Examples

| Packing Format | Interleaved | Sequential | Separate |
|---|---|---|---|
| ABGR | ABGRABGR<br>ABGRABGR<br>ABGRABGR | AAABBBGGGRRR<br>AAABBBGGGRRR | AAAAAAAA<br>BBBBBBBBB<br>GGGGGGGGG<br>RRRRRRRRR |
| 444 YCrCb,<br>with CbYCr packing | CbYCrCbYCr<br>CbYCr | CbCbCbYYYCrCrCr<br>CbCbCbYYYCrCrCr | CbCbCbCbCbCb<br>YYYYYY<br>CrCrCrCrCrCr |
| 420 YCrCb,<br>with CbYCrY packing | CbYCrYYYCb<br>YCrYYYCbY<br>CrYYY | CbCbCbYYYYYYYYYYYYYCrCrCr<br>CbCbCbYYYYYYYYYYYYYCrCrCr | CbCbCbCbCbCb<br>YYYYYYYYYYYY<br>YYYYYYYYYYYY<br>CrCrCrCrCrCr |

## Image Sample Rate

DM_IMAGE_RATE is the native display rate in frames per second of a movie file.

## Digital Audio Essentials

This section describes audio file formats and digital audio data attributes.

### Digital Audio Basics

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 Nyquist Theorem provides a way of determining the minimum sampling frequency required to accurately represent the information (in a given bandwidth) contained in an analog signal. Typically, digital audio information is sampled at a frequency that is at least double the highest interesting analog audio frequency. See *The Art of Digital Audio* or a similar reference on digital audio for more information.

### Digital Audio Attributes and Parameters

Parameters in *dmedia/dm_audio.h* provide a common language for describing digital audio attributes for the digital media libraries.

Digital audio has the following attributes:

- audio channels
- audio compression scheme
- audio sample format (e.g., twos-complement binary, floating point)
- audio sample rate
- audio sample width (number of bits per sample)
- PCM mapping

### Audio Channels

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. 4-channel samples are also interleaved, but each frame usually has two left-right sample pairs, but there can be other arrangements.

Figure 2-8 shows the relationship between the number of channels and the frame size of audio sample data.

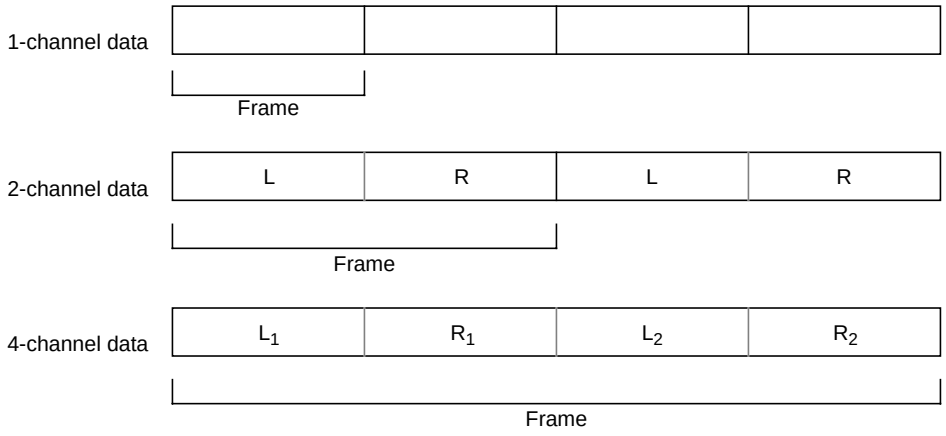



**Figure 2-8** Audio Samples and Frames

## Audio Sample Rate

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 48 kilohertz (kHz) sample rate, 48,000 digital samples are generated for every second of the signal.

To understand how the sample rate relates to sound quality, consider the fact that a telephone transmits voice-quality audio in a frequency range of about 320 Hz to 3.2 kHz. This frequency range can be represented accurately with a sample rate of 6.4 kHz. The range of human hearing, however, extends up to approximately 18–20 kHz, requiring a sample rate of at least 40 kHz.

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. The most common sample rates used for DATs are 44.1 kHz and 48 kHz. Higher sample rates result in higher-quality digital signals; however, the higher the sample rate, the greater the signal storage requirement.

## Audio Compression Scheme

All audio data on Silicon Graphics systems is considered to have a compression scheme. The scheme may be an industry standard such as MPEG-1 audio, or it may be no compression at all. For more information, see "The Digital Media Audio Conversion Library" in Chapter 6.

## Audio Sample Format

Uncompressed audio data is encoded in a digital data format called linear *pulse code modulation* (PCM) (see the audio references for a definition of this term) to represent digital audio samples.

The formats supported by the audio system are:

- 8-bit and 16-bit signed integer
- 24-bit signed, right-justified within a 32-bit integer
- 32-bit and 64-bit floating point

**Note:** The audio hardware supports 16-bit I/O for analog data and 24-bit I/O for AES/EBU digital data.

For floating point data, the application program specifies the desired range of values for the samples; for example, from –1.0 to 1.0. A method for relating data from one range of values to data with a different range of values is described next.

## PCM Mapping

PCM mapping describes the relationship between data with differing sample ranges. If the input and output mappings are different, a conversion consisting of clipping and transformation of values is necessary.

PCM mapping defines a reference value, denoted by DM_AUDIO_PCM_MAP_INTERCEPT, that is the midway point between a signal swing. It is convenient to assign a value of zero to this point. Adding a slope value, denoted by DM_PCM_MAP_SLOPE, to the intercept obtains the full-scale deflection.

The values DM_AUDIO_PCM_MAP_MINCLIP and DM_AUDIO_PCM_MAXCLIP define the minimum and maximum legal PCM values. Input and output values are clipped to these values. If *maxclip* <= *minclip*, then no clipping is done because all PCM values are legal, even if they are outside the true full-scale range.

To transform a PCM value to a corresponding value in the range +1.0 to -1.0:

## Audio Sample Width

The native data format used by the audio hardware is 24-bit two's complement integers. The audio hardware sign-extends each 24-bit quantity into a 32-bit word before delivering the samples to the Audio Library.

Audio input samples delivered to the Audio Library from the Indigo, Indigo$^2$, and Indy audio hardware have different levels of resolution, depending on the input source that is currently active; the AL provides samples to the application at the desired resolution. You can also write your own conversion routine if desired.

Microphone / line-level input samples come from analog-to-digital (A / D) converters, which have 16-bit resolution. These samples are treated as 24-bit samples with 0's in the low 8 bits.

AES / EBU digital input samples have either 20-bit or 24-bit resolution, depending on the device that is connected to the digital input; for the 20-bit case (the most common), samples are treated as 24-bit samples, with 0's in the least significant 4 bits. The AL passes these samples through to the application if 24-bit two's complement is specified. If two's complement with 8-bit or 16-bit resolution is specified, the AL right-shifts the samples so that they will fit into a smaller word size. For floating point data, the AL converts from the 24-bit format to floating point, using a scale factor specified by the application to map the peak integer values to peak float values.

For audio output, the AL delivers samples to the audio hardware as 24-bit quantities sign-extended to fill 32-bit words. The actual resolution of the samples from a given output port depends on the application program connected to the port. For example, an

application may open a 16-bit output port, in which case the 24-bit samples arriving at the audio processor will contain 0's in their least significant 8 bits.

The Audio Library is responsible for converting between the output sample format specified by an application and the 24-bit native format of the audio hardware. For 8-bit or 16-bit integer samples, this conversion is accomplished by left-shifting each sample written to the output port by 16 bits and 8 bits, respectively. For 32-bit or 64-bit floating point samples, this con version is accomplished by rescaling each sample from the range of floating point values that is specified by the application to the full 24-bit range and then rounding the sample to the nearest integer value.

Table 2-5 lists the audio parameters and the valid values for each (not all values are supported by all libraries).

**Table 2-5** Audio Parameters

| Parameter | Type | Values |
|---|---|---|
| DM_AUDIO_CHANNELS | Integer | 1, 2, or 4 |
| DM_AUDIO_COMPRESSION | String | DM_AUDIO_UNCOMPRESSED (default)<br>DM_AUDIO_G711_U_LAW<br>DM_AUDIO_G711_A_LAW<br>DM_AUDIO_MPEG<br>DM_AUDIO_MPEG1<br>DM_AUDIO_MULTIRATE<br>DM_AUDIO_G722<br>DM_AUDIO_G726<br>DM_AUDIO_G728<br>DM_AUDIO_DVI<br>DM_AUDIO_GSM |
| DM_AUDIO_FORMAT | DMaudioformat | DM_AUDIO_TWOS_COMPLEMENT (default)<br>DM_AUDIO_UNSIGNED<br>DM_AUDIO_FLOAT<br>DM_AUDIO_DOUBLE |
| DM_AUDIO_RATE | Double | Native rates are 8000, 11025, 16000, 22050, 32000, 44100, and 48000 Hz |
| DM_AUDIO_WIDTH | Integer | 8, 16, or 24 |

# Digital Media Synchronization Essentials

Most digital media applications use more than one medium in conjunction, for example, audio and video. This section explains how the data can be related to each other for the various digital media functions that perform capture and presentation of concurrent media streams.

## Timecodes

Timecodes are important for synchronizing and editing audio and video data.

There are different types of encoding methods, and standards. In general, a timecode refers to a number represented as *hours:minutes:seconds:frames*. This numerical representation is used in a variety of ways (in both protocols and user interfaces), but in all cases the numbering scheme is the same. The SMPTE 12M standard provides definitions and specifications for a variety of timecodes and timecode signal formats.

The numerical ranges for each field in a timecode are as follows:

| | |
|---|---|
| hours | 00 to 23 |
| minutes | 00 to 59 |
| seconds | 00 to 59 |
| frames | depends on the signal type |

Some signals use a drop-frame timecode, where some "hours:minutes:seconds:frame" combinations are not used; they are simply skipped in a normal progression of timecodes.

A timecode can refer to a

- timer
- timestamp
- signal on wire
- signal on tape

One example application where timecode is used merely as a way to express a time is *mediaplayer*, which displays the offset from the beginning of the movie either in seconds or as a timecode.

Another common computer application of timecode is as a timestamp on particular frames in a movie file. The Silicon Graphics movie file format and the QuickTime format offer the ability to associate each image in the file with a timecode. Sometimes these timecodes are synthesized by the computer, and sometimes they were captured along with the source material. These timecodes are often used as markers so that edited or processed material can be later correlated with material edited or processed on another machine. A/V professionals use an edit decision list (EDL) to indicate the timecodes frames to be recorded.

### Longitudinal Time Code

Longitudinal time code (LTC), sometimes ambiguously referred to as SMPTE time code, is a self-clocking signal defined separately for 525- and 625-line video signals, where the corresponding video signal itself is carried on another wire.The signal occupies its own channel and resembles an audio signal in voltage and bandwidth.

LTC is the most common way of slaving one machine's transport to that of another machine (by ensuring that both machines are on the same frame, not by genlocking signals on both machines). In some audio and MIDI applications, LTC is useful even though there is no video signal.

In a LTC signal, there is one codeword for each video frame. Each LTC codeword contains a timecode and other useful information.

See dmLTC(3dm) for routines for decoding LTC.

### Vertical Interval Time Code

Vertical interval time code (VITC) is a standardized part of a 525- or 625-line video signal. The code itself occupies some lines in the vertical blanking interval of each field of the video signal (not normally visible on monitors). It's a good idea to provide data redundancy by recording the VITC on 2 non-consecutive lines in case of video dropout.

Each VITC codeword contains a timecode, and a group of flag bits that include

- Dropframe
- Colorframe
- Parity
- Field Mark

The field mark bit is an F1/F2 field indicator; it is asserted for a specific field.

VITC also provides 32 user bits, where users can store information such as reel and shot number. This information can be used to help index footage after it is shot, and under the right circumstances (not always trivial), the original VITC recorded along with footage can even tag along with that footage as it is edited, allowing one to produce an edit list or track assets, given a final prototype edit.

See dmVITC(3dm) for routines for decoding VITC.

#### MIDI Time Code

MIDI time code is part of the standard MIDI protocol, which is carried over a serial protocol that is also called MIDI. Production studios often need to synchronize the transports of computers with the transports of multitrack audio tape recorders and dedicated MIDI sequencers. Sometimes LTC is used for this, and sometimes the MIDI time code is the clock signal of choice.

#### Time Code in AES Digital Audio Streams

The AES standard allows embedded timecodes in a digital audio signal.

### Unadjusted System Time and Media Stream Count

The Digital media libraries provide their own temporal reference, called unadjusted system time (UST). The UST is an unsigned 64-bit number that measures the number of nanoseconds since the system was booted. UST values are guaranteed to be monotonically increasing and are readily available for all the Digital Media Libraries.

Typically, the UST is used as a timestamp, that is, it is paired with a specific item or location in a digital media stream. Because each type of media, and similarly each of the libraries, possess unique attributes, the UST information is presented in a different

manner in each library. Table 2-6 describes how UST information is provided by each of the libraries.

**Table 2-6** Methods for Obtaining Unadjusted System Time

| Library | UST Method |
|---|---|
| Digital Media Library | **dmGetUST()** and **dmGetUSTCurrentTimePair()** |
| Audio Library | **ALgetframenumber()** and **ALgetframetime()** |
| MIDI Library | **mdTell()** and **mdSetTimestampMode()** |
| Video Library | *ustime* field in the DMediaInfo structure |

## Synchronization and UST/MSC

The media stream count (MSC), in conjunction with the UST, is used to synchronize buffered media data streams. UST / MSC pairs are used with libraries, such as the Audio Library and the Video Library, that provide timing information about the sampled data. The MSC is a monotonically increasing, unsigned 64-bit number that is applied to each sample in a media stream. This means the MSC of the most recent data sample has the largest value. By using the UST / MSC facility, an application can schedule accurately the use of data samples, and also can detect data underflow and overflow. To see how these things are done, we must lay some groundwork.

A media stream can be seen as travelling a *path*. An input path comprises electrical signals on an input *jack* (an electrical connection) being converted by a *device* to digital data that is placed in an input buffer for use by an application. An output path goes from the application to an output jack via an output buffer. As implied by this description, the data placed in a buffer by the device (input path) or application (output path) has the highest MSC and the data taken out by the application or device respectively has the lowest.

There are two kinds of MSCs, device and frontier. The *device* MSC is the basis for the other. An input device assigns a device MSC to a sample about to be placed in the input buffer. An output device assigns one to a sample about to be removed from the output buffer. The MSC of the sample at the application's end of a buffer is the *frontier* MSC. It is calculated based on the device MSC. In an input path, the frontier MSC is equal to the device MSC minus the number of samples waiting in the input buffer. In an output path, the frontier MSC equals the device MSC plus the number of waiting output buffer samples.

What does using MSCs enable your application to do? Assuming the data stream going to the buffer is not underflowing or overflowing, your application can precisely control the sample flow by using MSCs to determine corresponding USTs. Your application can synchronize data streams, such an audio stream and a video stream, by matching the USTs of their samples. Also, it can compensate for IRIX™ scheduling interruptions by using the USTs of the samples to controlling the contents of the buffer.

As shown below by the Video Library code sample, you can determine the time (UST) a media stream sample came in from or went out to a jack by using the functions **vlGetFrontierMSC()** and **vlGetUSTMSCPair()**.

```
double      ust_per_msc;
USTMSCpair  pair;
stamp_t     frontier_msc, desired_ust;
int         err;

ust_per_msc = vlGetUSTPerMSC(server, path);
err = vlGetUSTMSCPair(server, path, video_node, &pair);
frontier_msc = vlGetFrontierMSC(server, path, memNode);
desired_ust = pair.ust + ((frontier_msc - pair.msc) * ust_per_msc);
```

This sample works for both input and output paths. In either case, the sample indicated by *desired_ust* is the one with the frontier MSC. Thus for an input path, *desired_ust* is the UST of the next sample to be taken from the buffer by your application. For an output path, it is the UST of the next sample your application will place in the buffer. The USTs of other samples in the buffer can be found by adjusting the calculation on the last line.

The above techniques assume that there is no data underflow or overflow to the buffer. If there is an underflow or overflow condition, calculations like the above become unreliable. This is because the frontier MSC is based on the current device MSC. It is not a constant value attached to a specific data sample. Let's use an overflow condition in an input path as an example. The device MSC, and thus the frontier MSC, is increased by one every time the device is ready to place a sample in the input buffer. Because the buffer is full, the sample is discarded, but the MSCs retain their new values. Therefore, the UST / MSC pair associated with a given sample has changed and calculations like the one in the earlier code sample are no longer reliable.

This situation also demonstrates one of the advantages of the UST / MSC pairing. The design enables your application to determine buffer overflow or underflow immediately, based on the value of the frontier MSC. In the above example, your application can check for data overflow immediately after putting data samples into the buffer by checking if the difference between the current frontier MSC and the previous frontier MSC is greater

than the number of samples just enqueued. If it is greater there is an overflow condition. The size of the discrepancy is the actual magnitude of the overflow because putting the samples into the buffer relieved the overflow. Your application can make analogous determinations for input underflow, and output overflow and underflow. Notice that the overflow condition can be found without waiting for the samples with the discontinuous data to get to the front of the buffer. This allows your application to take corrective action immediately.

**Table 2-7** Methods for Using UST / MSC

| Function | Description |
|---|---|
| **vlGetFrontierMSC()** **ALgetframenumber()** | Get the frontier MSC associated with a particular node. See also vlGetFrontierMSC(3dm) and ALgetframenumber(3dm). |
| **vlGetUSTMSCPair()** **ALgetframetime()** | Get the time at which a field or frame came in or will go out. See also vlGetUSTMSCPair(3dm) and ALgetframetime(3dm). |
| **vlGetUSTPerMSC()** | Get the time interval between fields or frames in a path. See also vlGetUSTPerMSC(3dm). |

## Counting Video Fields with MSCs

The VL presents field numbers to a VL application in two contexts:

- For video-to-memory or memory-to-video paths whose VL_CAP_TYPE is set to VL_CAPTURE_NONINTERLEAVED (fields separate, each in its own buffer), the functions **vlGetFrontierMSC()** and **vlGetUSTMSCPair()** return MSCs which count fields (in other VL_CAP_TYPEs, the returned MSCs do not count fields).
- For any video-to-memory path, the user can use **vlGetDMediaInfo()** to return the **DMediaInfo** structure contained in an entry in a **VLBuffe**r. This structure contains a member called sequence which always counts fields (regardless of VL_CAP_TYPE).

In both of these cases, there should be the following correlation:

- these values should be 0%2 if they represent an F1 field
- these values should be 1%2 if they represent an F2 field

This is a relatively new convention and is not yet implemented on all devices.

# Digital Media File Format Essentials

## Image Containers

- RGB
- FIT
- GIF
- JFIF
- PNG
- PPM
- TIFF
- Photo CD

## Audio Containers

Currently, the Digital Media Libraries support the following audio file formats:

- Raw audio data
- AIFF / AIFC
- WAVE
- NeXT .snd
- Sun .au
- Berkeley IRCAM / CARL (BICSF)
- Digidesign Sound Designer II
- Raw MPEG1 audio bitstream
- AVR
- IFF 8SVX
- VOC
- Samplevision
- Soundfont2

In addition, the Digital Media Libraries recognize but do not support

- Sound Designer I
- NIST Sphere

## Movie Containers

A movie is a collection of digital media data contained in tracks, temporally organized in a storage medium, which is captured from and played to audio and video devices. Saying that movies are composed of time-based data means that each piece of data is associated (and usually timestamped) with a particular instant in time, and has a certain duration in time. Movies can contain multiple tracks of different media.

Movies are generally stored in a file format that contains both a descriptive header and the movie data. When a movie is opened, only the header information exists in memory. A movie also has properties or attributes which are independent of the file format and may not necessarily be stored in a file. This section describes movie file formats and attributes.

Parameters in *dmedia/dm_image.h* and *dmedia/dm_audio.h* provide a common language for specifying movie data attributes. The Movie Library also provides its own parameters in *libmovie/movifile.h*.

The Movie Library currently supports these file formats:

- QuickTime
- MPEG-1 systems and video bitstreams
- Silicon Graphics movie format

*Chapter 3*

# Digital Media Data Types and Parameter Lists

This chapter explains how to use digital media data structures that facilitate data specification and setting, getting, and passing parameters.

## Digital Media Data Type Definitions

The DM Library provides type definitions for digital media that are useful when programming with the family of Digital Media Libraries. Data types and constant names have an uppercase DM prefix; routines have a lowercase dm prefix.

The *dmedia/dmedia.h* header file provides these type definitions:

| | |
|---|---|
| DMboolean | integer for conditionals; DM_FALSE is 0 and DM_TRUE is 1 |
| DMfraction | integer numerator divided by integer denominator |
| DMstatus | enumerated type consisting of DM_SUCCESS and DM_FAILURE |

It is good programming practice to check the return values of functions. DMstatus provides a way to check return values. When a function succeeds, DM_SUCCESS is returned. When a function fails, DM_FAILURE is returned and a system error code is set that can be interpreted using the functions described in the next section.

### Digital Media Error Handling

Errors encountered while using the Digital Media Libraries can be diagnosed with the help of two routines. The function **dmGetError()** retrieves the number, summary, and detailed description of an error generated by the execution of the current process. The current process in this case is the same as determined by **getpid()**. The companion function, **dmGetErrorForPID()**, gets the same type of error information for a process your application specifies.

```
const char *dmGetError ( int *errornum,
                         char error_detail[DM_MAX_ERROR_DETAIL] )
const char *dmGetErrorForPID ( pid_t pid, int *errornum,
                               char error_detail[DM_MAX_ERROR_DETAIL] )
```

The functions **dmGetError()** and **dmGetErrorForPID()** enable your application to handle in a consistent manner errors generated while using the digital media libraries. Both functions return a pointer to a null-terminated character string that summarizes the error. The setting of the errors by the libraries and the retrieval of them by your application is guaranteed to be thread-safe. Only the most recent error for a given thread is returned. If there are no errors, the functions return NULL.

The parameter *pid* in **dmGetErrorForPID()** is the id of the process in which to check for an error. The last parameter in both functions, *error_detail*, is the address of a null-terminated character array of size DM_MAX_ERROR_DETAIL. If one exists, a detailed description of the error is loaded into the array. If you set *error_detail* to NULL, no description is loaded. The remaining parameter, *errornum*, is a pointer to an integer into which the number of the current error is loaded. If your application sets *errornum* to NULL, no number is loaded. The error numbers returned in *errornum* fall into ranges according to the digital media libraries that generated them. The currently defined error ranges and their libraries are as follows:

| | |
|---|---|
| 0-999 | UNIX® System (The error numbers are identical to those returned by oserror(3C).) |
| 1000-1999 | Color Space Library in *libdmedia* |
| 2000-2999 | Movie Library in *libmoviefile* or *libmovieplay* |
| 3000-3999 | Audio File Library in *libaudiofile* |
| 4000-4999 | DMbuffer in *libdmedia* |
| 5000-5999 | Audio Converter in *libdmedia* |
| 6000-6999 | Image Converter in *libdmedia* |
| 10000-10999 | Global Digital Media Library in *libdmedia* |
| 11000-11999 | FX Plug-in Utility Library in *libfxplugutils* |
| 12000-12999 | FX Plug-in Manager Library in *libfxplugmgr* |

## Digital Media Parameter Types

The DM Library provides definitions for the digital media parameter data types. Table 3-1 lists the digital media parameter type definitions that are defined in *dmedia/dm_params.h*.

**Table 3-1** Digital Media Parameter Data Types

| Parameter Type | Meaning |
|---|---|
| DM_TYPE_BINARY | Binary data |
| DM_TYPE_ENUM | Enumerated type |
| DM_TYPE_ENUM_ARRAY | Array of enumerated types |
| DM_TYPE_FLOAT | Floating point value (double) |
| DM_TYPE_FLOAT_ARRAY | Array of floats |
| DM_TYPE_FLOAT_RANGE | Range of floats |
| DM_TYPE_FRACTION | Ratio |
| DM_TYPE_FRACTION_ARRAY | Array of fractions |
| DM_TYPE_FRACTION_RANGE | Range of fractions |
| DM_TYPE_INT | Integer value |
| DM_TYPE_INT_ARRAY | Array of integers |
| DM_TYPE_INT_RANGE | Range of integers |
| DM_TYPE_LONG_LONG | Long long (64-bits) |
| DM_TYPE_PARAMS | DMparams list |
| DM_TYPE_STRING | String |
| DM_TYPE_STRING_ARRAY | Array of strings |
| DM_TYPE_TOC_ENTRY | Table-of-contents entry for ring buffers |

## Digital Media Parameter Lists

Parameter-value lists, which are contained in a DMparams structure supply configuration information for digital media objects such as audio ports, movie tracks, and video devices. A DMparams list is a list of pairs, where each pair contains the name of a parameter and the corresponding value for that parameter.

You can use a DMparams list to

- configure a digital media structure upon initialization by passing a complete list containing all the parameters and values needed to configure that object to a creation routine
- change the settings of an existing digital media structure by providing a list of parameters and corresponding values to replace

Most Digital Media Libraries provide convenience routines for setting, adjusting, and getting relevant parameter values.

Every DMparams list that describes a format includes the parameter DM_MEDIUM to indicate what kind of data it describes. DM_MEDIUM is an enumerated type consisting of:

| | |
|---|---|
| DM_IMAGE | which represents image data |
| DM_AUDIO | which represents audio data |
| DM_TIMECODE | which represents a timecode |
| DM_TEXT | which represents text |

Another common parameter, DM_CODEC, is an enumerated type that describes whether a codec is synchronous (DM_SYNC_CODEC) or asynchronous (DM_ASYNC_CODEC). The compressor and decompressor of a *synchronous codec* are linked such that there must be both uncompressed input available to the compressor and compressed input available to the decompressor before either can generate output. An *asynchronous codec* has no such linkage.

This section explains how to use the DM Library routines for

- creating and destroying DMparams lists
- creating default audio and image configurations
- setting and getting values in DMparams lists

- manipulating DMparams lists

The routines described in this section follow the general rule that ownership of data is not passed during procedure calls, except in the routines that create and destroy DMparams lists. Functions that take strings copy the strings if they want to keep them. Functions that return strings or other structures retain ownership and the caller must not free them.

In the initialization section of your application, you create and use DMparams lists to configure data structures for your application as described in the following steps:

1. Create an empty DMparams list by calling **dmParamsCreate()**.
2. Set the parameter values by one of the methods listed below:
    - Use a function that sets up a standard configuration for a particular type of data: **dmSetImageDefaults()** for images, **dmSetAudioDefaults()** for audio.
    - Use a generic function such as **dmParamsSetInt()** to set the values of individual parameters within an empty DMparams list or one that has already been initialized with the standard audio or image configuration. See “Setting and Getting Individual Parameter Values” on page 65 for a description of this method.
    - Use a library function such as **mvSetMovieDefaults()** to set a group of parameters specific to that library.
3. Free the DMparams list and its contents by calling **dmParamsDestroy()**.

These steps are described in detail in the sections that follow.

## Creating and Destroying DMparams Lists

Some libraries require you to allocate memory for DMparams lists, but with the DM library, you need not allocate memory for DMparams lists, because memory management is provided for you by the **dmParamsCreate()** and **dmParamsDestroy()** routines. These routines work together as a self-contained block within which you create the DMparams list, set the parameter value(s) and use them, and then destroy the structure, freeing its associated memory.

Only the **dmParamsCreate()** function can create a DMparams list, and only the **dmParamsDestroy()** function can free one. This means that DMparams lists are managed correctly when every call to create one is balanced by a call to destroy one. The creation

function can fail because of lack of memory, so it returns an error code. The destructor can never fail.

To create an empty DMparams list, call **dmParamsCreate()**. Its function prototype is:

```
DMstatus dmParamsCreate ( DMparams** returnNewList )
```

where:

*returnNewList* is a pointer to a handle that is returned by the DM Library

If there is sufficient memory to allocate the structure, a pointer to the newly created structure is put into **returnNewList* and DM_SUCCESS is returned; otherwise, DM_FAILURE is returned.

When you have finished using the DMparams list, you must destroy it to free the associated memory. To free both the DMparams list structure and its contents, call **dmParamsDestroy()**. Its function prototype is:

```
void dmParamsDestroy ( DMparams* params )
```

where:

*params* is a pointer to the DMparams list you want to destroy

Example 3-1 is a code fragment that creates a DMparams list called *params*, then calls a Movie Library routine, **mvSetMovieDefaults()**, to initialize the default movie parameters, and finally destroys the list, freeing both the structure and its contents.

**Example 3-1** Creating and Destroying a DMparams List

```
DMparams* params;
if ( dmParamsCreate( &params ) != DM_SUCCESS ) {
    printf( "s\n", dmGetError(NULL, NULL) );
    exit( 1 );
}
if ( mvSetMovieDefaults(params, MV_FORMAT_SGI_3) != DM_SUCCESS ) {
    printf( “s\n”, mvGetErrorStr(mvGetErrno()));
    exit( 1 );
}
dmParamsDestroy ( params );
```

## Setting and Getting Individual Parameter Values

After creating an empty DMparams list or a default audio or image configuration, you can use the routines described in this section to set and get values for individual elements of a DMparams list.

There is a routine for setting and getting the parameter values for each parameter data type defined in the DM Library, as listed in Table 3-1.

All of these functions store and retrieve entries in a DMparams list. They assume that the named parameter is present and is of the specified type; the debugging version of the library asserts that this is the case. All functions that can possibly fail return an error code indicating success or failure. Insufficient memory is the only reason these routines can fail. Type mismatch causes a failed assertion in the debug library and undefined results in the non-debug library.

Table 3-2 lists the DM Library routines for setting parameter values. All the routines except **dmParamsSetBinary()** require three arguments:

| | |
|---|---|
| *params* | a pointer to a DMparams list |
| *paramName* | the name of the parameter whose value you want to set |
| *value* | a value of the appropriate type for the given parameter |

**Table 3-2** DM Library Routines for Setting Parameter Values

| Routine | Purpose |
|---|---|
| **dmParamsSetBinary()** | Sets the contents of a data buffer. See dmParamsSetInt(3dm). |
| **dmParamsSetEnum()** | Sets the value of an enum parameter whose type is int. |
| **dmParamsSetEnumArray()** | Sets the value of a parameter whose type is DMenumarray. |
| **dmParamsSetFloat()** | Sets the value of a parameter whose type is double. |
| **dmParamsSetFloatArray()** | Sets the value of a parameter whose type is DMfloatarray. |
| **dmParamsSetFloatRange()** | Sets the value of a parameter whose type is DMfloatrange. |
| **dmParamsSetFract()** | Sets the value of a parameter whose type is DMfraction. |
| **dmParamsSetFractArray()** | Sets the value of a parameter whose type is DMfractionarray. |
| **dmParamsSetFractRange()** | Sets the value of a parameter whose type is DMfractionrange. |

**Table 3-2 (continued)** DM Library Routines for Setting Parameter Values

| Routine | Purpose |
|---|---|
| **dmParamsSetInt()** | Sets the value of a parameter whose type is int. |
| **dmParamsSetIntArray()** | Sets the value of a parameter whose type is DMintarray. |
| **dmParamsSetIntRange()** | Sets the value of a parameter whose type is DMintrange. |
| **dmParamsSetLongLong()** | Sets the value of a parameter whose type is long long. |
| **dmParamsSetParams()** | Sets the value of a parameter whose type is DMparam. |
| **dmParamsSetString()** | Sets the value of a parameter whose type is a character string. |
| **dmParamsSetStringArray()** | Sets the value of a parameter whose type is DMstringarray. |

These routines return either DM_SUCCESS or DM_FAILURE.

Table 3-3 lists the DM Library routines for getting parameter values. All the routines except **dmParamsGetBinary()** require two arguments:

*params* a pointer to a DMparams list

*paramName* the name of the parameter whose value you want to get

Routines that get values return either a pointer to a value or the value itself. For strings, DMparams lists, and table-of-contents entries, the pointer that is returned points into the internal data structure of the DMparams list. This pointer should never be freed and is only guaranteed to remain valid until the next time the list is changed. In general, if you need to keep a string value around after getting it from a DMparams list, it should be copied.

**Table 3-3** DM Library Routines for Getting Parameter Values

| Routine | Purpose |
|---|---|
| **dmParamsGetBinary()** | Returns a pointer to binary data. See dmParamsSetInt(3dm). |
| **dmParamsGetEnum()** | Returns an integer value for the given enum parameter. |
| **dmParamsGetEnumArray()** | Returns a pointer to a value of type DMenumarray. |
| **dmParamsGetFloat()** | Returns a value of type double for the given parameter. |
| **dmParamsGetFloatArray()** | Returns a pointer to a value of type DMfloatarray. |

**Table 3-3 (continued)** DM Library Routines for Getting Parameter Values

| Routine | Purpose |
|---|---|
| **dmParamsGetFloatRange()** | Returns a pointer to a value of type DMfloatrange. |
| **dmParamsGetFract()** | Returns a value of type DMfraction for the given parameter. |
| **dmParamsGetFractArray()** | Returns a pointer to a value of type DMfractionarray. |
| **dmParamsGetFractRange()** | Returns a pointer to a value of type DMfractionrange. |
| **dmParamsGetInt()** | Returns an integer value for the given parameter. |
| **dmParamsGetIntArray()** | Returns a pointer to a value of type DMintarray for the parameter. |
| **dmParamsGetIntRange()** | Returns a pointer to a value of type DMintrange for the parameter. |
| **dmParamsGetLongLong()** | Returns a 64-bit long for the given parameter. |
| **dmParamsGetParams()** | Returns a pointer to a value of type DMparams for the parameter. |
| **dmParamsGetString()** | Returns a pointer to a value of type const char for the parameter. |
| **dmParamsGetStringArray()** | Returns a pointer to a value of type DMstringarray. |

## Setting Parameter Defaults

### Setting Image Defaults

To initialize a DMparams list with the default image configuration, call **dmSetImageDefaults()**, passing in the width and height of the image frame, and the image packing format. Its function prototype is:

```
DMstatus dmSetImageDefaults ( DMparams* params, int width,
                              int height, DMpacking packing )
```

where:

*params* is a pointer to a DMparams list that was returned by **dmParamsCreate()**

*width* is the width of the image in pixels

*height* is the height of the image in pixels

*packing* is the image packing format

Table 3-4 lists the parameters and values set by **dmSetImageDefaults()**.

**Table 3-4** Image Defaults

| Parameter | Default |
|---|---|
| DM_MEDIUM | DM_IMAGE |
| DM_IMAGE_WIDTH | *width* |
| DM_IMAGE_HEIGHT | *height* |
| DM_IMAGE_RATE | 15.0 frames per second (Hz) |
| DM_IMAGE_INTERLACING | DM_IMAGE_NONINTERLACED |
| DM_IMAGE_PACKING | *packing* |
| DM_IMAGE_ORIENTATION | DM_BOTTOM_TO_TOP |
| DM_IMAGE_COMPRESSION | DM_IMAGE_UNCOMPRESSED |

### Determining the Buffer Size Needed to Store an Image Frame

To determine the image frame size for a given DMparams list, call **dmImageFrameSize()**. **dmImageFrameSize()** returns the number of bytes needed to store one uncompressed image frame in the given format. Its function prototype is:

```
size_t dmImageFrameSize ( const DMparams* params )
```

Example 3-2 is a code fragment that creates a DMparams list, fills in the image defaults, and then frees the structure and its contents.

**Example 3-2** Setting Image Defaults

```
DMparams* imageParams;

if ( dmParamsCreate( &imageParams ) != DM_SUCCESS ) {
    printf( "s\n", dmGetError(NULL, NULL) );
    exit( 1 );
}
if ( dmSetImageDefaults( imageParams,
       320,  /* width */
       240,  /* height */
       DM_PACKING_RGBX ) != DM_SUCCESS ) {
    printf( "s\n", dmGetError(NULL, NULL) );
    exit( 1 );
}
printf( "%d bytes per image frame.\n",
        dmImageFrameSize( imageParams ) );
dmParamsDestroy( imageParams );
```

### Setting Audio Defaults

To initialize a DMparams list with the default audio configuration, call **dmSetAudioDefaults()**, passing in the desired sample width, sample rate, and number of channels. Its function prototype is:

```
DMstatus dmSetAudioDefaults ( DMparams* params, int width,
                              double rate, int channels )
```

where:

*params* is a pointer to a DMparams list that was returned from **dmParamsCreate()**

*width* is the number of bits per audio sample: 8, 16, or 24

*rate* is the audio sample rate; the native audio sample rates are 8000, 11025, 16000, 22050, 32000, 44100, and 48000 Hz

*channels* is the number of audio channels

**dmSetAudioDefaults()** returns DM_SUCCESS if there was enough memory available to set up the parameters; otherwise, it returns DM_FAILURE.

Table 3-5 lists the parameters and values set by **dmSetAudioDefaults()**.

**Table 3-5** Audio Defaults

| Parameter | Default |
|---|---|
| DM_MEDIUM | DM_AUDIO |
| DM_AUDIO_WIDTH | *width* |
| DM_AUDIO_FORMAT | DM_AUDIO_TWOS_COMPLEMENT |
| DM_AUDIO_RATE | *rate* |
| DM_AUDIO_CHANNELS | *channels* |
| DM_AUDIO_COMPRESSION | DM_AUDIO_UNCOMPRESSED |

### Determining the Buffer Size Needed to Store an Audio Frame

To determine the audio frame size for a given DMparams list, call **dmAudioFrameSize()**. **dmAudioFrameSize()** returns the number of bytes needed to store one audio frame (one sample from each channel). Its function prototype is:

```
size_t dmAudioFrameSize ( DMparams* params )
```

Example 3-3 is a code fragment that creates a DMparams list, fills in the audio defaults, and then frees the structure and its contents.

**Example 3-3** Setting Audio Defaults

```
DMparams* audioParams;
if ( dmParamsCreate( &audioParams ) != DM_SUCCESS ) {
    printf( "s\n", dmGetError(NULL, NULL) );
    exit( 1 );
}
if ( dmSetAudioDefaults ( audioParams,
                          16,    /* width (in bits/sample) */
                          22050, /* sampling rate */
                          2      /* # channels (stereo) */
                          ) != DM_SUCCESS ) {
    printf( "s/n", dmGetError(NULL, NULL) );
    exit( 1 );
}
printf( "%d bytes per audio frame.\n",
         dmAudioFrameSize( audioParams ) );
dmParamsDestroy( audioParams );
```

Example 3-4 shows two equivalent ways of setting up a complete image format description; the first sets the parameter values individually, the second creates a default image configuration with the appropriate values.

**Example 3-4** Setting Individual Parameter Values

```
DMparams* format;
dmParamsCreate( &format );
dmParamsSetInt ( format, DM_IMAGE_WIDTH, 320 );
dmParamsSetInt ( format, DM_IMAGE_HEIGHT, 240 );
dmParamsSetFloat ( format, DM_IMAGE_RATE, 15.0 );
dmParamsSetString( format, DM_IMAGE_COMPRESSION, DM_IMAGE_UNCOMPRESSED );
dmParamsSetEnum( format, DM_IMAGE_INTERLACING, DM_IMAGE_NONINTERLEAVED );
dmParamsSetEnum ( format, DM_IMAGE_PACKING, DM_PACKING_RGBX );
dmParamsSetEnum ( format, DM_IMAGE_ORIENTATION, DM_BOTTOM_TO_TOP
);dmParamsDestroy ( format );
```

The following is equivalent:

```
DMparams* format;
dmParamsCreate ( &format );
dmSetImageDefaults ( format, 320, 240, DM_PACKING_RGBX );
dmParamsDestroy ( format );
```

## Manipulating DMparams Lists

This section explains how to manipulate DMparams lists. Some of the tasks you can two with DMparams lists include:

- testing two parameter values for equality
- copying either individual parameter-value pairs or entire DMparams lists
- determine how many parameter-value pairs are in a particular DMparams list
- getting information about parameter names, data types,

Table 3-6 lists the routines that perform operations on DMparams lists and the entries within them.

**Table 3-6** Routines for Manipulating DMparams Lists and Entries

| Routine | Purpose |
|---|---|
| dmParamsAreEqual() | Determine if the values of two parameters are equal |
| dmParamsCopyAllElems() | Copy the entire contents of one list to another |
| dmParamsCopyElem() | Copy one parameter-value pair from one DMparams list to another |
| dmParamsGetElem() | Get the name of a given parameter |
| dmParamsGetElemType() | Get the data type of a given parameter |
| dmParamsGetNumElems() | Get the number of parameters in a list |
| dmParamsGetType() | Get the data type of the named parameter |
| dmParamsIsPresent() | Determine if a given parameter is in the list |
| dmParamsRemoveElem() | Remove a given parameter from the list |
| dmParamsScan() | Scan all the entries of a digital media parameter list |

The sections that follow explain how to use each routine.

### Determining DMparams Equivalence

The function **dmParamsAreEqual()** compares two DMparams structures and tests for equality. Its function prototype is:

```
DMboolean dmParamsAreEqual ( const DMparams *params1,
                             const DMparams *params2 )
```

If *params1* and *params2* have the same number of parameter-value pairs, and if the parameters of the same name have the same type and value in both lists, then the function returns DM_TRUE.

### Determining the Number of Elements in a DMparams List

To perform any task that requires your application to loop through the contents of a DMparams list (for example, to print out a list of parameters and their values) you need to know how many parameters are in the list in order to set up a loop to step through the entries one-by-one.

To get the total number of elements present in a DMparams list, call **dmParamsGetNumElems()**. Its function prototype is:

```
int dmParamsGetNumElems ( const DMparams* params )
```

The number of elements and their position in a list is guaranteed to remain stable unless the list is changed by using one of the “set” functions, by copying an element into it, or by removing an element from it.

There is also a convenience function, **dmParamsScan()**, for looping through the contents of a DMparams list and performing the same operation on each element of the list. See for more information on

### Copying the Contents of One DMparams List into Another

To copy the entire contents of the *fromParams* list into the *toParams* list, call **dmParamsCopyAllElems()**. Its function prototype is:

```
DMstatus dmParamsCopyAllElems ( const DMparams* fromParams,
                                DMparams* toParams )
```

If there are any parameters of the same name in both lists, the corresponding value(s) in the destination list are overwritten. DM_SUCCESS is returned if there is enough memory to hold the copied data; otherwise, DM_FAILURE is returned. Type mismatch causes a failed assertion in the debug version of the library.

### Copying an Individual Parameter Value from One List into Another

If a parameter appears in more than one DMparams list, it is sometimes more convenient to copy the individual parameter or group of parameters from one list to another, rather than individually setting the parameter value(s) for each list.

To copy the parameter-value pair for the parameter named *paramName* from the *fromParams* list into the *toParams* list, call **dmParamsCopyElem()**. Its function prototype is:

```
DMstatus dmParamsCopyElem ( const DMparams* fromParams,
                            const char* paramName,
                            DMparams* toParams )
```

If there is a preexisting parameter with the same name in the destination list, that value is overwritten. DM_SUCCESS is returned if there is enough memory to hold the copied element; otherwise, DM_FAILURE is returned.

### Determining the Name of a Given Parameter

To get the name of the entry occupying the position given by *index* in the *params* list, call **dmParamsGetElem()**. Its function prototype is:

```
const char* dmParamsGetElem ( const DMparams* params, const int index )
```

The *index* must be from 0 to one less than the number of elements in the list.

### Determining the Data Type of a Given Parameter

To get the data type of the value occupying the position given by *index* in the *params* list, call **dmParamsGetElemType()**. Its function prototype is:

```
DMparamtype dmParamsGetElemType ( const DMparams* params,
                                  const int index )
```

Similarly, to get the data type of the parameter given by *name* in the *params* list, call **dmParamsGetType()**. Its function prototype is:

```
DMparamtype dmParamsGetType ( const DMparams* params,
                              const char* paramName )
```

See Table 3-1 for a list of valid return values.

### Determining if a Given Parameter Exists

To determine whether the element named *paramName* exists in the *params* list, call **dmParamsIsPresent()**. Its function prototype is:

```
DMboolean dmParamsIsPresent  ( const DMparams* params,
                                const char* paramName )
```

DM_TRUE is returned if *paramName* is in *params*; otherwise, DM_FALSE is returned.

### Scanning a DMparams List

Instead of creating your own loop to cycle through the contents of a DMparams list, you can use the convenience routine **dmParamsScan()**, which performs a specified operation on each element of the list. Its function prototype is:

```
DMstatus dmParamsScan ( const DMparams* params,
                        DMstatus (*scanFunc) ( const DMparams* params,
                                               const char* paramName,
                                               void* scanArg,
                                               DMboolean* stopScan ),
                        void* scanArg )
```

The function **dmParamsScan()** passes the name of each entry in a DMparams list and *scanArg* as parameters to *scanFunc*. If *scanFunc* sets the value of *stopScan* to DM_TRUE, **dmParamsScan()** stops the DMparams list scan and returns the value returned by *scanFunc*. Otherwise, **dmParamsScan()** processes all elements in the list and returns DM_SUCCESS.

### Removing an Element from a DMparams List

To remove the *paramName* entry from the *params* list, call **dmParamsRemoveElem()**. Its function prototype is:

```
const char* dmParamsRemoveElem ( DMparams* params,
                                 const char* paramName)
```

The element named *paramName* must be present.

Example 3-5 prints the contents of a DMparams list.

**Example 3-5** Printing the Contents of a Digital Media DMparams List

```
void PrintParams( DMparams* params ) {
    int i;
    int numElems = dmParamsGetNumElems( params );

   for ( i = 0;  i < numElems;  i++ ) {
        const char* name = dmParamsGetElem( params, i );
        DMparamtype type = dmParamsGetElemType( params, i );
        printf( "    %20s: ", name );
        switch( type ) {
            case DM_TYPE_ENUM:
                printf( "%d", dmParamsGetEnum( params, name ) );
                break;
            case DM_TYPE_INT:
                printf( "%d", dmParamsGetInt( params, name ) );
                break;
            case DM_TYPE_STRING:
                printf( "%s", dmParamsGetString( params, name ) );
                break;
            case DM_TYPE_FLOAT:
                printf( "%f", dmParamsGetFloat( params, name ) );
                break;
            case DM_TYPE_FRACTION:
                DMfraction f = dmParamsGetFract( params, name );
                printf( "%d/%d", f.numerator, f.denominator );
                break;
            case DM_TYPE_PARAMS:
                printf( "... param list ... " );
                break;
            case DM_TYPE_TOC_ENTRY:
                printf( "... toc entry ..." );
                break;
            default:
                assert( DM_FALSE );
            }
        printf( "\n" );
    }
}
```

## Compiling and Linking a Digital Media Library Application

Applications that call DM Library routines must include the *libdmedia* header files to obtain definitions for the library; however, these files are usually included in the header file of the library you are using.

This code fragment includes all the *libdmedia* header files:

```
#include <dmedia/dmedia.h>
#include <dmedia/dm_audio.h>
#include <dmedia/dm_image.h>
#include <dmedia/dm_params.h>
#include <dmedia/dm_buffer.h>
#include <dmedia/dm_imageconvert.h>
#include <dmedia/dm_audioconvert.h>
```

Link with the DM Library when compiling an application that makes DM Library calls by including **-ldmedia** on the link line. It's likely that you'll be linking with other libraries as well, and because the linking order is usually specific, follow the linking instructions for the library you are using.

## Debugging a Digital Media Library Application

The debugging version of the DM Library checks for library usage violations by setting assertions that state the requirements for a parameter or value.

To debug your DM application, link with the debugging version of the DM Library, *libdmedia.so.1*, by setting your LD_LIBRARY_PATH environment variable to the directory containing the debug library before linking with **-ldmedia**, and then run your program. For example, use **`setenv LD_LIBRARY_PATH /usr/lib/debug`** to set the path.

Your application will abort with an error message if it fails an assertion. The message explains the situation that caused the error and, by implication or by explicit description, suggests a corrective action.

When you have finished debugging your application, you should relink with the nondebugging library, *libdmedia.a*, because the runtime checks imposed by the debugging library cause an undesirable size and performance overhead for a packaged application.

*Chapter 4*

# Digital Media I/O

This chapter explains how to use the digital media library routines that facilitate real-time input and output between live media devices.

## Video I/O Concepts

This section explains basic video I/O concepts.

Programming video I/O involves

- *devices*, for processing video (each including sets of nodes)
- *nodes*, for defining endpoints or internal processing points of a video transport path
- *paths*, for routing video data by connecting nodes
- *ports*, for producing or consuming video data
- *controls*, parameters for modifying the behavior of video nodes and transport paths
- *events*, for monitoring video I/O status
- *buffers*, for sending video data to and receiving video data from host memory; these can be either VLbuffers or DMbuffers.

Each of these topics is discussed in a separate section.

The manner in which video data transfer is accomplished differs slightly depending on the buffering method, but the essential concepts of using paths, nodes, control, and events apply to both methods.

### Devices

There are two types of video devices: external devices that are connected to a video jack on the workstation and VL video devices which are internal video boards and options

for processing video data. The application should perform a query to determine which external video devices are connected and powered on, by calling **vlGetDeviceList()**,

```
int vlGetDeviceList ( VLServer svr, VLDevList *devlist )
```

which fills the supplied VLDevList structure with a list of available devices, including the number of devices available and an array of VLDevice structures describing the available devices. A VLDevice structure contains the index of the device, the device name, the number of nodes available and a list of VLNodeInfo structures describing the nodes available on that device.

To select the desired node, find the entry in the node list for the device name you want in the return argument of **vlGetDeviceList()**, then pass in the corresponding node number to **vlGetNode()**.

## Nodes

A node is an endpoint or internal processing element of the video transport path, such as a video *source* like a camera, a video *drain* (such as to the workstation screen), a *device* (video), or the *blender* in which video sources are combined for output to a drain.

Nodes have three attributes:

- *type*, which specifies the node's function in a path
- *class*, which identifies the type of system resource associated with the node
- *number*, which differentiates among multiple node instances and typically corresponds to the numbering of the video connectors on the video board

Node types are:

VL_SRC the origination point (source) of a video stream

VL_DRN the destination point (drain) to which video is sent

VL_INTERNAL a mid-stream filter such as a blender

VL_DEVICE a special node for device-global controls shared by all paths

**Note:** For VL_DEVICE, set the node class to 0.

Putting a VL_DEVICE node on a path gives that path access to global device controls that can effect all paths on the device.

Node classes are:

| | |
|---|---|
| VL_VIDEO | a hardware video port that connects to a piece of video equipment such as a video tape deck or camera. All video devices have at least one port. The VL_SRC node type signifies an input port; VL_DRN signifies an output port. |
| VL_MEM | a memory buffer used to send or receive video data |
| VL_GFX | a direct connection between a video device and a graphics framebuffer |
| VL_SCREEN | a direct connection between a video device and a graphics display device, but different from VL_GFX because the video data does not interact directly with the graphics framebuffer and cannot be manipulated with graphics routines |
| VL_TEXTURE | an interface to graphics hardware for transferring video data directly to or from texture memory |
| VL_BLENDER | a filter that operates on data flowing from source to drain |
| VL_CSC | an interface to an optional realtime colorspace converter on systems which support it (and which have the option board installed) |
| VL_FB | an internal framebuffer node for freezing video on certain systems |

Additional node classes may be available on certain video options; refer to the documentation that came with your video option for details.

To create a video node, call **vlGetNode().** Its function prototype is

```
VLNode vlGetNode ( VLServer vlSvr, int type, int class, int number )
```

Upon successful completion, **vlGetNode()** returns a VL Node, a handle to a node, which can be used to identify the node for functions that perform an action on a node.

To use the default node for a device, specify its number as VL_ANY:

```
nodehandle = vlGetNode( svr, VL_SRC, VL_VIDEO, VL_ANY );
```

## Paths

A path is a route between video nodes for directing the flow of video data.

Using a path involves

- creating the path
- getting the device ID
- adding nodes (if needed)
- specifying the data transfer characteristics of the path
- setting up the data path

These steps are explained individually in the sections that follow.

### Creating a Video Data Transfer Path

Use **vlCreatePath()** to create the video data transfer path. Its function prototype is

```
VLPath vlCreatePath ( VLServer svr, VLDev dev, VLNode source, VLNode drain )
```

You can create a path using any available node by specifying the generic value VL_ANY for the device. This code fragment creates a path if the device is unknown:

```
if ((path = vlCreatePath(vlSvr, VL_ANY, src, drn)) < 0) {
    vlPerror(_progName);
    exit(1);
}
```

This code fragment creates a path that uses a device specified by parsing a *devlist*:

```
if ((path = vlCreatePath(vlSvr, devlist.devices[devicenum].dev, src,
    drn)) < 0) {
    vlPerror(_progName);
    exit(1);
}
```

**Note:** If the path contains one or more invalid nodes, **vlCreatePath()** returns VLBadNode.

### Getting the Device ID

If you specify VL_ANY as the device when you create the path, use **vlGetDevice()** to discover the device ID selected. Its function prototype is

```
VLDev vlGetDevice ( VLServer vlSvr, VLPath path )
```

For example:

```
devicenum = vlGetDevice(vlSvr, path);
deviceName = devlist.devices[devicenum].name;
printf("Device is: %s/n", deviceName);
```

### Adding Nodes to an Existing Video Path

You can add nodes to an existing path to provide additional processing or I/O capabilities. For this optional step, use **vlAddNode()**. Its function prototype is

```
int vlAddNode ( VLServer vlSvr, VLPath vlPath, VLNodeId node )
```

where

| | |
|---|---|
| *vlSvr* | names the server to which the path is connected |
| *vlPath* | is the path as defined with **vlCreatePath()** |
| *node* | is the node ID |

### Specifying Video Data Transfer Path Characteristics

Path attributes specify usage rules for video controls and data transfers. Even though the names are the same, the intent and function of the usage attributes depend on whether they specify control or stream (data) usage.

Control usage attributes are:

- VL_SHARE, meaning other paths can set controls on this node; this control is the desired setting for other paths, including *vcp*, to work

  **Note:** When using VL_SHARE, pay attention to events. If another user has changed a control, a VLControlChanged event occurs.

- VL_READ_ONLY, meaning controls cannot be set, only read; for example, this control can be used to monitor controls

- VL_LOCK, which prevents other paths from setting controls on this path; controls cannot be used by another path
- VL_DONE_USING, meaning the resources are no longer required; the application releases this set of paths for other applications to acquire

Stream (data) usage attributes are:

- VL_SHARE, meaning transfers can be preempted by other users; paths contend for ownership

  **Note:** When using VL_SHARE, pay attention to events. If another user has taken over the node, a VLStreamPreempted event occurs.
- VL_READ_ONLY, meaning the path cannot perform transfers, but other resources are not locked; set this value to use the path for controls
- VL_LOCK, which prevents other paths that share data transfer resources with this path from transferring (except that 2 paths can share a video source when locked); existing paths that share resources with this path will be preempted
- VL_DONE_USING, meaning the resources are no longer required; the application releases this set of paths for other applications to acquire

### Setting up a Video Transfer Data Path

Once the path has been created and usage attributes assigned, its settings do not go into effect until the path is set up with **vlSetupPaths()**. Its function prototype is

```
int vlSetupPaths ( VLServer vlSvr, VLPathList paths,
                   u_int count, VLUsageType ctrlusage,
                   VLUsageType streamusage )
```

where

| | |
|---|---|
| *vlSvr* | names the server to which the path is connected |
| *paths* | specifies a list of paths you are setting up |
| *count* | specifies the number of paths in the path list |
| *ctrlusage* | specifies usage for path controls |
| *streamusage* | specifies usage for the data |

This example fragment sets up a path with shared controls and a locked stream:

```
if (vlSetupPaths(vlSvr, (VLPathList)&path, 1, VL_SHARE,
    VL_LOCK) < 0)
{
    vlPerror(_progName);
    exit(1);
}
```

**Note:** The Video Library infers the connections on a path if **vlBeginTransfer()** is called and no drain nodes have been connected using **vlSetConnection()** (implicit routing). To specify a path that does not use the default connections, use **vlSetConnection()** (explicit routing).

- For each internal node on the path, all unconnected input ports are connected to the first source node added to the path. Pixel ports are connected to pixel ports and alpha ports are connected to alpha ports.
- For each drain node on the path, all unconnected input ports are connected to the first internal node placed on the path, if there is an internal node, or to the first source node placed on the path. Pixel ports are connected to pixel ports and alpha ports are connected to alpha ports.

**Note:** Do not combine implicit and explicit routing.

## Controls

Controls determine the behavior of a node or path and provide information about them. Controls are specific to the path and node, and can also be device-dependent, depending on the control type. In general, controls on a video node are independent of controls on a memory or screen node. Even though controls on different types of nodes have the same names, they have different meanings, different units, and different behavior, depending on what node class they control.

The type definition of a VL control is:

```
typedef int VLControlType;
```

To get the value of a control, call **vlGetControl()**:

```
int vlGetControl ( VLserver svr, VLPath path, VLnode node,
                   VLControlType type, VLControlValue *value )
```

The control is located according to the *svr*, *path*, *node*, and *type* and its value is returned in a pointer to a VLControlValue structure:

```
typedef union {
 VLFraction       fractVal;
 VLBoolean        boolVal;
 int              intVal;
 VLXY             xyVal;
 char             stringVal[96];
 float            matrixVal[3][3];
 uint             pad[24];
 VLExtendedValue  extVal;
} VLControlValue;

typedef struct {
 int  x, y;
} VLXY;

typedef struct {
 int  numerator;
 int  denominator;
} VLFraction;
```

To obtain information about the valid values for a given control, call **vlGetControlInfo()**:

```
VLControlInfo *vlGetControlInfo ( VLserver svr, VLPath path, VLnode node,
                                  VLControlType type )
```

The control is located according to the *svr*, *path*, *node*, and *type* and its value is returned in a pointer to a VLControlInfo structure:

```
typedef struct __vlControlInfo {
 char                name[VL_NAME_SIZE]; /* name of control */
 VLControlType       type;            /* e.g. WINDOW, HUE */
 VLControlClass      ctlClass;       /* SLIDER, DETENT, KNOB, BUTTON */
 VLControlGroup      group;          /* BLEND, VISUAL QUALITY, SYNC */
 VLNode              node;           /* associated node */
 VLControlValueType  valueType;      /* what kind of data */
 int                 valueCount;     /* how many data items */
 int                 numFractRanges; /* number of ranges */
 VLFractionRange     *ranges;        /* range of values of control */
 int                 numItems;       /* number of enumerated items */
 VLControlItem       *itemList;      /* the actual enumerations */
} VLControlInfo;
```

These controls are highly interdependent, so the order in which they are set is important. In most cases, the value being set takes precedence over other values that were previously set.

There are 2 types of controls: “path” controls and “device” controls. The distinction between the two is detailed below.

Path controls are controls such as VL_SIZE, VL_OFFSET, and VL_ZOOM, which are capable of actively controlling a transfer. These controls are private to a path and any changes (with some exceptions) cause events to be sent *only* to the process owning the path. These controls are active while the path is transferring, and retain their values when the transfer is suspended for any reason. In practice, this means that the user program can set up the desired transfer controls, and then restart a preempted transfer without restoring controls to their previous values.

Device controls are controls such as VL_BRIGHTNESS and VL_CONTRAST, which are outside the realm of a “path” and can possibly effect the data that another path is processing. Because most of these controls directly affect some hardware change, they retain their values after the paths are removed.

### Establishing the Default Input Source

VL_DEFAULT_SOURCE specifies which of the input nodes is to be considered the “default” input. This is automatically setup when the video driver is loaded according to Table 4-1, which indicates which input signal(s) are active.

**Table 4-1** Default Video Source

| S-video | Composite | Camera | Default_Source |
|---|---|---|---|
| yes | x | x | svideo |
| no | yes | x | composite |
| no | no | yes | camera |
| no | no | no | composite |

For example, if a VCR is connected to the SVideo input and it is powered on, then it is the default input.

When the VL_DEFAULT_SOURCE is changed, a VLDefaultSource event is sent to all processes that have this event enabled in their vlEventMask.

## Getting Video Source Controls

Most source controls are read-only values that are set either by the user (from the Video Control Panel) or automatically, according to the characteristics of the video input signal. However, reading the values of these controls is useful for obtaining information about the input video stream that is necessary for setting controls on the drain node.

### Getting Video Input Format Using the VL_FORMAT Control

The VL_FORMAT control on the video source node is usually set using the Video Control Panel. It is often of no concern to a vid-to-mem application, except with Sirius video, where it is used to determine color space conversion.

VL_FORMAT selects the input video format (use VL_MUXSWITCH if there are more than one to select):

- VL_FORMAT_COMPOSITE selects analog composite video
- VL_FORMAT_SVIDEO selects analog composite video
- VL_FORMAT_DIGITAL_COMPONENT and VL_FORMAT_DIGITAL_COMPONENT_SERIAL select digital video
- VL_FORMAT_DIGITAL_INDYCAM and VL_FORMAT_DIGITAL_CAMERA select the connected camera

### Getting Video Input Timing Using the VL_TIMING Control

The VL_TIMING control on the video source node is usually set from the Video Control Panel. The input source timing also affects the value returned by the VL_SIZE control on the video source node.

Use VL_TIMING to determine whether the input source timing is PAL or NTSC, and whether the input pixels are square or not. Knowing whether the input signal is PAL or NTSC timing is useful for setting the VL_RATE control on the memory drain node. (For Sirius Video, it is also used to determine the value for the VL_TIMING control on the memory drain node. An easy way to set the VL_TIMING value for the memory node is

to read the value of the VL_TIMING control from the video source node, and then set that value into the VL_TIMING control for the memory node.

The VL_TIMING control is an integer value that adjusts the video filter for different video standards.

The 525 (NTSC) or 625 (PAL) timing standards are specified and the pixels are considered to be in the accepted video aspect ratio for those standards (also known as “non-square”) for VL_TIMING_525_CCIR601 and VL_TIMING_625_CCIR601

The 525 (NTSC) or 625 (PAL) timing standards are specified and, depending on the VL video device and the connector type, a non-square to square pixel filter can be engaged so that in memory, the pixels are in a 1:1 aspect ratio (which is compatible with OpenGL) for VL_TIMING_525_SQ_PIX and VL_TIMING_625_SQ_PIX.

When these timings are applied to a path that has a standard digital camera attached, then the 525 (NTSC) or 625 (PAL) timing standards are interpreted to mean that the external pixels are in a 1:1 aspect ratio, and there is no non-square format available for the internal pixels. If a non-square imager becomes available, then non-square pixels will be available. The pixel conversion applies a ratio of 11/10 for NTSC, and a ratio of 11/12 for PAL.

**Note:** The application program should always check the default VL_SIZE after a timing change to determine the size of the resultant images.

#### Getting Video Input SIze Using the VL_SIZE Control

The VL_SIZE control on the video source node is a read-only control. The *x* and *y* values returned by this control are affected by the setting of the VL_TIMING control on the video source node. The x and y values of this control are not, in general, affected by the settings of any controls in the memory drain node, including VL_ZOOM, VL_SIZE, and VL_CAP_TYPE.

The *x* component value of this control reveals the width, in pixels, of the unzoomed, unclipped video input images (in fields or frames, depending on the VL_CAP_TYPE). The meaning of the *y* component value of the video source node’s VL_SIZE control depends on the video device. On Sirius, the *y* value is the number of pixel rows in each field, and includes the count of rows of pixel samples taken from the field’s Vertical Retrace Interval. On EV1 and VINO, the y value is the number of pixel rows in each frame (pair of fields), and does not include any pixel rows from the Vertical Retrace Interval.

## Setting Memory Drain Node Controls

This section describes setting controls on the memory drain node.

### Setting the Memory Packing Controls Using the VL_PACKING control

A vid-to-mem application chooses the color space (that is. the set of components that make up each pixel, for example: RGB, RGBA, YUV, YCrCb, Y, YIQ) and the particular packing of those pixel components into memory using the VL_PACKING control (on all video devices) and also with the VL_FORMAT control on Sirius video.

On all VL video devices except Sirius, VL_FORMAT is not applicable to memory drain nodes, and VL_PACKING is used to select the color space as well as the pattern by which the components are packed into memory buffers. Packings that imply RGB or RGBA color spaces select those spaces. Packings that imply Y, or YUV or YCrCb color spaces select one of those spaces.

### Setting the Memory Capture Mode Using the VL_CAP_TYPE Control

On all VL video devices except Sirius, the capture mode can be set by the application. Its setting determines whether the images in the buffers returned by the VL are individual fields, or interleaved frames, or pairs of non-interleaved fields.

VL_CAP_TYPE specifies the capture mode:

- VL_CAPTURE_INTERLEAVED captures or sends buffers that contain both the F1 and F2 fields interlaced in memory. A side effect of changing from “non-interleaved” to “interleaved” is that the VL_RATE will be halved.
- VL_CAPTURE_NONINTERLEAVED captures or sends buffers that contain only one field each but are transferred in pairs keeping the F1 and even field of a picture together. A side effect of this characteristic, if a transfer error occurs in the second field, then the first is not transferred.
- VL_CAPTURE_FIELDS captures or sends buffers that contain only one field each and are transferred individually. Since these are separate fields then VL_RATE is effective on individual fields, and a single field may be dropped. Also, changing from “interleaved” to “fields” causes the VL_RATE to be doubled.
- VL_CAPTURE_EVEN_FIELDS captures only the F1 fields. For output the field is transferred during both field times.

- VL_CAPTURE_ODD_FIELDS captures only the F1 fields. For output the field is transferred during both field times

There is no single VL_CAP_TYPE that is available, and implemented in the same way, on all VL video devices. VL_CAPTURE_NONINTERLEAVED is available on all devices, but has different meanings on different platforms. VL_CAPTURE_INTERLEAVED, VL_CAPTURE_EVEN_FIELDS, and VL_CAPTURE_ODD_FIELDS are available and common to all VL video devices except Sirius.

On Sirius Video, VL_CAP_TYPE is read-only, and is permanently set to VL_CAPTURE_NONINTERLEAVED. Each captured buffer contains exactly one field, unclipped, unzoomed, with *n* leading pixel rows of samples from the vertical retrace interval.

EV1 implements VL_CAPTURE_NONINTERLEAVED differently from all other VL video devices. On all VL video devices except EV1, when VL_CAP_TYPE is set to VL_CAPTURE_NONINTERLEAVED, each image buffer that the VL gives to the application contains one field, either F1 or F2, and VL_RATE (the rate at which these buffers are returned) is in fields per second, not frames per second. But on EV1 video devices, when VL_CAP_TYPE is set to VL_CAPTURE_NONINTERLEAVED, each image buffer contains two non-interleaved fields, and VL_RATE is in frames per second.

#### Setting the Memory Capture Target Rate Using the VL_RATE Control

On all VL video devices except Sirius, VL_RATE sets the target rate (upper bound) of image buffers per second to be captured and returned to the application. The VL will not deliver more buffers per second than the rate you specify, but it can deliver less.

The contents of each image buffer is either a frame or a field, as determined by the VL_CAP_TYPE control. Accordingly, VL_RATE is in units of fields per second or frames per second, as determined by the VL_CAP_TYPE control.

VL_RATE is effective on a pair of fields, though it is still interpreted as a field rate. What this means is that if a field is to be dropped because of the effects of VL_RATE, then both fields are dropped (for output, if the VL_RATE causes some fields to be dropped, then the preceding fields are repeated). Also, changing from “interleaved” to “non-interleaved” mode causes the VL_RATE to be doubled.

VL_RATE is expressed as a fractional value (an integer numerator divided by an integer denominator) and ranges from the maximum rate (60/1 for NTSC, 50/1 for PAL, and half of each value for VL_CAPTURE_INTERLEAVED) down to 1/0xffff in any increment.

Both the numerator and denominator must be specified. The usual value for the denominator is 1. Some devices convert the fraction to an integer number of images per second by truncating rather than rounding, so using values that are equivalent to integer values is the safest thing to do. Because VL_RATE is a fraction, **vlGetControlInfo()** cannot be used to obtain the minimum or maximum values for VL_RATE.

Acceptable values are determined from the following list of devices:

- VL_CAPTURE_NONINTERLEAVED for all devices except EV1 and Sirius
  - NTSC: all multiples of 10 and 12 between 10 and 60
  - PAL: all multiples of 10 between 10 and 50
- VL_CAPTURE_NONINTERLEAVED (EV1)
  - NTSC: all multiples of 5 and 6 between 5 and 30
  - PAL: all multiples of 5 between 5 and 25
- VL_CAPTURE_INTERLEAVED, VL_CAPTURE_EVEN_FIELDS, and VL_CAPTURE_ODD_FIELDS
  - NTSC: all multiples of 5 and 6 between 5 and 30
  - PAL: all multiples of 5 between 5 and 25
- VL_CAPTURE_NONINTERLEAVED for Sirius Video. This control is read-only. Its value is determined by the setting of the VL_TIMING control on the memory node.
  - NTSC: 60 fields per second
  - PAL: 50 fields per second

VINO's VL_RATE cannot be set to a value less than 5/1.

## Setting Video Capture Region Controls

Figure 4-1 shows a diagram of an NTSC F1 field.

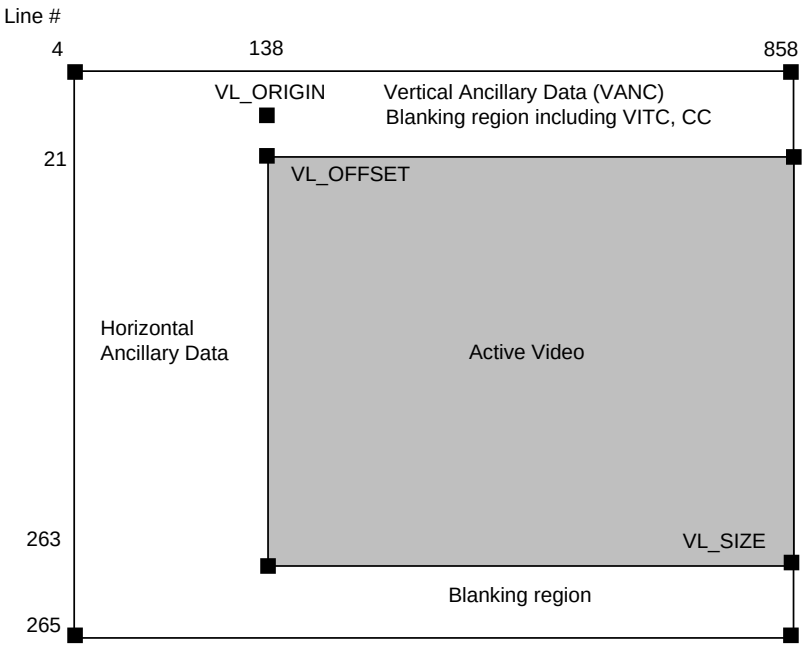



**Figure 4-1** Video Image Parameter Controls

The data contained within the area labeled “Active Video” is the default data transferred to and from memory, but the hardware and video driver allow the transfer to include most all the portion of the “hidden” video, or the Horizontal and/or Vertical Ancillary Data (HANC/VANC).

The following controls specify the capture region (all these controls are path controls):

- VL_ORIGIN is used on the screen capture device to specify the origin of the capture area. For Video input, the VL_ORIGIN can be used to specify a “black fill” region.
- VL_OFFSET is used on a source or drain memory node to specify an (*x*, *y)* value that signifies the upper left corner of the active video region. For input, the area to the left and above the VL_OFFSET is omitted. For output, the same region is filled with “black”.

The VL_OFFSET values are in “ZOOMED” coordinates (see VL_ZOOM below). VL_OFFSET has a default of 0,0. Negative values of VL_OFFSET specify non-picture data such as horizontal and vertical ancillary data, which must be decoded separately from the picture data.

Certain restrictions apply to the value of VL_OFFSET. The resultant offset must be on a 2-pixel boundary, and the minimum offset is restricted to the values listed in the reference pages for the VL video devices. See also “Using VL_SIZE and VL_OFFSET on the Memory Drain Node,”for detailed information about these values for memory drain nodes.

**Note:** The actual minimum offset is affected by VL_ZOOM and VL_ASPECT (see VL_ASPECT below).

- VL_SIZE is used on a source or drain memory node to specify an (*x*, *y)* value that defines the extent of the active video region. Adding the VL_SIZE coordinates to the VL_OFFSET coordinates gives the coordinates of the lower right corner of the active video region (VL_OFFSET + VL_SIZE = lower right corner). For input, the area to the right and below this corner is omitted. For output, the same region is filled with “black”.

  The VL_SIZE values are in “ZOOMED” coordinates. See “Using VL_SIZE and VL_OFFSET on the Memory Drain Node,”for details about VL_SIZE values.

  Certain restrictions apply to the value of VL_SIZE: the resultant size must be on a 2 pixel boundary and the number of bytes to be transferred must be a multiple of 8.

  The maximum VL_SIZE is defined by the total number of lines in the video standard. Increasing the VL_SIZE beyond the maximum horizontal dimension causes VL_OFFSET to assume negative values. Out of range values return vlErrno VLValueOutOfRange.

The use of all these controls is explained in the sections that follow.

### Using VL_SIZE and VL_OFFSET on the Memory Drain Node

This section discusses the VL_SIZE control and the VL_OFFSET control on the memory drain node.

The VL_SIZE control on the memory drain node determines the number of rows of pixels, and the number of pixels in each row, in each image buffer (field or frame) that the VL returns to the application. If zooming (decimation) is being done, the VL_SIZE control on the memory drain node specifies the size of the image after it has been decimated.

The VL_SIZE control on the memory drain node can be used to “clip” a region out of an image by setting the X and/or Y components to values that are smaller than the size of the captured (and decimated, if applicable) image.

When the (possibly decimated) image is being clipped, the clipped region does not have to come from the upper left hand corner of the (possibly decimated) source image. The VL_OFFSET control on the memory drain node determines the number of top pixel rows to skip and the number of leading pixels to skip in each row to find the first pixel in the (possibly decimated) image to place in the image buffer, the first pixel of the clipping region.

When zooming (decimation) is being used, VL_OFFSET is always in coordinates of the zoomed image. It is as if the entire source image is decimated down, and then the clipping function is applied to the decimated image. In practice, the hardware usually clips before decimating, but the VL API always specifies the VL_OFFSET in the coordinates of the decimated (virtual) image.

On all VL devices except Sirius, the vertical (Y) component of VL_OFFSET may be specified with a negative value. This causes the clipping region to include row of samples taken before the top of the image, e.g. rows from the Vertical Retrace Interval. This feature is usually used with VL_ZOOM of 1/1, since the information in the Vertical Retrace Interval isn’t an image and doesn’t make sense to decimate or average, at least not in the vertical direction.

The VL imposes these requirements on the values of VL_OFFSET and VL_SIZE:

- The sum of the vertical components of VL_OFFSET and VL_SIZE must not exceed the height of the virtual (zoomed) image, and
- The sum of the horizontal components of VL_OFFSET and VL_SIZE must not exceed the width of the virtual (zoomed) image. When an attempt to set either one of these controls would violate either of the rules above, the call to **vlSetControl()** fails with the vlErrno VLValueOutOfRange, and the offending component (horizontal or vertical) is set to the largest non-negative value that does not violate the rule, or to zero if no such non-negative value exists.

VL_OFFSET and VL_SIZE cannot be both set in one atomic operation. A change in either component of either control could violate one of the rules above (or below), especially after VL_ZOOM is set to a smaller fraction. It may be necessary to alternately and repeatedly set VL_OFFSET and VL_SIZE until no VLValueOutOfRange errors are reported.

Every VL video device places additional limitations on the range of acceptable values of VL_SIZE and VL_OFFSET. Each device has different limitations.

- Sirius doesn't clip at all. VL_SIZE and VL_OFFSET are read-only in Sirius.
- EV1 supports clipping only in the vertical (Y) direction. The entire width of the (possibly decimated) image is always placed in the image buffer. Application-specified horizontal clipping values are ignored.
- VINO imposes an additional list of requirements on VL_SIZE and VL_OFFSET. See below. VINO imposes the following additional clipping requirements:
- The right side edge of the clipped image must always coincide with the right side edge of the virtual (possibly decimated) image. That is, the clipped image must always come from the right side of the (possibly decimated) source image. Consequently, when vlSetControl is called to set the VL_OFFSET or VL_SIZE control on a memory node, if the sum of the horizontal components of the (new) settings of VL_OFFSET and VL_SIZE is less than the width of the virtual (zoomed) image, the vlSetControl call will succeed, and the horizontal component of the other control will be adjusted so that the sum of the two components exactly equals the width of the virtual (zoomed) image. This is done only in the horizontal direction.
- Each pixel row in the image buffer must be a multiple of 8-bytes in length. This means that the horizontal component of VL_SIZE must be a multiple of 2, 4, or 8 pixels, depending on the pixel packing (size of the individual pixels in memory).

#### Using VL_ZOOM on the Memory Drain Node

VL_ZOOM controls the expansion or decimation of the video image. Values greater than one expand the video; values less than one perform decimation. The only value of VL_ZOOM that works on all VL devices is 1/1. Acceptable values for vid-to-mem applications follow.

VINO may exhibit the following effects at these decimation factors: 1/4, 1/5, 1/6, 1/7, and 1/8:

- Y values that are not adjacent horizontally are averaged together
- the decimated images appear extremely green.

As a workaround, the VINO driver implements decimation by 1/4 and 1/6 by decimating in hardware by 1/2 or 1/3, and then decimates by an additional factor of 1/2 in software. This produces acceptable looking images, but at significant cost in CPU time.

The three other VL_ZOOM factors, 1/5, 1/7, and 1/8 all exhibit the green image effect described above.

For example, the listed zoom factors on VINO may behave as follows:

| | |
|---|---|
| 1/1, 1/2, 1/3 | implemented in hardware, looks OK. |
| 1/4, 1/6 | implemented partially in hardware, partially in software. Looks OK, but slower and uses 10% of an R4600 CPU. |
| 1/5, 1/7, 1/8 | implemented in hardware. Exhibits green shift. |

For example, the listed zoom factors on EV1 may behave as follows:

| | |
|---|---|
| 1/1, 1/2, 1/4, 1/8 | works OK for vid-to-mem |
| 1/3, 1/5, 1/7 | works only for vid-to-screen, not vid-to-mem, and only with VL_CAPTURE_INTERLEAVED |
| 2/1, 4/1 | works only for vid-to-screen, not vid-to-mem. |

**Note:** Sirius and Galileo 1.5 only accept a 1/1 zoom factor (Sirius and Galileo 1.5 don't zoom).

VL_ZOOM specifies the decimation of the input video to some fraction of its original size. Scaling from 1/1 down to 1/256 is available; the actual increments are: 256 to 1/256. The actual zoom value is affected by VL_ASPECT.

**Note:** VL_ZOOM is available only on the VL_DRN/VL_MEM (input) node.

VL_SYNC selects the type of sync used for video output. The choices are:

- VL_SYNC_INTERNAL means that the timing for the output is generated using an internal oscillator appropriate for the timing required (NTSC or PAL).
- VL_SYNC_GENLOCK means that the timing for the output is "genlocked" to the VL_SYNC_ SOURCE.
- VL_SYNC_SOURCE selects which sync source is used when VL_SYNC is set to VL_SYNC_GENLOCK.

VL_LAYOUT specifies the pixel layout (same as DM_IMAGE_LAYOUT):

- VL_LAYOUT_LINEAR means that video pixels are arranged in memory linearly.

- VL_LAYOUT_GRAPHICS means that video pixels are arranged in memory in a Pbuffer fashion that is compatible with the O2 OpenGL.
- VL_LAYOUT_MIPMAP means that video pixels are arranged in memory in a texture or mipmapped fashion that is compatible with the O2 OpenGL.

### Signal Quality Controls

The following signal quality controls are available (as supported by the video device):

- VL_BRIGHTNESS
- VL_CONTRAST
- VL_H_PHASE
- VL_HUE
- VL_SATURATION
- VL_RED_SETUP
- VL_GREEN_SETUP
- VL_GRN_SETUP
- VL_BLUE_SETUP
- VL_BLU_SETUP
- VL_ALPHA_SETUP
- VL_V_PHASE

Each of these controls is defined if they are provided in the analog encoder or decoder. They are not available in the digital domains.

VL_SIGNAL can be either VL_SIGNAL_NOTHING, VL_SIGNAL_BLACK, or VL_SIGNAL_REAL_IMAGE

VL_FLICKER_FILTER enables or disables the “flicker” filter.

VL_DITHER_FILTER enables or disables the “dither” filter.

VL_NOTCH_FILTER enables or disables the “notch” filter.

To determine default values, use **vlGetControl()** to query the values on the video source or drain node before setting controls. For all these controls, it pays to track return codes. If the value returned is VLValueOutOfRange, the value set is not what you requested.

Table 4-2 summarizes the VL controls. For each control the ASCII name of the control, the type of value it takes, and the node types and classes to which it can be applied is listed.

**Table 4-2** Summary of VL Controls

| Control | ASCII Name | Value | Node Type/Class |
|---|---|---|---|
| VL_DEFAULT_SOURCE | default_input | intVal | VL_SRC / VL_VIDEO |
| VL_TIM ING | timing | intVal | VL_SRC / VL_VIDEO |
| VL_ORIGIN | origin | xyVal | VL_ANY / VL_MEM |
| VL_SIZE | size | xyVal | VL_ANY / VL_MEM |
| VL_RATE | fieldrate | fractVal | VL_ANY / VL_MEM |
| VL_ZOOM | zoom | fractVal | VL_ANY / VL_MEM |
| VL_ASPECT | aspect | fractVal | VL_ANY / VL_MEM |
| VL_CAP_TYPE | fieldmode | intVal | VL_ANY / VL_MEM |
| VL_PACKING | packing | intVal | VL_ANY / VL_MEM |
| VL_FORMAT | format | intVal | VL_SRC / VL_VIDEO, VL_ANY / VL_MEM |
| VL_SYNC | sync | intVal | VL_DRN / VL_VIDEO |
| VL_SYNC_SOURCE | sync_source | intVal | VL_DRN / VL_VIDEO |
| VL_LAYOUT | layout | intVal | VL_ANY / VL_MEM |
| VL_SIGNAL | signal | intVal | VL_DRN / VL_VIDEO |
| VL_FLICKER_FILTER | flicker_filter | boolVal | VL_SRC / VL_SCREEN |
| VL_DITHER_FILTER | dither_filter | boolVal | VL_SRC / VL_VIDEO |
| VL_NOTCH_FILTER | notch_filter | boolVal | VL_DRN / VL_VIDEO |

The ASCII name is used to assign values to controls in the VL Resources file and can also be found in the control table returned by **vlGetControlList()**.

The following list is a key to which nodes the control can be applied:

- VL_SRC/VL_VIDEO - source video node
- VL_DRN/VL_VIDEO - drain video node
- VL_ANY/VL_VIDEO - source or drain video node
- VL_SRC/VL_SCREEN - source screen node
- VL_SRC/VL_MEM - source memory node
- VL_DRN/VL_MEM - drain memory node
- VL_ANY/VL_MEM - source or drain memory node

## Video Events

Video events provide a way to monitor the status of a video I/O stream. Typically, a number of events are combined into an event mask that describes the events of interest. Use **vlSelectEvents()** to specify the events you want to receive. Its function prototype is

```
int vlSelectEvents( VLServer vlSvr, VLPath path, VLEventMask eventmask )
```

where

*vlSvr* names the server to which the path is connected

*path* specifies the data path.

*eventmask* specifies the event mask; Table 4-3 lists the possibilities

Table 4-3 lists and describes the VL event masks.

**Table 4-3** VL Event Masks

| Symbol | Meaning |
|---|---|
| VLStreamBusyMask | Stream is locked |
| VLStreamPreemptedMask | Stream was grabbed by another path |
| VLStreamChangedMask | Video routing on this path has been changed by another path |
| VLAdvanceMissedMask | Time was already reached |
| VLSyncLostMask | Irregular or interrupted signal |

**Table 4-3 (continued)** VL Event Masks

| Symbol | Meaning |
|---|---|
| VLSequenceLostMask | Field or frame dropped |
| VLControlChangedMask | A control has changed |
| VLControlRangeChangedMask | A control range has changed |
| VLControlPreemptedMask | Control of a node has been preempted, typically by another user setting VL_LOCK on a path that was previously set with VL_SHARE |
| VLControlAvailableMask | Access is now available |
| VLTransferCompleteMask | Transfer of field or frame complete |
| VLTransferFailedMask | Error; transfer terminated; perform cleanup at this point, including **vlEndTransfer()** |
| VLEvenVerticalRetraceMask | Vertical retrace event, even field |
| VLOddVerticalRetraceMask | Vertical retrace event, odd field |
| VLFrameVerticalRetraceMask | Frame vertical retrace event |
| VLDeviceEventMask | Device-specific event, such as a trigger |
| VLDefaultSourceMask | Default source changed |

When transferring video, the main event is a VLTransferComplete.

## Video I/O Model

In the traditional video I/O model, you use the buffering, data transfer, and event handling routines supplied by the VL. One of the consequences of this approach is that it might require you to copy data passed outside the VL. (See the next chapter for the DMbuffers I/O method for O2 workstations.)

A basic VL application has the following components:

Preliminary path setup:

- **vlOpenVideo()** - open the video server

- **vlGetDeviceList()** - discover which devices and nodes are connected to this system.
- **vlGetNode()** - get the source and drain nodes
- **vlCreatePath()** - create a video path with the source and drain nodes specified.
- **vlSetupPath()** - set the path up to be usable given the access requested.
- **vlDestroyPath()** - remove a video path.

Specific control settings:

- **vlSetControl()** - set various parameters associated with the video transfer.
- **vlGetControl()** - get various parameters associated with the video transfer.

Preparing to capture or output video to/from memory:

- **vlCreateBuffer()** - create a VLbuffer
- **vlRegisterBuffer()** - register this buffer with the path

Starting and controlling the video transfer:

- **vlBeginTransfer()** - initiate the transfer
- **vlEndTransfer()** - terminate the transfer
- **vlNextEvent()** - handle events from the video device
- **vlGetNextValid()** - get incoming buffers with captured video
- **vlPutValid()** - send outgoing buffers with inserted video

## Freezing Video

Showing a still frame from a recorded video sequence (either uncompressed or compressed using JPEG) presents an enigma. Displaying a still frame requires a complete set of spatial information at a single instant of time—the data is simply not available to display a still frame correctly.

One way to display a still frame is to combine the lines from two adjacent fields, as shown in Figure 4-2. No matter which pair of fields you choose, the resulting still frame exhibits artifacts.

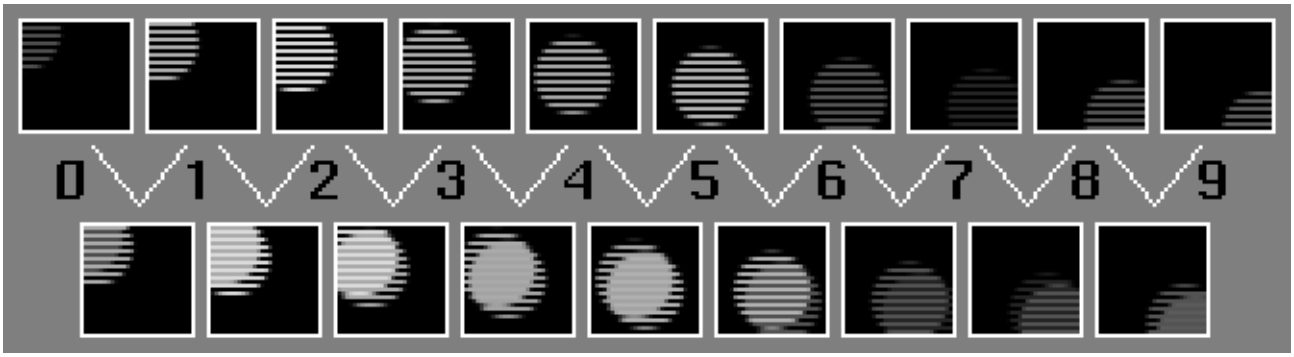



**Figure 4-2** Tearing

Figure 4-2 shows a display artifact known as *tearing* or *fingering*, which is an inevitable consequence of putting together an image from bits of images snapped at different times. You don't notice the artifact if the fields are flashed in rapid succession at the field rate, but when you try to freeze motion and show a frame, the effect is visible. You wouldn't notice the artifact if the objects being captured were not moving between fields.

These types of artifacts cause trouble for most compressors. If you are capturing still frames in order to pass frame-sized images on to a compressor, you definitely should avoid tearing. A compressor will waste lots of bits trying to encode the high-frequency information in the tearing artifacts and fewer bits encoding your actual picture. Depending on the size and quality of compressed image you want, you might consider sending every other field (perhaps decimated horizontally) to the compressor, rather than trying to create frames that will compress well.

Another possible technique for producing still-frames is to double the lines in a single field, as shown in Figure 4-3.

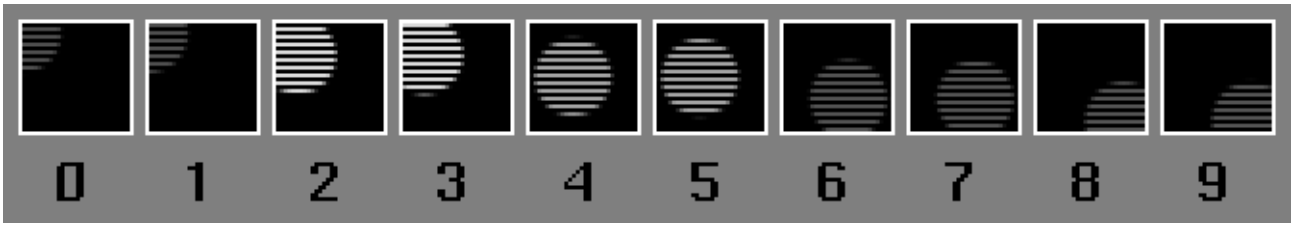



**Figure 4-3** Line Doubling on a Single Field

This looks a little better, but there is an obvious loss of spatial resolution (as evidenced by the visible “jaggies” and vertical blockiness).

To some extent, this can be reduced by interpolating adjacent lines in one field to get the lines of the other field, as shown in Figure 4-4.

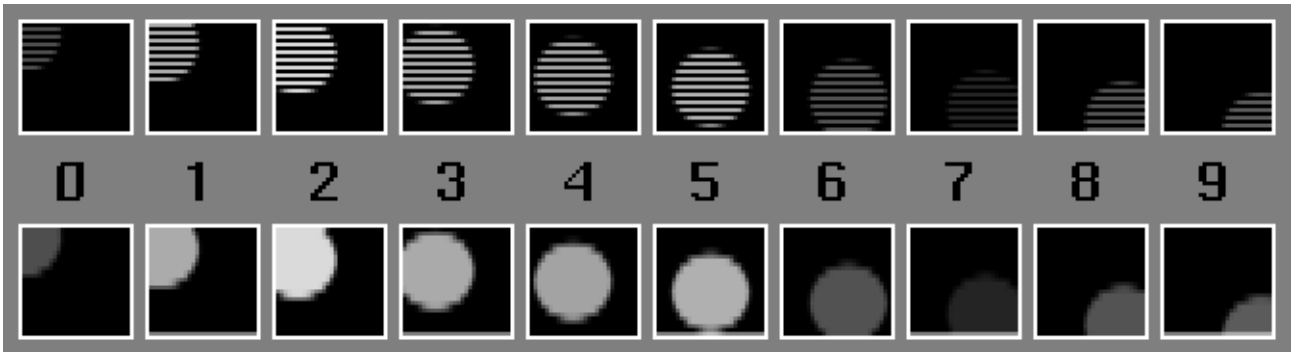



**Figure 4-4** Interpolating Alternate Scan Lines from Adjacent Fields

There are an endless variety of more elaborate tricks you can use to come up with good still frames, all of which come under the heading of “de-interlacing methods.” Some of these tricks attempt to use data from both fields in areas of the image that are not moving (so you get high spatial resolution), and double or interpolate lines of one field in areas of the image that are moving (so you get high temporal resolution). Many of the tricks take more than two fields as input. Since the data is simply not available to produce a spatially complete picture for one instant, there is no perfect solution. But depending on why you want the still frame, the extra effort may well be worth it.

When a CRT-based television monitor displays interlaced video, it doesn’t flash one frame at a time on the screen. During each field time (each 50th or 60th of a second), the CRT lights up the phosphors of the lines of that field only. Then, in the next field interval, the CRT lights up the phosphors belonging to the lines of the other field. So, for example, at the instant when a pixel on a given picture line is refreshed, the pixels just above and below that pixel have not been refreshed for a 50th or 60th of a second, and will not be refreshed for another 50th or 60th of a second.

So if that’s true, then why don’t video images don’t flicker or jump up and down as alternate fields are refreshed?

This is partially explained by the persistence of the phosphors on the screen. Once refreshed, the lines of a given field start to fade out slowly, and so the monitor is still emitting some light from those lines when the lines of the other field are being refreshed. The lack of flicker is also partially explained by a similar persistence in your visual system.

Unfortunately though, these are not the only factors. Much of the reason why you do not perceive flicker on a video screen is that good-looking video signals themselves have built-in characteristics that reduce the visibility of flicker. It is important to understand these characteristics, because when you synthesize images on a computer or process digitized images, you must produce an image that also has these characteristics. An image which looks good on a non-interlaced computer monitor can easily look abysmal on an interlaced video monitor.

A complete understanding of when flicker is likely to be perceivable and how to get rid of it requires an in-depth analysis of the properties of the phosphors of a particular monitor (not only their persistence but also their size, overlap, and average viewing distance), it requires more knowledge of the human visual system, and it may also require an in-depth analysis of the source of the video (for example, the persistence, size, and overlap of the CCD elements used in the camera, the shape of the camera's aperture, etc.). This description is only intended to give a general sense of the issues.

Standard analog video (NTSC and PAL) has characteristics (such as bandwidth limitations) which can introduce many similar artifacts to the ones we are describing here into the final result of video output from a computer. These artifacts are beyond the scope of this document, but are also important to consider when creating data to be converted to an analog video signal. Examples of this would be antialiasing (blurring) data in a computer to avoid chroma aliasing when the data is converted to analog video.

Here are some of the major areas to be concerned about when creating data for video output:

- Abrupt Vertical Transitions: One-Pixel-High Lines

  First of all, typical video images do not have abrupt vertical changes. For example, say you output an image that is entirely black except for one, one-pixel-high line in the middle.

Since the non-black data is contained on only one line, it will appear in only one field. A video monitor will only update the image of the line 30 times a second, and it will flicker on and off quite visibly. To see this on a video-capable machine, run *videoout*, turn off the anti-flicker-filter, and point *videoout*'s screen window at the image above.

You do not have to have a long line for this effect to be visible: thin, non-antialiased text exhibits the same objectionable flicker.

Typical video images are more vertically blurry; even where there is a sharp vertical transition (the bottom of an object in sharp focus, for example), the method typical cameras use to capture the image will cause the transition to blur over more than one line. It is often necessary to simulate this blurring when creating synthetic images for video.

- Abrupt Vertical Transitions: Two-Pixel-High Lines

  These lines include data in both fields, so part of the line is updated each 50th or 60th of a second. Unfortunately, when you actually look at the image of this line on a video monitor, the line appears to be solid in time, but it appears to jump up and down, as the top and bottom line alternate between being brighter and darker. You can also see this with the *videoout* program.

- Flicker Filter

  The severity of both of these effects depends greatly on the monitor and its properties, but you can pretty much assume that someone will find them objectionable. One partial solution is to vertically blur the data you are outputting. Turning on the "flicker filter" option to *videoout* will cause some boards (such as ev1) to vertically prefilter the screen image by a simple 3-tap (1/4,1/2,1/4) filter. This noticeably improves (but does not remove) the flickering effect.

There is no particular magic method that will produce flicker-free video. The more you understand about the display devices you care about, and about when the human vision system perceives flicker and when it does not, the better a job you can do at producing a good image.

### Synthetic Imagery Must Also Consist of Fields

When you modify digitized video data or synthesize new video data, the result must consist of fields with all the same properties, but temporally offset and spatially disjointed. This may not be trivial to implement in a typical renderer without wasting lots of rendering resources (rendering 50/60 images a second, throwing out unneeded lines in each field) unless the developer has fields in mind from the start.

You might think that you could generate synthetic video by taking the output of a frame-based renderer at 25/30 frames per second and pulling two fields out of each frame image. This will not work well: the motion in the resulting sequence on an interlaced video monitor will noticeably stutter, due to the fact that the two fields are scanned out at different times, yet represent an image from a single time. Your renderer must know that it is rendering 50/60 temporally distinct images per second.

### Playing Back "Slow," or Synthesizing Dropped Fields

Two tasks which are relatively easy to do with frame-based data, such as movies, are playing slowly (by outputting some frames more than once) or dealing with frames that are missing in the input stream by duplicating previous frames. Certainly there are more elaborate ways to generate better-looking results in these cases, and they too are not so hard on frame-based data.

Suppose you are playing a video sequence, and run up against a missing field as shown in Figure 4-5 (the issues we are discussing also come up when you want to play back video slowly).

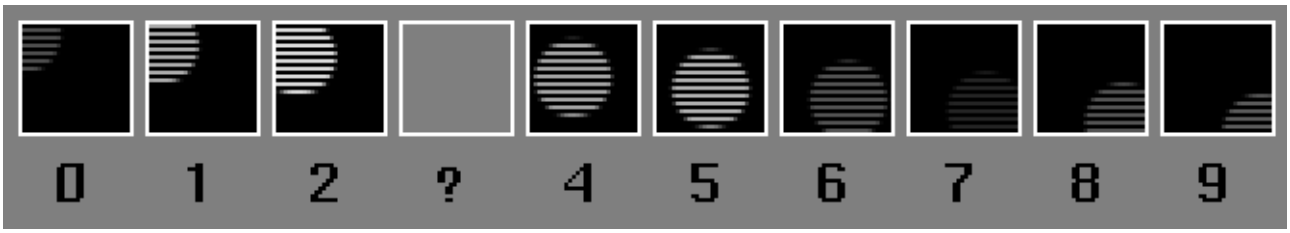



**Figure 4-5** Dropped Frame

To keep the playback rate of the video sequence constant, you need to put some video data in that slot, so which field do you choose? Suppose you chose to duplicate the previously displayed field (field 2), as shown in Figure 4-6.

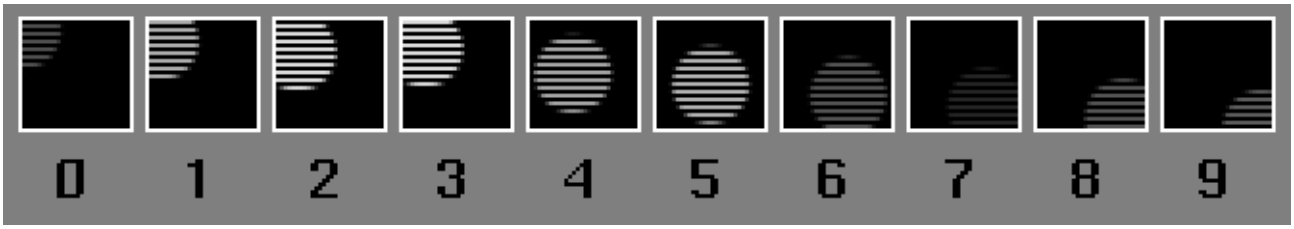



**Figure 4-6** Field Duplication

You could also try duplicating field 4 or interpolating between 2 and 4, but with all of these methods there is a crucial problem: those fields contain data from a different spatial location than the missing field. If you viewed the resulting video, you would immediately notice that the image visually jumps up and down at this point. This is a large-scale version of the same problem that made the two-pixel-high line jump up and down: your eye is very good at picking up on the vertical "motion" caused by an image being drawn to the lines of one field, then being drawn again one picture line higher, into the lines of the other field. You would see this even if the ball was not in motion.

Suppose you instead choose to fill in the missing field with the last non-missing field that occupies the same spatial locations, as shown in Figure 4-7.

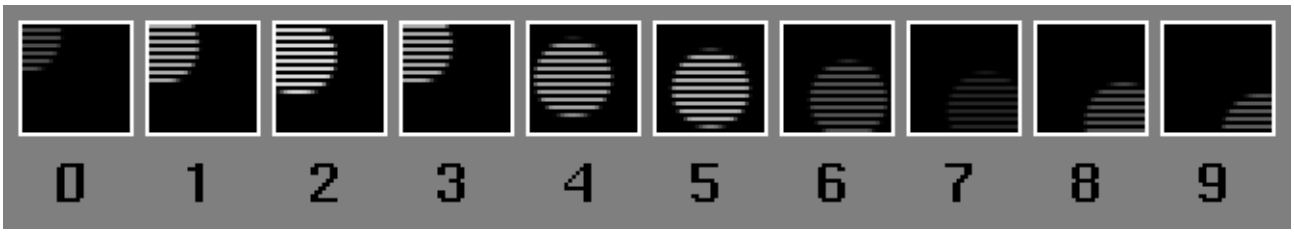



**Figure 4-7** Field Replacement

Now you have a more obvious problem: you are displaying the images temporally out of order. The ball appears to fly down, fly up again for a bit, and then fly down. Clearly, this method is not good for video which contains motion. But for video containing little or no motion, it would work pretty well, and would not suffer the up-and-down jittering of the previous approach.

Which of these two methods is best thus depends on the video being used. For general-purpose video where motion is common, you'd be better off using the first technique, the "temporally correct" technique. For certain situations such as computer

screen capture or video footage of still scenes, however, you can often get guarantees that the underlying image is not changing, and the second technique, the “spatially correct” technique, is best.

As with de-interlacing methods, there are many more elaborate methods for interpolating fields which use more of the input data. For example, you could interpolate 2 and 4 and then interpolate the result of that vertically to guess at the content of the other field’s lines. Depending on the situation, these techniques may or may not be worth the effort.

### Still Frames on Video Output

The problem of getting a good still frame from a video input has a counterpart in video output. Suppose you have a digitized video sequence and you want to pause playback of the sequence. Either you, the video driver, or the video hardware must continue to output video fields even though the data stream has stopped, so which fields do you output?

If you choose the “temporally correct” method and repeatedly output one field (effectively giving you the “line-doubled” look described above), then you get an image with reduced vertical resolution. But you also get another problem: as soon as you pause, the image appears to jump up or down, because your eye picks up on an image being drawn into the lines of one field, and then being drawn one picture line higher or lower, into the lines of another field. Depending on the monitor and other factors, the paused image may appear to jump up and down constantly or it may only appear to jump when you enter and exit pause.

If you choose the “spatially correct” method and repeatedly output a pair of fields, then if there happened to be any motion at the instant where you paused, you will see that motion happening back and forth, 60 times a second. This can be very distracting.

There are, of course, more elaborate heuristics that can be used to produce good looking pauses. For example, vertically interpolating an F1 to make an F2 or vice versa works well for slow-motion, pause, and vari-speed play. In addition, it can be combined with inter-field interpolation for “super slow-motion” effects.

## Audio I/O Concepts

This section describes how to get audio data in and out of the computer.

### Audio Library Programming Model

Programming audio I/O involves three basic concepts:

Audio device(s) The audio hardware used by the AL, which is shared among audio applications. Audio devices contains settings pertaining to the configuration of both the internal audio system and the external electrical connections.

ALport A one-way (input or output) audio data connection between an application program and the host audio system. An ALport contains:

- an audio sample queue, which stores audio sample frames awaiting input or output
- settings pertaining to the attributes of the digital audio data it transports

Some of the settings of an ALport are static; they cannot be changed once the ALport has been opened. Other settings are dynamic; they can be changed while an ALport is open.

ALconfig An opaque data structure for configuring these settings of an ALport:

- audio device (static setting)
- size of the audio sample queue (static setting)
- number of channels (static setting)
- format of the sample data (dynamic setting)
- width of the sample data (dynamic setting)
- range of floating point sample data (dynamic setting)

To enable audio input and output, your application must create and configure the required audio I/O connections. This section describes how to set up and use the Audio Library facilities that provide audio I/O capability.

## Audio Ports

An ALport provides a one-way (input or output) interface between an application program and the host audio system. More than one ALport can be opened by the same application; the number of ALports that can be active at the same time depends on the hardware and software configurations you are using. Open ALports use CPU resources, so be sure to close an ALport when I/O is completed and free the ALconfig when it is no longer needed.

An ALport consists of a queue, which stores audio data awaiting input or output, and static and dynamic state information.

Audio I/O is accomplished by opening an audio port and reading audio data from or writing audio data to the port. For audio input, the hardware places audio sample frames in an input port's queue at a constant rate, and your application program reads the sample frames from the queue. Similarly, for audio output, your application writes audio sample frames to an output port's queue, and the audio hardware removes the sample frames from the queue. A minimum of two ALports are necessary to provide input and output capability for an audio application.

## Using ALconfig Structures to Configure ALports

You can open an ALport with the default configuration or you can customize an ALconfig for configuring an ALport suited to your application needs.

The default ALconfig has:

- a buffer size of 100,000 sample frames
- stereo data
- a two's complement sample format
- a 16-bit sample width

These settings provide an ALport that is compatible with CD- and DAT-quality data, but if your application requires different settings, you must create an ALconfig with the proper settings before opening a port. The device, channel, and queue-size settings for an ALport are static—they cannot be changed after the port has been opened.

The steps involved in configuring and opening an ALport are listed below, followed by a sample code fragment that illustrates each of these steps. The sample program is followed by subsections that describe these concepts more fully and explain the use of each routine listed here.

1. Turn off the default error handler by passing a 0 to **ALseterrorhandler()**.
2. If the default ALconfig settings are satisfactory, you can simply open a default ALport by using 0 for the configuration in the **alOpenPort()** routine; otherwise, create a new ALconfig by calling **alNewConfig()**.
3. If nondefault values are needed for any of the ALconfig settings, set the desired values as follows:
   - Call **alSetChannels()** to change the number of channels.
   - Call **alSetQueueSize()** to change the sample queue size.
   - Call **alSetSampFmt()** to change the sample data format.
   - Call **alSetWidth()** to change the sample data width.
   - Call **alSetFloatMax()** to set the maximum amplitude of floating point data (not necessary for integer data formats).
4. Open an ALport by passing the ALconfig to the **alOpenPort()** routine.
5. Create additional ALports with the same settings by using the same ALconfig to open as many ports as are needed.

Example 4-1 demonstrates how to configure and open an output ALport that accepts floating point mono sample frames.

**Example 4-1** Configuring and Opening an ALport

```
ALconfig audioconfig;
ALport audioport;
int err;

void audioinit /* Configure an audio port */
{
ALseterrorhandler(0);
audioconfig = alNewConfig();

alSetSampFmt(audioconfig, AL_SAMPFMT_FLOAT);
alSetFloatMax(audioconfig, 10.0);
alSetQueueSize(audioconfig, 44100);
alSetChannels(audioconfig,AL_MONO);

audioport = alOpenPort("surreal","w",audioconfig);
if (audioport == (ALport) 0) {
    err = oserror();
    if (err == AL_BAD_NO_PORTS) {
        fprintf(stderr, " System is out of audio ports\n");
    } else if (err == AL_BAD_DEVICE_ACCESS) {
        fprintf(stderr, " Couldn't access audio device\n");
    } else if (err == AL_BAD_OUT_OF_MEM) {
        fprintf(stderr, " Out of memory\n");
    }
    exit(1);
}
```

The sections that follow explain how to use ALconfigs in greater detail.

### Creating a New ALconfig

To create a new ALconfig structure that is initialized to the default settings, call **alNewConfig()**. Its function prototype is:

```
ALconfig alNewConfig ( void )
```

The ALconfig that is returned can be used to open a default ALport, or you can modify its settings to create the configuration you need. In Example 4-1, the channel, queue size,

sample format, and floating point data range settings of an ALconfig named *audioconfig* are changed.

**alNewConfig()** returns an ALconfig structure upon successful completion; otherwise, it returns 0 and sets an error code that you can retrieve by calling oserror(3C). Possible errors include:

| | |
|---|---|
| AL_BAD_OUT_OF_MEM | insufficient memory available to allocate the ALconfig structure |

Audio ports are opened and closed by using **alOpenPort()** and **alClosePort()** respectively. Unless you plan to use the default port configuration, you should set up an ALconfig structure by using **alNewConfig()** and then use the routines for setting ALconfig fields, such as **alSetChannels()**, **alSetQueueSize()**, and **alSetWidth()** before calling **alOpenPort()**.

## Audio Sample Queues

Audio sample frames are placed in the sample queue of an ALport to await input or output. The audio system uses one end of the sample queue; the audio application uses the other end.

During audio input, the audio hardware continuously writes audio sample frames to the tail of the input queue at the selected input rate, for example, 44,100 sample pairs per second for 44.1 kHz stereo data. If the application can't read the sample frames from the head of the input queue at least as fast as the hardware writes them, the queue fills up and some incoming sample data is irretrievably lost.

During audio output, the application writes audio sample frames to the tail of the queue. The audio hardware continuously reads sample frames from the head of the output queue at the selected output rate, for example, 44,100 sample pairs per second for 44.1 kHz stereo data, and sends them to the outputs. If the application can't put sample frames in the queue as fast as the hardware removes them, the queue empties, causing the hardware to send 0-valued sample frames to the outputs (until more data is available), which are perceived as pops or breaks in the sound.

For example, if an application opens a stereo output port with a queue size of 100,000, and the output sample rate is set to 48 kHz, the application needs to supply ($2 \times 48{,}000 = 96{,}000$) sample frames to the output port at the rate of at least 1 set of sample frames per second, because the port contains enough space for about one second of stereo data at

that rate. If the application fails to supply data at this rate, an audible break occurs in the audio output.

On the other hand, if an application tries to put 40,000 sample frames into a queue that already contains 70,000 sample frames, there isn't enough space in the queue to store all the new sample frames, and the program will *block* (wait) until enough of the existing sample frames have been removed to allow for all 40,000 new sample frames to be put in the queue. The AL routines for reading and writing block; they do not return until the input or output is complete.

To allocate and initialize an ALport structure, call **alOpenPort()**. Its function prototype is:

```
ALport alOpenPort ( char *name, char *direction, ALconfig config )
```

where:

*name* — is an ASCII string used to identify the port for humans (much like a window title in a graphics program). The name is limited to 20 characters and should be both descriptive and unique, such as an acronym for your company name or the application name, followed by the purpose of the port

*direction* — specifies whether the port is for input or output:

- "*r*" configures the port for reading (input)
- "*w*" configures the port for writing (output)

*config* — is an ALconfig that you have previously defined or is null (0) for the default configuration.

Upon successful completion, **alOpenPort()** returns an ALport structure for the named port; otherwise, it returns a null-valued ALport, and sets an error code that you can retrieve by calling *oserror*(3C). Possible errors include:

| | |
|---|---|
| AL_BAD_CONFIG | *config* is either invalid or null |
| AL_BAD_DIRECTION | *direction* is invalid |
| AL_BAD_OUT_OF_MEM | insufficient memory available to allocate the ALport structure |
| AL_BAD_DEVICE_ACCESS | audio hardware is inaccessible |
| AL_BAD_NO_PORTS | no audio ports currently available |

**alClosePort()** closes and deallocates an audio port—any sample frames remaining in the port will not be output. Its function prototype is:

```
int alClosePort ( ALport port )
```

where:

*port* is the ALport you want to close

Example 4-2 opens an input port and an output port and then closes them.

**Example 4-2** Opening Input and Output ALports

```
input_port = alOpenPort("waycoolinput", "r", 0);
if (input_port == (ALport) 0 {
    err = oserror();
    if (err == AL_BAD_NO_PORTS) {
        fprintf(stderr, " System is out of audio ports\n");
    } else if (err == AL_BAD_DEVICE_ACCESS) {
        fprintf(stderr, " Couldn't access audio device\n");
    } else if (err == AL_BAD_OUT_OF_MEM) {
        fprintf(stderr, " Out of memory: port open failed\n");
    }
    exit(1);
}
...
output_port = alOpenPort("killeroutput", "w", 0);
if (input_port == (ALport) 0 {
    err = oserror();
    if (err == AL_BAD_NO_PORTS) {
        fprintf(stderr, " System is out of audio ports\n");
    } else if (err == AL_BAD_DEVICE_ACCESS) {
        fprintf(stderr, " Couldn't access audio device\n");
    } else if (err == AL_BAD_OUT_OF_MEM) {
        fprintf(stderr, " Out of memory: port open failed\n");
    }
    exit(1);
...
alClosePort(input_port);
alClosePort(output_port);
```

## Reading and Writing Audio Data

This section explains how an audio application reads and writes audio sample frames to and from ALports.

Audio input is accomplished by reading audio data sample frames from an input ALport's sample queue. Similarly, audio output is accomplished by writing audio data sample frames to an output ALport's sample queue.

**alReadFrames()** and **alWriteFrames()** provide mechanisms for transferring audio sample frames to and from sample queues. They are *blocking* routines, which means that a program will halt execution within the **alReadFrames()** or **alWriteFrames()** call until the request to read or write sample frames can be completed.

### Reading Sample Frames from an Input ALport

**alReadFrames()** reads a specified number of sample frames from an input port to a sample data buffer, blocking until the requested number of sample frames have been read from the port. Its function prototype is:

```
int alReadFrames ( const ALport port, void *samples, const int framecount )
```

where:

*port* is an audio port configured for input

*samples* is a pointer to a buffer into which you want to transfer the sample frames read from input. *samples* is treated as one of the following types, depending on the configuration of the ALport:

| | |
|---|---|
| char * | for integer sample frames of width AL_SAMPLE_8 |
| short * | for integer sample frames of width AL_SAMPLE_16 |
| long * | for integer sample frames of width AL_SAMPLE_24 |
| float * | for floating point sample frames |
| double * | for double-precision floating point sample frames |

*framecount* is the number of sample frames to read

To prevent blocking, *samplecount* must be less than the return value of **alGetFilled()**.

**Note:** When the application is reading sample frames into an ALport that has *channels* set to 4, *samplecount* must be an integer multiple of the frame size, or an error will be returned and no sample frames will be transferred.

When 4-channel data is input on systems that do not support 4 line-level electrical connections, that is, when setting AL_CHANNEL_MODE to AL_4CHANNEL is not possible, **alReadFrames()** will provide 4 sample frames per frame, but the second pair of sample frames will be set to 0.

Table 4-4 shows the input conversions that are applied when reading mono, stereo, and 4-channel input in stereo mode (default) and in 4-channel mode hardware configurations. Each entry in the table represents a sample frame.

**Table 4-4** Input Conversions for **alReadFrames()**

| | Hardware Configuration | |
|---|---|---|
| **Input** | **Indigo, and Indigo$^2$ or Indy in Stereo Mode** | **Indigo$^2$ or Indy in 4-channel Mode** |
| Frame at physical inputs | ($L_1$, $R_1$) | ($L_1$, $R_1$, $L_2$, $R_2$) |
| Frame as read by a mono port | ($L_1$ + $R_1$) / 2 | (Clip ($L_1$ + $L_2$), Clip ($R_1$ + $R_2$)) / 2 |
| Frame as read by a stereo port | ($L_1$, $R_1$) | (Clip ($L_1$ + $L_2$), Clip ($R_1$ + $R_2$)) |
| Frame as read by a 4-channel port | ($L_1$, $R_1$, 0, 0) | ($L_1$, $R_1$, $L_2$, $R_2$) |

**Note:** If the summed signal is greater than the maximum allowed by the audio system, it is clipped (limited) to that maximum, as indicated by the Clip function.

### Writing Sample Frames to an Output ALport

Sample frames placed in an output queue are played by the audio hardware after a specific amount of time, which is equal to the number of sample frames that were present in the queue before the new sample frames were written, divided by the (sample rate × number of channels) settings of the ALport.

**alWriteFrames()** writes a specified number of sample frames to an output port from a sample data buffer, blocking until the requested number of sample frames have been written to the port. Its function prototype is:

```
int alWriteFrames ( ALport port, void *samples, long framecount )
```

where:

| | |
|---|---|
| *port* | is an audio port configured for input |
| *samples* | is a pointer to a buffer from which you want to transfer the sample frames to the audio port |
| *framecount* | is the number of sample frames you want to read |

**Note:** When the application is writing sample frames from an ALport that has *channels* set to 4, *samplecount* must be an integer multiple of the frame size, or an error will be returned and no sample frames will be transferred.

Table 4-5 shows the output conversions that are applied when writing mono, stereo, and 4-channel data to stereo mode (default) and 4-channel mode hardware configurations.

**Table 4-5** Output Conversions for **alWriteFrames()**

| | | Hardware Configuration | |
|---|---|---|---|
| **Output** | **Frame as Written into Port** | **Indigo, and Indigo$^2$ or Indy in Stereo Mode** | **Indigo$^2$ or Indy in 4-channel Mode** |
| Mono Port | ($L_1$) | ($L_1$, $L_1$) | ($L_1$, $L_1$, 0, 0) |
| Stereo Port | ($L_1$, $R_1$) | ($L_1$, $R_1$) | ($L_1$, $R_1$, 0, 0) |
| 4-channel Port | ($L_1$, $R_1$, $L_2$, $R_2$) | (Clip ($L_1$ + $L_2$), Clip ($R_1$ + $R_2$)) | ($L_1$, $R_1$, $L_2$, $R_2$) |

*Chapter 5*

# Digital Media Buffers

This chapter describes the Digital Media buffers (DMbuffers) real-time visual data transport facility, which is currently supported on O2 workstations. Time-sensitive visual data is moved through system memory using DMbuffers. This method provides a unified approach to facilitating data flow between live video devices. DMbuffers provide a pipelined I/O model—the application can direct the flow of multiple images on multiple paths simultaneously. This chapter builds on video I/O concepts presented in Chapter 4, "Digital Media I/O."

## About Digital Media Buffers

DMbuffers feature

- an operating system-generic live image data storage and transport facility
- a library-transparent interface that allows real-time data interchange with compression/decompression (dmIC) and OpenGL
- an application-centered nonblocking I/O method that provides event processing by means of a single **select()** loop
- a software-configurable memory allocation method that provides the dedicated memory and throughput resources to suit visual data transport requirements using general-purpose system memory

Because the bandwidth of digital video signals far exceeds the bandwidth of typical data storage devices and communications links, video is usually compressed when storing to disk or sending over a network. This software interface supports specialized multimedia hardware on O2 workstations which further boosts visual data processing performance.

DMbuffers and DMbufferpools provide the method for allowing your application to allocate and use general-purpose system memory for transporting visual data.

A DMbufferpool is a custom storage facility created by the application. Video I/O devices, compression devices and algorithms, and graphics devices have direct access to

this storage on a compartmental basis called a DMbuffer. The application can define what this compartment represents, but in general, a DMbuffer represents an image. An image can be in the form of raw uncompressed pixel data for a video field or frame, or in the form of a picture's worth of compressed data (JPEG, MPEG, or so on).

When creating the storage facility, the application describes its data requirements in a DMparams structure. The application also queries every device that plans on using the facility about their needs, and obtains from them a DMparams list describing their requirements for allocating and apportioning memory. Using this information, the storage facility(DMbufferpool) is configured with the proper number, size, and type compartments (DMbuffers) sufficient for containing the data and its associated bookkeeping information.

Because a DMbuffer is created according to all necessary data and device requirements, and with universal content descriptors, every device that needs access to it can tell what's stored inside and can share and use the contents without making modifications.

To transfer the contents of a compartment, it is much easier and faster to transport only a pointer to the storage location rather than moving the actual contents. DMbuffers are really placeholders which contain only the pointers to the actual data; the data itself is stored elsewhere. When data is transferred, only pointers describing the data are passed, not the actual data. This means that the process doesn't have to handle or copy the data in order to transport it.

DMbufferpools contain a fixed number of DMbuffers, all the same size. It's up to you to decide how many DMbuffers to use, but, in general, three DMbuffers are sufficient for most applications: one to receive data, one to send data, and one available in case of contention between the sending and receiving processes.

DMbuffers give the application access to memory which is

- reserved at start-up and guaranteed for the life of the application
- not visited by the page daemon and therefore can't be swapped

When an image needs to be transported, a DMbuffer is allocated from the pool. Once allocated, a DMbuffer can go to a video I/O device, an image converter, or graphics I/O in any order, regardless of the order of allocation. Multiple DMbuffers with independent agendas can coexist. When the transfer is complete, a DMbuffer can be returned to the pool for reuse or retained for future use. It is not necessary to keep the same DMbuffer waiting to complete processing before accepting more input because another DMbuffer can be allocated from the pool.

Compared with the traditional VLbuffers, DMbuffers

- use the same event mechanism to deliver input and output buffer events to the application, rather than using different file descriptors for input and output
- deliver data buffers ordered with VL events on the video input path
- allow buffers to be sent to video multiple times, held by the application, or reordered

Currently, the DM buffers method is supported by:

- Image Converter Library (dmIC)
- Video Library
- Movie Library
- OpenGL

DMbuffers have the following attributes:

- Type
- Size
- MSC / UST

## DMbuffer Live Data Transport Paths

This section presents the framework for eight types of live data transport paths that can be realized using DMbuffers:

- Memory to Video
- Video to Memory
- Memory to Image Converter
- Image Converter to Memory
- Memory to Movie File
- Movie File to Memory
- Memory to OpenGL
- OpenGL to Memory

Figure 5-1 shows the eight live data transport paths that can use DMbuffers.

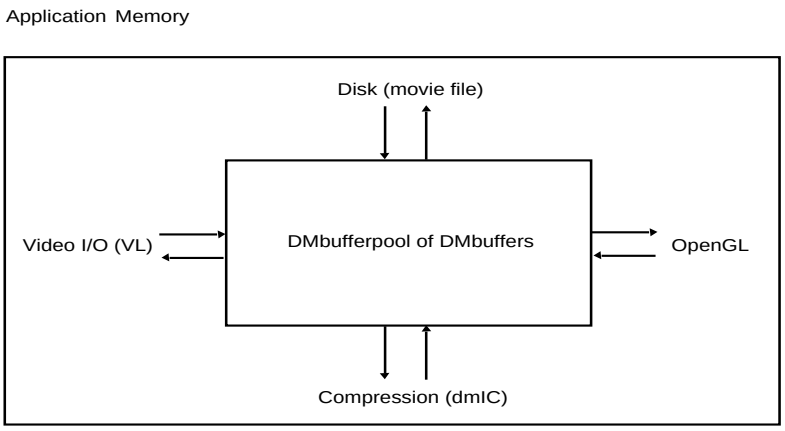



**Figure 5-1** DMbuffer Live Data Transport Paths

Each path can be used as a stage in a more complex path. For example:

video-to-memory—>memory-to-dmIC—>dmIC-to-memory—>memory-to-movie file

are the 4 paths involved in recording compressed live video to disk.

In general, the calls outlined in this section simply move data around; they do not directly encode, decode, or otherwise process the images. This means that data is typically passed untouched along these pathways.

**Note:** The outlines presented in this section are intended to provide a sketch of the calls necessary for each path; the sequence of the calls may vary within each step and the actual coding may involve more routines. See the reference pages for details about using these routines.

## Memory to Video

This section presents a basic sketch for memory to video output.

1. Open video output by calling:

   **vlOpenVideo(3dm)**, to open the video server

   **vlGetNode(3dm)**, to get the video drain node (VL_DRN, VL_VIDEO)

   **vlGetNode(3dm)**, to get the memory source node (VL_SRC, VL_MEM)

2. Create a DMbufferpool for output of video fields or frames by calling:

   **dmBufferSetPoolDefaults(3dm)**, to initialize the DMparams list

   **vlDMPoolGetParams(3dm)**, to determine the video I/O device requirements

   **dmBufferCreatePool(3dm)**, to create a pool using list modified by previous query

3. Associate the DMbufferpool with this path by calling:

   **vlDMPoolRegister(3dm)**

4. Prepare the pool to use **select()** to be notified of a free DMbuffer by calling:

   **vlGetTransferSize(3dm)**, to get the size of one field or frame in this path

   **dmBufferSetPoolSelectSize(3dm)**, to set the memory available threshold for waking from select

   **dmBufferGetPoolFD(3dm)**, to get the input file descriptor for the buffer pool

5. Get a video output path file descriptor (for notification of errors/drops) by calling:

   **vlGetFD(3dm)**

6. Start the video flow by calling:

   **vlBeginTransfer(3dm)**

7. Get notification of a free DMbuffer in the output pool by calling:

   **select(2)**

8. Fill the DMbuffer with pixel data from the desired DMbuffer path

9. Enqueue the DMbuffer for video output by calling:

   **vlDMBufferSend(3dm)**

## Video to Memory

This section presents a basic sketch for video input to memory using DMbuffers:

1. Open video input by calling:

   **vlOpenVideo(3dm)**, to open the video server

   **vlGetNode(3dm)**, to get the video source node (VL_SRC, VL_VIDEO)

   **vlGetNode(3dm)**, to get the memory drain node (VL_DRN, VL_MEM)

2. Create a DMbufferpool for input of video fields or frames by calling:

   **dmBufferSetPoolDefaults(3dm)**, to initialize the DMparams list

   **vlDMPoolGetParams(3dm)**, to determine the video I/O device requirements

   **dmBufferCreatePool(3dm)**, to create a pool with a list modified by previous query

3. Associate the DMbufferpool with this path by calling:

   **vlDMPoolRegister(3dm)**

4. Get a video input file descriptor (for use with **select(2)**) by calling:

   **vlPathGetFD(3dm)**

5. Start the video flow by calling:

   **vlSelectEvents(3dm)**, to select events, namely *VLTransferCompleteMask*, and *VLSequenceLostMask* at a minimum, and others that the process might require

   **vlBeginTransfer(3dm)**, to initiate video transfer

6. Get notification of each new field or frame by calling:

   **select(2)**, to wait for an event

7. Get a new field or frame in the form of a DMbuffer by calling:

   **vlEventRecv(3dm)**, to dequeue the next VLEvent

   **vlEventToDMBuffer(3dm)**, to convert the VLEvent into a DMbuffer

### Memory to Image Converter

1. Find the appropriate image converter by calling:

   **dmICGetNum(3dm)**, to get the number of image converters installed in the system

   **dmICGetDescription(3dm)**, to get the description of a given converter, then search them all (using the total returned) to find the index of the one that performs the desired conversion

2. Create the image converter context / instance by calling:

   **dmICCreate(3dm)**

3. Configure the image converter by calling:

   **dmSetImageDefaults(3dm)**, to initialize the DMparams list with image defaults

   **dmICSetSrcParams(3dm)**, to configure the source (input) image parameters

   **dmICSetDstParams(3dm)**, to configure the destination (output) image parameters

   **dmICSetConvParams(3dm)**, to configure the conversion algorithm (quality, etc.)

4. Create a DMbufferpool for the image converter input by calling:

   **dmBufferSetPoolDefaults(3dm)**, to initialize the DMparams list

   **dmICGetSrcPoolParams(3dm)**, to modify the list to reflect the image converter's buffering requirements (other DMbuffer paths using this pool must also be queried for their requirements before creating the buffer pool)

   **dmBufferCreatePool(3dm)**, to create a buffer pool with the required attributes

5. Prepare the pool to use **select()** to be notified of a free DMbuffer by calling:

   **dmBufferSetPoolSelectSize(3dm)**, to set the memory available threshold for waking from select

   **dmBufferGetPoolFD(3dm)**, to get the buffer pool file descriptor

6. Get notification of a free DMbuffer in the input pool by calling:

   **select(2)**, to wait for a free DMbuffer

   **dmBufferAllocate(3dm)**, to allocate a DMbuffer

7. Fill the DMbuffer with pixel data (if encoding) or bits (if decoding) by calling:

   **dmBufferMapData(3dm)**, to get a pointer to the data area of the DMbuffer

**dmBufferSetSize(3dm)**, to set the image size, which is possibly obtained from one of: **dmImageFrameSize(3dm)**, **vlTransferSize(3dm)**, **mvGetTrackDataInfo(3dm)**, or **mvGetTrackDataFieldInfo(3dm)**

**dmBufferSetUSTMSCpair(3dm)**, to set the sequence number of this image

8. Enqueue the DMbuffer for input to the converter by calling:

   **dmICSend(3dm)**

### Image Converter to Memory

1. Find the appropriate image converter by calling:

   **dmICGetNum(3dm)**, to get the number of image converters installed in the system

   **dmICGetDescription(3dm)**, to get the description of a given converter, then search them all (using the total returned) to find the index of the one that performs the desired conversion

2. Create the image converter context / instance by calling:

   **dmICCreate(3dm)**

3. Configure the image converter by calling:

   **dmSetImageDefaults(3dm)**, to initialize the DMparams list with image defaults

   **dmICSetSrcParams(3dm)**, to configure the source (input) image parameters

   **dmICSetDstParams(3dm)**, to configure the destination (output) image parameters

   **dmICSetConvParams(3dm)**, to configure the conversion algorithm (quality, etc.)

4. Create a DMbufferpool for the image converter output by calling:

   **dmBufferSetPoolDefaults(3dm)**, to initialize the DMparams list

   **dmICGetDstPoolParams(3dm)**, to modify the list to reflect the destination parameters (other DMbuffer paths using this pool must also be queried for their requirements before creating the buffer pool)

   **dmBufferCreatePool(3dm)**, to create a pool with the required attributes

   **dmICSetDstPool(3dm)**, to attach this pool to the image converter

5. Get a converter file descriptor for notification of new output from the image converter by calling:

   **dmICGetDstQueueFD(3dm)**

6. Get notification of a new DMbuffer of converted data by calling:

   **select(2)**

7. Get the new DMbuffer by calling:

   **dmICReceive(3dm)**, to dequeue the converted image from the converter

   **dmBufferGetUSTMSCpair(3dm)**, to get the sequence number of this image

   **dmBufferGetSize(3dm)**, to get the size of the image data in bytes

   **dmBufferMapData(3dm)**, to get a pointer to the image data

### Memory to Movie File

1. Open or create a movie file with an image track using one of the following methods:
   - Create a new movie file by calling:

     **mvCreateFile(3dm)**, to get a handle to the new movie file

     **mvAddTrack(3dm)**, to insert an empty image track into the movie file
   - Open an existing movie file by calling:

     **mvOpenFile(3dm)**, to get a handle to the existing movie

     **mvFindTrackByMedium(3dm)**, to find the movie's image (DM_IMAGE) track

     **mvGetTrackLength(3dm)**, to find the playing time of the current track
2. Determine the image size by calling:

   **dmBufferMapData(3dm)**, to get a pointer to the image

   **dmBufferGetSize(3dm)**, to get the size of the image in bytes

   **Note:** Perform these operations twice for field-based data (once per field)
3. Save the image to the movie file using one of the following methods (pass the pointer and the size returned in the previous step to the function):
   - Write the data from a single DMbuffer as a frame by calling:

     **mvInsertTrackData(3dm)**
   - Write the data from 2 DMbuffers (one per field) by calling:

     **mvInsertTrackDataFields(3dm)**

### Movie File to Memory

1. Open a movie file for reading by calling:

   **mvOpenFile(3dm)**

   **mvFindTrackByMedium(3dm)**, to find the DM_IMAGE track

   **mvGetImageWidth(3dm)**, to get the width of the image

   **mvGetImageHeight(3dm)**, to get the height of the image

   **mvGetTrackMaxFieldSize(3dm)**, to get the size (in bytes) of the largest compressed field, which can be used for configuring DMbuffer size when creating a buffer pool

2. Get a free DMbuffer by calling:

   **dmBufferAllocate(3dm)**, to allocate a DMbuffer

   **dmBufferMapData(3dm)**, to get a pointer to the data area of the DMbuffer

3. Get the movie file *index* for the desired image by calling:

   **mvGetTrackDataIndexAtTime(3dm)**

4. Import the compressed image data into memory using one of the following methods:

   - If image data is stored as a frame, call:

     **mvGetTrackDataInfo(3dm)**, to get the size of the compressed data

     **mvReadTrackData(3dm)**, to copy the compressed data from the movie file directly into the DMbuffer

   - If image data is stored as fields, call:

     **mvGetTrackDataFieldInfo(3dm)**, to get the sizes of the compressed fields

     **mvReadTrackDataFields(3dm)**, to copy the 2 compressed fields directly into 2 DMbuffers

5. Set the DMbuffer size(s) by calling:

   **dmBufferSetSize(3dm)**

## Memory to OpenGL

This section presents a sketch of transporting video images to a window on the graphics display.

1. Get a pointer to the image data in a DMbuffer by calling:

   **dmBufferMapData(3dm)**

2. Display image in a graphics context (window or offscreen buffer) by calling:

   **glPixelZoom**(*zoomx*, *-zoomy*), to set a negative y zoom indicating top-to-bottom orientation (needed for video)

   **glDrawPixels()**, using

   GL_YCRCB_422_SGIX, (to convert from YCrCb to RGB)

   GL_INTERLACE_SGIX (as supported— an OpenGL interlacing extension for sending two noninterlaced fields of video to an interlaced graphics display by automatically interlacing alternate lines from two video fields)

## OpenGL to Memory

The OpenGL can render to offscreen memory (pbuffer) which can be accessed directly as a DMbuffer.

1. Create an OpenGL/X pbuffer by calling:

   **glXCreateGLXPbufferSGIX()**

2. Relate the two buffers by calling:

   **glXMakeCurrent()**

   **glXAssociateDMPbufferSGIX()** with *pbuffer*, *dmbuffer*

## A Detailed Look at Recording Compressed Live Video to Disk

This section explains how to get video into memory using DMbuffers. Refer to the sample program *dmplay.dmic* in Video Capture with Compression in Chapter 4 of "Digital Media Programmer's Examples" for a demonstration of this method.

Figure 5-2 shows the video compression path.

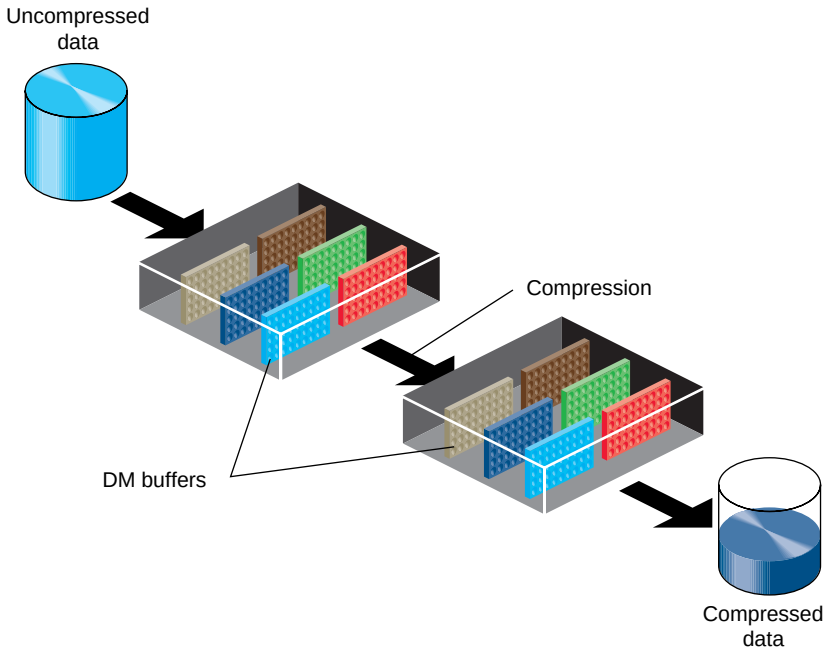



**Figure 5-2** Compression Path Using DMbuffers

The first step in video I/O is getting the video data into a DMbuffer. Once you have a buffer of video data, you can send it to dmIC to compress it, send it to graphics for display, applying effects along the way, or write it to a movie file. You can create separate threads for the video input and image conversion to ensure that the processing remains event-driven. In addition, the DMbuffers method provides the capability of performing multiple image conversion operations simultaneously within the same application with no performance penalty.

Getting video into memory requires

- a video source (input) node
- a memory drain (output) node
- a path that connects the two nodes
- an video input DMbufferpool associated with the memory drain node

This method of compressing video into memory also uses

- an input DMbufferpool on the input to the DM image converter (this is the DM buffer pool associated with the memory drain node)
- an output DMbufferpool to contain the output from the converter

The first three items are the same regardless of whether you use a DMbuffer or a VLbuffer to transport visual data, and are explained in Chapter 4.

The basic model for video input to memory using DMbuffers is:

1. Initialize the DMparams list by calling **dmBufferSetPoolDefaults(3dm).**
2. Query each device about its requirements for memory allocation:
   - Determine video I/O device requirements using **vlDMPoolGetParams(3dm)**
   - Determine image converter requirements using **dmICGetSrcPoolParams(3dm)** and **dmICGetDstPoolParams(3dm)** as described in "3. Creating Data Buffers Using the Image Conversion API" in Chapter 6.
3. Create buffer pools using the DMParams lists returned by the previous queries, by calling **dmBufferCreatePool(3dm)**.
4. Register the input buffer pool as the video input destination (drain) by calling **vlDMPoolRegister(3dm)**.

This application can be divided into the following major areas (each of which is further broken down into detailed steps):

- Video initialization, which involves
  - opening a connection to the video server (**vlOpenVideo()**)
  - if necessary, determining which device the application will use (**vlGetDevice()**, **vlGetDeviceList()**)
  - getting input information, such as the size of the video images

**Note:** Image size (in pixels) can be determined from the video timing and packing. Use **dmImageGetSize()** to determine the size of an image in bytes. The *dmplay.dmic* program first sets the capture mode to noninterlaced to get the size of each field, and then resets the capture mode to interlaced before initiating video transfer.

- Creating and setting up a video data transfer path, which involves
  - specifying nodes on the data path (**vlGetNode()**)
  - creating a path connecting the specified nodes (**vlCreatePath()**)
  - setting up the hardware for the path (**vlSetupPaths()**)
- Getting and setting controls for video data transfer, which involves
  - getting input parameters (controls) for the nodes on the path (**vlGetControl()**)
  - setting output parameters (controls) for the nodes on the path (**vlSetControl()**)
- Creating and registering DMbufferpools to handle video data, which involves
  - initializing the DMparams list by calling **dmBufferSetPoolDefaults()**
  - querying each device about its requirements for memory allocation:

    Determine video I/O device requirements using **vlDMPoolGetParams().**

    Determine image converter requirements using **dmICGetSrcPoolParams()** and **dmICGetDstPoolParams()** as described in "3. Creating Data Buffers Using the Image Conversion API" in Chapter 6.
  - creating buffer pools using the DMParams lists modified by the previous queries, by calling **dmBufferCreatePool()**
  - registering the input buffer pool as the video input destination (drain) by calling **vlDMPoolRegister()**
- Transfer initiation, which involves:
  - setting the video transfer mode using VL_CAP_TYPE
  - specifying path-related events to be captured by calling **vlSelectEvents()**
  - obtaining a VL file descriptor to wait upon by calling **vlGetFD()**
  - initiating the video transfer by calling **vlBeginTransfer()**

- Main loop, which involves
  - waiting for video events using **select()**
  - dequeueing the next video event using **vlEventReceive()**
  - obtaining the video field or frame as a DMbuffer for further transport to the appropriate destination (dmIC, OpenGL, etc.) using **vlEventtoDMbuffer()**
- Cleanup, which involves
  - freeing the DMBuffers and destroying the DMbufferpools

*Chapter 6*

# Digital Media Data Conversion

The file input/output routines of digital media libraries, such as the Audio File Library and the Movie Library, use digital media converters to provide automatic data format conversion. This chapter describes how to use the digital media conversion libraries to create and use these converters in your application.

## About Digital Media Data Conversion

Digital media *data conversion* includes compression and decompression based on industry standards, such as JPEG and MPEG-1. It also encompasses transforming image color spaces, changing audio sample rates, and other basic data modifications. There are two generalized *conversion libraries* to effect digital media data conversions: the Image Conversion Library for video data, and the Audio Conversion Library for audio data. Applications can access these libraries to convert streams of digital media data. (Because they incur some overhead, the conversion libraries usually are not used for small amounts of data like single still images.)

The conversion library APIs provide an interface to *codecs* (*co*mpressor-*dec*ompressors) such as JPEG or MPEG-1, to the Color Space Library and to other transformation libraries. The codecs and transformation libraries do the actual data conversions. Codecs may be either software modules or hardware devices with software interfaces. The hardware codecs are faster, but the software codecs may offer more options. If your application chooses a software codec in lieu of a hardware one, you may want to have your application notify the user of this fact. For more information about specific codecs see Chapter 2, “Digital Media Essentials.”

The digital media conversion libraries offer a number of benefits:

- They permit memory-to-memory data format conversions. These are more flexible than conversions between memory and I/O devices because the converted data is retained in memory for reuse.
- They integrate the use of software codecs, hardware codecs, and transformation libraries.

- They enable applications to perform conversions by merely specifying three sets of parameters:
  - one that describes the source data format
  - one that describes the destination data format
  - one that gives codec-specific settings
- They enable an application to use a number of codecs simultaneously.
- They have a modular design that allows codecs to be loaded dynamically as the application needs them.
- They allow applications to obtain a codec's parameters and default values, and to change these values appropriately.

## Using The Digital Media Converters

A digital media *converter* is the *encoder* (compressor) or *decoder* (decompressor) of a codec. Converters are grouped by the conversion library that uses them. As you might expect, the Image Conversion Library uses the converters for image data, while the Audio Conversion Library uses the converters for audio data. Despite these groupings, converters are used in similar ways.

An application creates a *converter instance* by using the API of either of the conversion libraries. A converter instance includes a converter, any necessary transformation libraries, an input and an output buffer, and state information in the form of parameters that describe the input data, output data, and conversion process. A number of converter instances can use the same converter simultaneously. A converter instance that includes an encoder takes uncompressed data, applies the transformation libraries needed to make the data usable to the encoder, and then uses the encoder to produce the converted data. A converter instance with a decoder reverses this process.

A converter instance can be viewed as a pipeline. On one end of the pipeline is uncompressed data in some specified format. On the other end is data in the format native to the converter. The uncompressed data is the pipeline's input if the converter is a encoder, otherwise it is the output. The pipeline processing is done by the transformation libraries and the converter. When multiple transformation libraries are needed, the conversion libraries make sure that they are used in an order that best maintains the quality of the data.

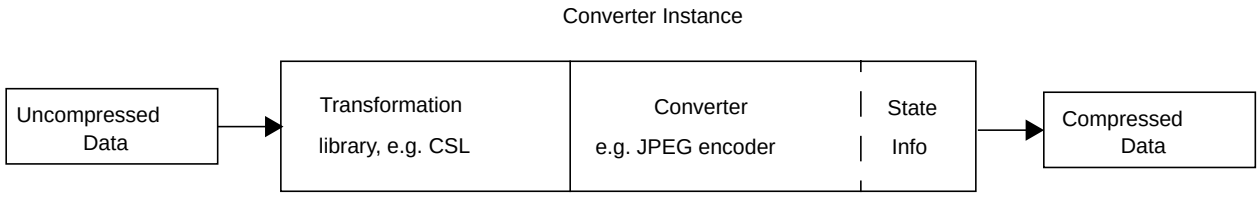



**Figure 6-1** The Conversion Pipeline

To perform a data conversion your application follows these five steps:

1. Create a converter instance
2. Configure the converter instance
3. Create data buffers for the converter instance (image conversion only)
4. Convert the data using the converter instance
5. Destroy the converter instance

To use the digital media conversion libraries to create a converter instance, you must link your application with *libdmedia.so*.

## Image Data Conversion

This section describes the Digital Media Image Conversion Library and its converters. What follows is a discussion of how to use the Image Conversion API to execute the five steps of an image data conversion.

### The Digital Media Image Conversion Library

The API of the Digital Media Image Conversion Library provides an interface for memory-to-memory image conversion that is independent of the algorithm. The Image Conversion Library, see also dmic(4), enables you to create an image converter instance based on a codec's encoder or decoder. Table 6-1 lists commonly installed codecs that can be accessed through the Image Conversion API. As shown in the table, these codecs usually have both types of converters, and may be performed by hardware on properly

equipped systems. The columns “DM_IC_ENGINE Value” and “DM_IC_ID Value” contain identification values that are used with **dmICGetDescription()** as described in the section “1. Creating a Converter Instance Using the Image Conversion API.”

**Table 6-1** Digital Media Image Converters

| Codec Name | DM_IC_ENGINE Values | DM_IC_ID Value |
|---|---|---|
| Apple QuickTime Animation | “The Apple Animation compressor”<br>“The Apple Animation decompressor” | ‘rle ‘ |
| Cinepak | ”The Cinepak compressor”<br>“The Cinepak decompressor” | ‘cvid’ |
| Intel Indeo | “The Indeo Video compressor”<br>“The Indeo Video decompressor” | ‘IV32’ |
| H.261 | “The H261 software encoder”<br>“The H261 software decoder” | ‘h261’ |
| JPEG | “Software JPEG Encoder”<br>“Software JPEG Decoder”<br>“Vice” | ‘jpeg’ |
| MPEG-1 Video | “Vice” | ‘mpeg’ |
| Motion Video Compressor 1 | “The MVC1 compressor”<br>”The MVC1 decompressor” | ‘mvc1’ |
| Motion Video Compressor 2 | ”The MVC2 compressor”<br>”The MVC2 decompressor” | ‘mvc2’ |
| 8-bit Run Length Encode | “The RLE compressor”<br>“The RLE decompressor” | ‘rle1’ |
| 24-bit Run Length Encode | “The RLE24 compressor”<br>“The RLE24 decompressor” | ‘rle2’ |
| Apple QuickTime Video | ”The Apple Video compressor”<br>”The Apple Video decompressor” | rpza’ |

In addition to the operations done by the codecs, an image converter instance may also perform transformations involving

- color space
- size

- clipping
- orientation
- interlacing
- image rate

For some of these, the image converter instance calls the Color Space Library during the conversion process. This library is discussed in "The Digital Media Color Space Library." Also shown is how the Image Conversion Library provides access to it independently of a converter instance.

As the next five sections demonstrate, The Image Conversion Library enables you to effectively use codecs and the Color Space Library by following the five steps listed in "Using The Digital Media Converters." To use the Image Conversion API, you must include these header files:

```
#include <dmedia/dmedia.h>
#include <dmedia/dm_imageconvert.h>
```

### 1. Creating a Converter Instance Using the Image Conversion API

The Image Conversion Library allows you to choose among the image converters in the system. To find a converter and create an instance of it, the Image Conversion Library enables you to do the following:

- Get the number of image converters present in the system with **dmICGetNum()**
- Get the description of a specific converter with **dmICGetDescription()**
- Choose a converter based on your application's conversion needs with **dmICChooseConverter()**
- Create a converter instance with **dmICCreate()**

**Note:** Many of the Image Conversion Library functions return a DMstatus value. The value is DM_SUCCESS if they succeed, DM_FAILURE if not. After a receiving a DM_FAILURE, your application can call the function **dmGetError()** or **dmGetErrorForPID()** to return an error message and error number. See "Digital Media Error Handling" in Chapter 3 for more information.

Use the function **dmICGetNum()** to find the number of image converters available:

```
int dmICGetNum ( void )
```

The function returns an integer that is the number of available image converters.

Use the function **dmICGetDescription()** to obtain the description of a converter:

```
DMstatus dmICGetDescription ( int i, DMparams *converterParams )
```

The parameter *i*, which is the index of the converter, is the key to the description returned. The converter's index is an integer between 0 and *n*-1 where *n* is the number of image converters returned by **dmICGetNum()**. The parameter *converterParams* points to a previously created DMparams data structure. If **dmICGetDescription()** is successful, the data structure contains six parameters whose values characterize the converter.

**Tip:** The application controls the creation and destruction of DMparams structures. See the upcoming section "Using the Image Conversion API" for examples.

To extract the converter's description from the structure pointed to by *converterParams*, you submit parameter keywords to the appropriate DMparams-querying functions. The left side of the following table lists the six parameter keywords and the relevant DMparams-querying functions. On the right side are possible values returned by the querying functions.

| | |
|---|---|
| DM_IC_ID<br>and **dmParamsGetInt()** | The codec's DM_IC_ID value, as shown in Table 6-1 |
| DM_IC_ENGINE<br>and **dmParamsGetString()** | The codec's DM_IC_ENGINE value, as shown in Table 6-1 |
| DM_IC_VERSION<br>and **dmParamsGetInt()** | The codec's version number, for example 1. |
| DM_IC_REVISION<br>and **dmParamsGetInt()** | The codec's revision number, for example 2. |
| DM_IC_CODE_DIRECTION<br>and **dmParamsGetEnum()** | DM_IC_CODE_DIRECTION_UNDEFINED<br>DM_IC_CODE_DIRECTION_ENCODE<br>DM_IC_CODE_DIRECTION_DECODE |
| DM_IC_SPEED<br>and **dmParamsGetEnum()** | DM_IC_SPEED_UNDEFINED<br>DM_IC_SPEED_REALTIME<br>DM_IC_SPEED_NONREALTIME |

**Note:** Although they are not shown here, DMparams-querying functions have corresponding setting functions. For example, **dmParamsSetEnum()** corresponds with **dmParamsGetEnum()**. However, some parameter values, such as those above, cannot be set by your application. See Chapter 3, "Digital Media Data Types and Parameter Lists," for a discussion of using DMparams.

Your application should check the value of DM_IC_CODE_DIRECTION to determine if the codec is an encoder or decoder. The contents of DM_IC_ID and DM_IC_ENGINE may not have enough information to make this decision. If the value of DM_IC_SPEED is DM_IC_SPEED_REALTIME, the codec is suitable for real-time encoding or decoding. Usually, this indicates a hardware codec.

Use the function **dmICChooseConverter()** to choose a converter based on parameter values:

```
int dmICChooseConverter ( DMparams *srcParams, DMparams *dstParams,
                          DMparams *convParams )
```

The function returns the index of the converter that matches the requirements specified by the DMparams structures pointed to by *srcParams*, *dstParams*, and *convParams*. For **dmICChooseConverter()** to succeed, either the DM_IC_CODE_DIRECTION and DM_IC_ID parameters must be set in *convParams*, or the DM_IMAGE_COMPRESSION parameter must be set in *srcParams* and *dstParams*.

This function opens the converter that matches the specified parameters and values. If **dmICChooseConverter()** is successful, *srcParams*, *dstParams*, and *convParams* can be used as inputs to **dmICSetSrcParams()**, **dmICSetDstParams()**, and **dmICSetConvParams()** respectively. (See "2. Configuring a Converter Instance Using the Image Conversion API.") The converter chosen is guaranteed to be optimal only for the given parameters. If the user changes any of the parameter values, your application may need to call **dmICChooseConverter()** again.

Use the function **dmICCreate()** to create an image converter instance:

```
DMstatus dmICCreate ( int i, DMimageconverter *converter )
```

This function opens the converter corresponding to *i*, an index returned by **dmICChooseConverter()**, and initializes *converter* (the address of a previously declared DMimageconverter variable) with the *handle* (the address of a pointer) of a converter instance. The handle is declared as follows:

```
typedef struct _DMimageconverter *DMimageconverter;
```

An application normally may open multiple converter instances based on one or more converters. However, some converters may limit the number of instances based on them. A converter is always used as part of a converter instance.

**Tip:** Because converters can be loaded dynamically, a converter's index can vary from system to system, or even from time to time on the same system. Before using an index, your application should use either **dmICGetDescription()** to check its validity, or **dmICChooseConverter()** to establish its value. Once determined, the value of the index will not change while an application is running.

The code that follows the comment "/ / 1. Creating a converter instance" in the section "Using the Image Conversion API" makes use of some of the above functions.

### 2. Configuring a Converter Instance Using the Image Conversion API

Once a converter instance has been created, it must be configured. The application does this by using DMparams data structures to specify a set of source parameters, a set of destination parameters, and, optionally, a set of conversion parameters. The *source parameters* describe the image data that is the converter instance's input. The input is uncompressed data if the converter is a compressor, otherwise it is in the format native to the converter. The *destination parameters* describe the converter instance's output. The output is in the format native to the converter if the converter is a compressor, otherwise it is uncompressed data. The *conversion parameters* fine tune settings that are specific to the converter.

The Image Conversion Library enables you to do the following:

- Get the source, destination, and conversion parameters, and their default settings, with **dmICGetDefaultSrcParams()**, **dmICGetDefaultDstParams()**, and **dmICGetDefaultConvParams()**
- Get the actual settings of the source, destination, and conversion parameters with **dmICGetSrcParams()**, **dmICGetDstParams()**, and **dmICGetConvParams()**
- Change the settings of the source, destination, and conversion parameters with **dmICSetSrcParams()**, **dmICSetDstParams()**, and **dmICSetConvParams()**

In each of the nine functions described in this section, the first parameter, *converter*, is a DMimageconverter previously initialized by **dmICCreate()**. The second parameter is a pointer to a DMparams structure.

Use the functions **dmICGetDefaultSrcParams()**, **dmICGetDefaultDstParams()**, and **dmICGetDefaultConvParams()** to get the default parameter settings:

```
DMstatus dmICGetDefaultSrcParams ( DMimageconverter converter,
                                   DMparams *srcParams )

DMstatus dmICGetDefaultDstParams ( DMimageconverter converter,
                                   DMparams *dstParams )

DMstatus dmICGetDefaultConvParams ( DMimageconverter converter,
                                    DMparams *convParams )
```

After each of these functions returns, the DMparams structure contains the parameters, along with their default values, that your application can set for the converter instance indicated by *converter* using the setting functions described in this section. Although the conversion parameters, pointed to by *convParams*, vary markedly from converter to converter, those shown below are supported by many. The left side of the table lists each conversion parameter and its DMparams-querying function. Possible values for the parameter are listed on the right. For a more complete discussion of retrieving parameter values, see Chapter 3, “Digital Media Data Types and Parameter Lists.”.

| | |
|---|---|
| DM_IMAGE_BITRATE and **dmParamsGetInt()** | An integer value that indicates the rate, in bits per second, of a compressed video stream. |
| DM_IMAGE_QUALITY_SPATIAL and **dmParamsGetFloat()** | DM_IMAGE_QUALITY_NORMAL DM_IMAGE_QUALITY_LOSSLESS several others. |
| DM_IMAGE_QUALITY_TEMPORAL and **dmParamsGetFloat()** | DM_IMAGE_QUALITY_NORMAL DM_IMAGE_QUALITY_LOSSLESS several others. |

Use the functions **dmICGetSrcParams()**, **dmICGetDstParams()**, and **dmICGetConvParams()** to get the actual parameter settings:

```
DMstatus dmICGetSrcParams ( DMimageconverter converter,
                            DMparams *srcParams )

DMstatus dmICGetDstParams ( DMimageconverter converter,
                            DMparams *dstParams )

DMstatus dmICGetConvParams ( DMimageconverter converter,
                             DMparams *convParams )
```

After these functions return successfully, the DMparams structures contain the parameters and values that describe the input image, the output image, and the conversion settings for the data in the buffer obtained by the last successful call to **dmICReceive()**. The parameters are not defined prior to a successful call to **dmICReceive()**. See the section "4. Converting Data Using the Image Conversion API" for more information on **dmICReceive()**.

Use the functions **dmICSetSrcParams()**, **dmICSetDstParams()**, and **dmICSetConvParams()** to change the parameter values:

```
DMstatus dmICSetSrcParams ( DMimageconverter converter,
                            DMparams *srcParams )

DMstatus dmICSetDstParams ( DMimageconverter converter,
                            DMparams *dstParams )

DMstatus dmICSetConvParams ( DMimageconverter converter,
                             DMparams *convParams )
```

The parameters and values to be used are specified in the DMparams structures pointed to by *srcParams*, *dstParams*, and *convParams*. The settings describe the input image, the output image, and the conversion settings for the data in the buffer that will be sent by the next **dmICSend()** operation. See the section "4. Converting Data Using the Image Conversion API" for more information on **dmICSend()**.

The left side of the following table lists the image conversion parameters that *must* be specified in the source and destination DMparams structures. Also listed are the appropriate DMparams-setting functions. Possible values of the parameters are listed on the right. See *dm_image.h* for all the image parameters, and Chapter 3, "Digital Media Data Types and Parameter Lists" for a complete discussion of them.

| | |
|---|---|
| DM_IMAGE_WIDTH<br>and **dmParamsSetInt()** | Image pixel width, e.g. 640. |
| DM_IMAGE_HEIGHT<br>and **dmParamsSetInt()** | Image pixel height, e.g. 480. |
| DM_IMAGE_PACKING<br>and **dmParamsSetEnum()** | DM_IMAGE_PACKING_RGBX<br>many others. |
| DM_IMAGE_ORIENTATION<br>and **dmParamsSetEnum()** | DM_IMAGE_TOP_TO_BOTTOM<br>DM_IMAGE_BOTTOM_TO_TOP |

The code that follows the comment "/ / 2. Configuring a converter instance" in the section "Using the Image Conversion API" makes use of some of the above functions.

### 3. Creating Data Buffers Using the Image Conversion API

The Image Conversion Library uses a buffering system to transmit data between your application and the converter instance. Typically, a data buffer contains a single video image, either compressed or uncompressed. You use the Image Conversion Library to create an source (input) *buffer pool* and an destination (output) buffer pool. Your application allocates individual buffers from the source buffer pool, and uses them to send data to the converter instance. The converter instance returns processed data to your application with buffers it creates from the destination buffer pool. The buffer pools, which are fixed in size, are created during an application's initialization. Before creating the buffer pools with **dmBufferCreatePool()**, your application must make sure they satisfy the requirements of all the libraries that are going to use them. This last point is explained more fully below with the functions **dmICGetSrcPoolParams()** and **dmICGetDstPoolParams()**.

The Image Conversion Library allows you to determine the buffering requirements of a converter instance prior to creating a buffer pool. To find a converter instance's buffering needs, the Image Conversion Library enables you to do the following:

- Get the source and destination pool requirements with **dmICGetSrcPoolParams()** and **dmICGetDstPoolParams()**
- Select the buffer pool from which to receive data with **dmICSetDstPool()**

Use the functions **dmICGetSrcPoolParams()** and **dmICGetDstPoolParams()** to determine a converter instance's buffering needs:

```
DMstatus dmICGetSrcPoolParams ( DMimageconverter converter,
                                DMparams *poolParams )

DMstatus dmICGetDstPoolParams ( DMimageconverter converter,
                                DMparams *poolParams )
```

The converter instance indicated by *converter* modifies the DMparams structure pointed to by *poolParams* to describe its buffering needs. Your application should also pass this *poolParams* to other libraries that may share the same buffer pool. For example, if the buffer pool must be shared with the Video Library or the Graphics Library, *poolParams* should be passed to **vlDMPoolGetParams()** or **dmBufferGetGLPoolParams()**. This function returns an error if it detects that requirements set in *poolParams* by another library conflict with those of the converter instance.

Use **dmICSetDstPool()** to choose the buffer pool for a converter instance's output:

```
DMstatus dmICSetDstPool ( DMimageconverter converter,
                          DMbufferpool *pool )
```

The variable *pool* points to the destination buffer pool from which the converter instance, *converter*, allocates output buffers. Your application is responsible for creating and disposing of *pool*. The function returns an error if the converter instance does not meet the requirements of *pool*. This function must be called prior to **dmICSend()** or **dmICReceive()**, which are described in the next section.

The code that follows the comment "/ / 3. Creating data buffers for the converter instance" in the section "Using the Image Conversion API" makes use of some of the above functions.

#### 4. Converting Data Using the Image Conversion API

During the actual conversion process, your application sends data buffers to a converter instance, and receives buffers of processed data from it. A converter instance has a source and destination queue to handle buffer traffic. Your application allocates, fills, and sends data buffers to the source queue. The instance attaches the buffers in the queue, that is prevents them from being deleted, until it can process the data. The destination queue, containing data buffers created by the instance from the destination buffer pool selected with **dmICSetDstPool()**, holds the processed data until your application is ready to receive it. The source and destination queues allow the converter instance to interact asynchronously with your application.

The Image Conversion Library enables your application to do the following:

- Send data to and receive it from a converter instance with **dmICSend()** and **dmICReceive()**
- Use a file descriptor to be notified of data arriving from a converter instance with **dmICGetDstQueueFD()**
- Determine the number of buffers that are ready to be received from the converter instance with **dmICGetDstQueueFilled()** and the number that are ready to be processed by the converter instance with **dmICGetSrcQueueFilled()**

Use the function **dmICSend()** to send data to a converter instance for conversion:

```
DMstatus dmICSend ( DMimageconverter converter, DMbuffer srcBuffer,
                    int numRefBuffers, DMbuffer *refBuffers )
```

This function adds the data buffer *srcBuffer* to the input queue of the converter instance *converter*. It is an asynchronous operation, and the conversion may not have taken place when **dmICSend()** returns. However, if it is no longer needed by your application, the buffer can be freed with **dmBufferFree()** immediately after **dmICSend()** returns. The buffer is attached by the converter instance until it can be used, and will not be deleted until the converter instance is finished with it. The integer *numRefBuffers* is the number of reference buffers in an array pointed to by *refBuffers*. The converter instance uses the *reference buffers* to interpret subsequent buffers which are coded in terms of them.

The **dmICSend()** function returns errors if the queue is full, or if any of the parameters are invalid. If the requirements of the source, destination, and conversion parameters set previously cannot be met during the instance's processing, an error is returned on a subsequent call to **dmICReceive()**.

Use the function **dmICReceive()** to receive data from a converter instance:

```
DMstatus dmICReceive ( DMimageconverter converter,
                       DMbuffer *dstBuffer )
```

This function removes a completed data buffer, pointed to by *dstBuffer*, from the output queue of *converter*. The buffer is automatically attached to the caller. When your application no longer needs it, the buffer must be freed with **dmBufferFree()**. The function returns an error of DM_IC_Q_EMPTY if there are no buffers ready.

Use the function **dmICGetDstQueueFD()** to obtain a file descriptor associated with the data from a converter instance:

```
int dmICGetDstQueueFD ( DMimageconverter converter )
```

The function returns the file descriptor associated with the converter instance *converter*. Your application can use this file descriptor with **select()** or **poll()** to be notified when data is available to be retrieved with **dmICReceive()**. The code sample in "Using the Image Conversion API" shows an example of this technique.

Use **dmICGetDstQueueFilled()** to determine how many data buffers have been processed by a converter instance and are now available to your application. Use the function **dmICGetSrcQueueFilled()** to find how many data buffers have been sent to the converter instance, but not processed:

```
int dmICGetDstQueueFilled ( DMimageconverter converter )
int dmICGetSrcQueueFilled ( DMimageconverter converter )
```

The function **dmICGetDstQueueFilled()** returns an integer that is the number of buffers from *converter* that can be removed by **dmICReceive()**. The **dmICGetSrcQueueFilled()** function returns an integer that is the number of buffers that have been sent to *converter* with **dmICSend()** and have yet to be processed. Buffers that are being processed by the converter instance are not counted by either function.

The code following the comment "/ / 4. Converting the data using a converter instance" in the section "Using the Image Conversion API" uses some of the above functions.

#### 5. Destroying a Converter Instance Using the Image Conversion API

The Image Conversion Library enables you to free a converter instance's resources with **dmICDestroy()**:

```
void dmICDestroy ( DMimageconverter converter )
```

This function closes the converter instance indicated by *converter*. The converter instance's internal storage and other resources are freed and *converter* is no longer valid.

The code that follows the comment "/ / 5. Destroying a converter instance" in the next section uses this function.

### Using the Image Conversion API

What follows is a code sample that demonstrates the use of the Image Conversion API.

```
#include <stdio.h>
#include <unistd.h>
#include <dmedia/dmedia.h>
#include <dmedia/dm_image.h>
#include <dmedia/dm_buffer.h>
#include <dmedia/dm_imageconvert.h>

//
// main()
//
int main( int argc, char *argv[] )
{
    int               nNumConverters = 0;
    DMimageconverter  ImgCvt;
    DMparams          *pParams;
    DMbufferpool      InPool, OutPool;
    DMbuffer          InBuffer, OutBuffer;
```

```
    int              i;

// 1. Creating a converter instance

    // Get the number of converters
    nNumConverters = dmICGetNum();
    if (nNumConverters <= 0)  {
        fprintf( stderr, “ICGetNum() = (%d)\n”, nNumConverters );
        return( -1 );
    }

    // Open the first converter
    if (dmICCreate( 0, &ImgCvt ) != DM_SUCCESS)  {
        fprintf( stderr, “ICCreate() failed.\n” );
        return( -1 );
    }

    // Get description of first converter and print it
    dmParamsCreate( &pParams );
    if (dmICGetDescription( 0, pParams ) != DM_SUCCESS)  {
        fprintf( stderr, “GetDescription() Failed\n” );
        return( -1 );
    }
    PrintDescription( pParams );
    dmParamsDestroy( pParams );

// 2. Configuring a converter instance

    // Set SrcParams and print
    dmParamsCreate( &pParams );
    dmSetImageDefaults( pParams, 320, 240, DM_IMAGE_PACKING_RGBX );
    if (dmICSetSrcParams( ImgCvt, pParams ) != DM_SUCCESS)  {
        fprintf( stderr, “SetSrcParams() Failed\n” );
        return( -1 );
    }
    if (dmICGetSrcParams( ImgCvt, pParams ) != DM_SUCCESS)  {
        fprintf( stderr, “GetSrcParams() Failed\n” );
    }
    PrintImageParams( pParams, “Src” );
    dmParamsDestroy( pParams );

    // Set DstParams and print
    dmParamsCreate( &pParams );
    dmSetImageDefaults( pParams, 160, 120, DM_IMAGE_PACKING_RGBX );
    if (dmICSetDstParams( ImgCvt, pParams ) != DM_SUCCESS)  {
```

```
        fprintf( stderr, “SetDstParams() Failed\n” );
        return( -1 );
    }
    if (dmICGetDstParams( ImgCvt, pParams ) != DM_SUCCESS)  {
        fprintf( stderr, “GetDstParams() Failed\n” );
    }
    PrintImageParams( pParams, “Dst” );
    dmParamsDestroy( pParams );

// 3. Creating data buffers for  the converter instance

    // Create Pool
    dmParamsCreate( &pParams );
    dmBufferSetPoolDefaults( pParams, 5, 1024, DM_TRUE, DM_FALSE );
    if (dmBufferCreatePool( pParams, &InPool ) != DM_SUCCESS)  {
        fprintf( stderr, “Create Input Pool Failed\n” );
        return( -1 );
    }
    if (dmBufferCreatePool( pParams, &OutPool ) != DM_SUCCESS)  {
        fprintf( stderr, “Create Output Pool Failed\n” );
        return( -1 );
    }
    dmParamsDestroy( pParams );

    if (dmICSetDstPool( ImgCvt, OutPool ) != DM_SUCCESS)  {
        fprintf( stderr, “SetDstPool() Failed\n” );
        return( -1 );
    }

// 4. Converting data using  the converter instance

    // We loop twice to test whether the codec does a coredump
    // when no src/dst/conv params are sent.
    // A real application would not have this arbitrary cut off.
    for ( i = 0 ; i < 2; ++i )  {
        dmBufferAllocate( InPool, &InBuffer );

        if (dmICSend( ImgCvt, InBuffer, 0, NULL ) != DM_SUCCESS)  {
            fprintf( stderr, “ICSend() Failed\n” );
            return( -1 );
        }

        if (dmBufferFree( InBuffer ) != DM_SUCCESS)  {
            fprintf( stderr, “dmBufferFree() Failed\n” );
            return( -1 );
```

```
        }

        {
            int     FD;
            fd_set  fdset;

            FD = dmICGetDstQueueFD( ImgCvt );
            FD_ZERO( &fdset );
            FD_SET( FD, &fdset );
            fprintf( stderr, "Waiting on FD %d\n", FD );
            select( FD+1, &fdset, NULL, NULL, NULL );
        }
        if (dmICReceive( ImgCvt, &OutBuffer ) != DM_SUCCESS)  {
           fprintf( stderr, "ICReceive() Failed\n" );
           return( -1 );
        }
        // OutBuffer can now be sent anywhere a DMbuffer is accepted.
        // We just free it here.
        if (dmBufferFree( OutBuffer ) != DM_SUCCESS)  {
           fprintf( stderr, "dmBufferFree() Failed\n" );
           return( -1 );
        }
    }

// 5. Destroying a converter instance

    // Close converter
    dmICDestroy( ImgCvt );
    return( 0 );
}

//
// PrintDescription()
//   Prints the description obtained by dmICGetDescription().
//
void PrintDescription( DMparams *pParams )
{
    fprintf( stderr, "Name: %s\n",
             dmParamsGetString( pParams, DM_IC_ENGINE ) );
    switch( dmParamsGetEnum( pParams, DM_IC_SPEED ) )  {
        case DM_IC_SPEED_UNDEFINED:
            fprintf( stderr, "Speed Undefined\n" );
            break;
        case DM_IC_SPEED_REALTIME:
            fprintf( stderr, "Speed Real Time\n" );
```

```
            break;
        case DM_IC_SPEED_NONREALTIME:
            fprintf( stderr, “Speed Not Real Time\n” );
            break;
        default:
            fprintf( stderr, “Speed Unknown\n” );
    }
    switch( dmParamsGetEnum( pParams, DM_IC_CODE_DIRECTION ) )  {
        case DM_IC_CODE_DIRECTION_UNDEFINED:
            fprintf( stderr, “Code Direction Undefined\n” );
            break;
        case DM_IC_CODE_DIRECTION_ENCODE:
            fprintf( stderr, “Code Direction Encode\n” );
            break;
        case DM_IC_CODE_DIRECTION_DECODE:
            fprintf( stderr, “Code Direction Decode\n” );
            break;
        default:
            fprintf( stderr, “Code Direction Unknown\n” );
    }
    fprintf( stderr, “Version: %d\n”,
             dmParamsGetInt( pParams, DM_IC_VERSION ) );
    fprintf( stderr, “Revision: %d\n”,
             dmParamsGetInt( pParams, DM_IC_REVISION ) );
}

//
// PrintImageParams()
//   Prints the descriptions obtained by dmICGetxxxParams().
//
void PrintImageParams( DMparams *pParams, const char *pOrigin )
{
    fprintf( stderr, “Width: %d   Height: %d\n”,
             dmParamsGetInt( pParams, DM_IMAGE_WIDTH ),
             dmParamsGetInt( pParams, DM_IMAGE_HEIGHT ) );
    switch( dmParamsGetEnum( pParams, DM_IMAGE_PACKING ) )  {
        case DM_IMAGE_PACKING_RGBX:
            fprintf( stderr, “RGBX Packing\n” );
            break;
        case DM_IMAGE_PACKING_XBGR:
            fprintf( stderr, “XBGR Packing\n” );
            break;
        case DM_IMAGE_PACKING_RGBA:
            fprintf( stderr, “RGBA Packing\n” );
            break;
```

```
        case DM_IMAGE_PACKING_ABGR:
            fprintf( stderr, “ABGR Packing\n” );
            break;
        default:
            fprintf( stderr, “Unknown Packing\n” );
    }
}
```

## The Digital Media Color Space Library

The Digital Media Color Space Library is a lower level library that is called by the Image Conversion Library. It provides the support for many of the parameter keyword operations discussed in “2. Configuring a Converter Instance Using the Image Conversion API,”such as DM_IMAGE_WIDTH, DM_IMAGE_HEIGHT, DM_IMAGE_PACKING, DM_IMAGE_ORDER, DM_IMAGE_ORIENTATION, and DM_IMAGE_MIRROR. More explicitly, the Color Space Library enables your application to do the following

- Convert between these color spaces: RGB, YCrCb, and Y
- Convert between many packings, such as DM_IMAGE_PACKING_RGB, DM_IMAGE_PACKING_CbYCr, and DM_IMAGE_PACKING_LUMINANCE
- Convert between many data types, such as DM_IMAGE_DATATYPE_BIT, DM_IMAGE_DATATYPE_CHAR, and DM_IMAGE_DATATYPE_SHORT10L
- Provide gamma correction for the RGB components of source and destination data
- Adjust the hue, saturation, brightness, contrast, bias, and scale of individual components
- Enable colorimetry adjustments of specific monitors

Normally, your application does not need to call the Color Space Library directly. The Image Conversion Library calls it as determined by the values in the source, destination, and conversion parameters of your converter instance. However, to enable easy access to the Color Space Library for conversions involving only uncompressed data, the Image Conversion Library provides the convenience function **dmICAnyToAny()**.

Use **dmICAnyToAny()** to make color space conversions on uncompressed data:

```
DMstatus dmICAnyToAny ( void *pBufferSrc, void *pBufferDst,
                        DMparams *pParamsSrc, DMparams *pParamsDst,
                        DMparams *pParamsConv )
```

The two variables *pBufferSrc* and *pBufferDst*, are pointers to the source and destination data which is uncompressed. Under some circumstances, *pBufferSrc* and *pBufferDst*, can be the same, allowing in-place data conversions. See dmColor(3dm) for more information. The variables *pParamsSrc*, *pParamsDst*, and *pParamsConv*, are pointers to DMparams structures which are set as shown in "Using the Image Conversion API." For example, the next sample changes the packing of a 640-by-480 pixel image from CbYCrY to XBGR:

```
/* Points to the source and destination buffers. */
void        *pBufferSrc, *pBufferDst;

/* Points to the source and destination parameters. */
DMparams    *pParamsSrc, pParamsDst;

/* Allocate buffers and fill source buffer. */
...
/* Do other preliminary processing */
...

/* Set the source parameters. */
dmParamsCreate( &pParamsSrc );
dmParamsSetInt( pParamsSrc,  DM_IMAGE_WIDTH, 640 );
dmParamsSetInt( pParamsSrc,  DM_IMAGE_HEIGHT,480 );
dmParamsSetEnum( pParamsSrc, DM_IMAGE_PACKING,
                             DM_IMAGE_PACKING_CbYCrY);

/* Set the destination parameters. */
dmParamsCreate( &pParamsDst );
dmParamsSetInt( pParamsDst,  DM_IMAGE_WIDTH, 640 );
dmParamsSetInt( pParamsDst,  DM_IMAGE_HEIGHT,480 );
dmParamsSetEnum( pParamsDst, DM_IMAGE_PACKING,
                             DM_IMAGE_PACKING_XBGR);

/* Do the conversion and clean up. */
dmICAnyToAny( pBufferSrc, pBufferDst,
              pParamsSrc, pParamsDst, NULL );
dmParamsDestroy( pParamsSrc );
dmParamsDestroy( pParamsDst );
```

For more information about the Color Space Library, see "The Color Space Library" in Appendix A.

## Summary of the Digital Media Image Conversion Library

These are the functions of the Digital Media Image Conversion Library API. More details about specific functions, such as the errors they return, can be found by looking at the reference pages mention in the “Description” column.

**Table 6-2** The Digital Media Image Conversion Library API

| Function | Description |
|---|---|
| int<br>**dmICGetNum** (<br>void ) | Return the number of image converters available. See also dmICGetNum(3dm). |
| int<br>**dmICChooseConverter** (<br>DMparams *_srcParams_,<br>DMparams *_dstParams_,<br>DMparams *_convParams_ ) | Return the index of an image converter that matches the specified image parameters. See also dmICChooseConverter(3dm). |
| DMstatus<br>**dmICGetDescription** (<br>int _i_, DMparams *_converterParams_ ) | Get the description of the converter indicated by index _i_. See also dmICGetDescription(3dm). |
| DMstatus<br>**dmICCreate** (<br>int _i_, DMimageconverter *_converter_ ) | Create an instance of image converter _i_. See also dmICCreate(3dm). |
| void<br>**dmICDestroy** (<br>DMimageconverter _converter_ ) | Destroy the image converter instance. See also dmICDestroy(3dm). |
| DMstatus<br>**dmICGetSrcParams** (<br>DMimageconverter _converter_,<br>DMparams *_srcParams_ ) | Get the actual settings of source parameters of an image converter instance. See also dmICGetSrcParams(3dm). |
| DMstatus<br>**dmICSetSrcParams** (<br>DMimageconverter _converter_,<br>DMparams *_srcParams_ ) | Change the settings of source parameters of an image converter instance. See also dmICSetSrcParams(3dm). |
| DMstatus<br>**dmICGetDefaultSrcParams** (<br>DMimageconverter _converter_,<br>DMparams *_srcParams_ ) | Get the source parameters and default values of an image converter instance. See also dmICGetDefaultSrcParams(3dm). |

**Table 6-2 (continued)** The Digital Media Image Conversion Library API

| Function | Description |
|---|---|
| DMstatus **dmICGetDstParams** ( DMimageconverter *converter*, DMparams **dstParams* ) | Get the actual settings of destination parameters of an image converter instance. See also dmICGetDstParams(3dm). |
| DMstatus **dmICSetDstParams** ( DMimageconverter *converter*, DMparams **dstParams* ) | Change the settings of destination parameters of an image converter instance. See also dmICSetDstParams(3dm). |
| DMstatus **dmICGetDefaultDstParams** ( DMimageconverter *converter*, DMparams **dstParams* ) | Get the destination parameters and default values of an image converter instance. See also dmICGetDefaultDstParams(3dm). |
| DMstatus **dmICGetConvParams** ( DMimageconverter *converter*, DMparams **convParams* ) | Get the actual settings of conversion parameters of an image converter instance. See also dmICGetConvParams(3dm). |
| DMstatus **dmICSetConvParams** ( DMimageconverter *converter*, DMparams **convParams* ) | Change the settings of conversion parameters of an image converter instance. See also dmICSetConvParams(3dm). |
| DMstatus **dmICGetDefaultConvParams** ( DMimageconverter *converter*, DMparams **convParams* ) | Get the conversion parameters and default values of an image converter instance. See also dmICGetDefaultConvParams(3dm). |
| DMstatus **dmICGetSrcPoolParams** ( DMimageconverter *converter*, DMparams **poolParams* ) | Get the input buffering needs of the image converter instance. See also dmICGetSrcPoolParams(3dm). |
| DMstatus **dmICGetDstPoolParams** ( DMimageconverter *converter*, DMparams **poolParams* ) | Get the output buffering needs of the image converter instance. See also dmICGetDstPoolParams(3dm). |

**Table 6-2 (continued)** The Digital Media Image Conversion Library API

| Function | Description |
|---|---|
| DMstatus<br>**dmICSetDstPool** (<br>DMimageconverter *converter*,<br>DMbufferpool *pool* ) | Set the pool from which *converter* allocates each output DMbuffer. See also dmICSetDstPool(3dm). |
| int<br>**dmICGetDstQueueFD** (<br>DMimageconverter *converter* ) | Get the queue file descriptor of the image converter instance. See also dmICGetDstQueueFD(3dm). |
| int<br>**dmICGetDstQueueFilled** (<br>DMimageconverter *converter* ) | Get the number of buffers ready to be received from the converter instance. See also dmICGetDstQueueFilled(3dm). |
| int<br>**dmICGetSrcQueueFilled** (<br>DMimageconverter *converter* ) | Get the number of buffers sent to the converter instance, but not yet processed. See also dmICGetSrcQueueFilled(3dm). |
| int<br>**dmICSend** (<br>DMimageconverter *converter*,<br>DMbuffer *srcBuffer*, int *numRefBuffers*,<br>DMbuffer **refBuffers* ) | Transfer source buffer and reference buffers to the image converter instance. See also dmICSend(3dm). |
| DMstatus<br>**dmICReceive** (<br>DMimageconverter *converter*,<br>DMbuffer **dstBuffer* ) | Transfer data from the image converter instance. See also dmICReceive(3dm). |
| DMstatus<br>**dmICAnyToAny** (<br>void **src*, void **dst*, DMparams **srcParams*,<br>DMparams **dstParams*,<br>DMparams **convParams* ) | A convenience function. |

## Audio Data Conversion

This section describes the Digital Media Audio Conversion Library, its converters, and how to use the Audio Conversion API to execute the four steps of an audio data conversion.

### The Digital Media Audio Conversion Library

The Digital Media Audio Conversion Library provides data format conversion for applications that do real-time audio capture, playback and file conversion. It makes it possible to efficiently move data between any audio producer and any audio consumer, regardless of their native formats. The library provides a single API for performing memory-to-memory sound compression and conversion. Table 6-1 lists commonly installed codec options that can be accessed through the Audio Conversion API. The "DM_AUDIO_COMPRESSION Value" column contains the identification values that are used with **dmACSetParams()** as described in "Configuring a Converter Instance Using the Audio Conversion API."

**Table 6-3** Digital Media Audio Codecs

| DM_AUDIO_COMPRESSION Value | Description |
|---|---|
| DM_AUDIO_DVI | Intel's Digital Video Interactive. See also "The DVI Audio Compression Library" in Appendix A. |
| DM_AUDIO_G711_ALAW<br>DM_AUDIO_G711_ULAW | International Telecommunication Union Standard G.711. See "The G.711 Audio Compression Library" in Appendix A. |
| DM_AUDIO_G722 | International Telecommunication Union Standard G.722. See "The G.722 Audio Compression Library" in Appendix A. |
| DM_AUDIO_G726 | International Telecommunication Union Standard G.726. See "The G.726 Audio Compression Library" in Appendix A. |
| DM_AUDIO_G728 | International Telecommunication Union Standard G.728. See "The G.728 Audio Compression Library" in Appendix A. |
| DM_AUDIO_GSM | Global System for Mobile Telecommunications. See "The GSM Audio Compression Library" in Appendix A. |

**Table 6-3 (continued)** Digital Media Audio Codecs

| DM_AUDIO_COMPRESSION Value | Description |
|---|---|
| DM_AUDIO_MPEG1 | MPEG-1 Audio. See "The MPEG-1 Audio Compression Library" in Appendix A. |
| DM_AUDIO_MULTIRATE | Aware MultiRate near-lossless compression. |

In addition to the compression and decompression done by the codecs, an audio converter instance may also perform such transformations as

- converting between different numerical representations, such as unsigned integer and two's complement signed integer
- converting between big-endian and little-endian byte orders
- audio sampling rate conversion (see "The Audio Rate Conversion Library" in Appendix A)
- converting between different numbers of interleaved channels, such as mono and stereo
- Pulse Code Modulation (PCM) mapping

As the next sections demonstrate, The Audio Conversion Library enables you to effectively use the codecs and transformation libraries by following the steps listed in "Using The Digital Media Converters." To use the Audio Conversion API, you must use these header files:

```
#include <dmedia/dm_audioconvert.h>
#include <dmedia/dm_audioutil.h>
```

### Creating a Converter Instance Using the Audio Conversion API

Use **dmACCreate()** to create an audio converter instance:

```
DMstatus dmACCreate ( DMaudioconverter* converter )
```

This function creates and initializes *converter*, a handle to a DMaudioconverter instance. All the Audio Conversion Library functions use this handle which is declared as follows:

```
typedef struct _DMaudioconverter *DMaudioconverter;
```

**Note:** All of the Audio Conversion Library functions return a DMstatus value of DM_SUCCESS if they succeed, DM_FAILURE if not. After a receiving a DM_FAILURE, your application can call the function the function **dmGetErrorForPID()** or **dmGetError()** to return an error message and error number. See "Digital Media Error Handling" in Chapter 3 for more information.

### Configuring a Converter Instance Using the Audio Conversion API

Once a converter instance has been created, it must be configured. As with the Image Conversion Library, your application does this by using DMparams data structures to specify a set of source parameters, a set of destination parameters, and, optionally, a set of conversion parameters.

The Audio Conversion Library enables you to do the following:

- Configure an audio converter instance by setting the source, destination, and conversion parameters with **dmACSetParams()**
- Get the source, destination, and conversion parameter settings of a configured audio converter instance with **dmACGetParams()**
- Reset an audio converter instance to its original configuration with **dmACReset()**

Use **dmACSetParams()** to set the DMaudioconverter parameter values:

```
DMstatus dmACSetParams ( DMaudioconverter converter,
                         DMparams *sourceparams, DMparams *destparams,
                         DMparams *conversionparams )
```

The handle *converter* indicates an audio converter instance created by a previous call to **dmACCreate()**. The DMparams structures pointed to by *sourceparams* and *destparams* describe the formats of the audio data prior to and after conversion. The variable *conversionparams* points to a DMparams structure that contains parameters specific to the conversion process. The variables *destparams* and *conversionparams* are optional and may be set to NULL.

Use **dmACGetParams()** to get the DMaudioconverter parameter values:

```
DMstatus dmACGetParams ( DMaudioconverter converter,
                         DMparams *sourceparams, DMparams *destparams,
                         DMparams *conversionparams )
```

The handle *converter* indicates an audio converter instance previously configured by a call to **dmACSetParams()**. The DMparams structures pointed to by *sourceparams* and

*destparams* describe the formats of the audio data prior to and after conversion. The variable *conversionparams* points to a DMparams structure that contains parameters specific to the conversion process. The variables *destparams* and *conversionparams* are optional and may be set to NULL.

Use **dmACReset()** to reset a DMaudioconverter handle to its original configuration:

```
DMstatus dmACReset ( DMaudioconverter converter )
```

The handle *converter* indicates an audio converter instance already configured by a call to **dmACSetParams()**.

The next three sections describe the DMparams structures for source, destination, and conversion parameters in more detail. They are followed by a section that discusses parameters relevant to specific converters.

**Source Parameters**

The source parameters, which describe the data to be converted, are contained in the DMparams structure indicated by *sourceparams*. The source parameters that must be specified are shown below. There are no default values. The parameters and their DMparams-setting functions follow the bullets. Legal values for the parameters follow the dashes.

- DM_AUDIO_FORMAT and **dmParamsSetEnum():**

  DM_AUDIO_TWOS_COMPLEMENT

  DM_AUDIO_UNSIGNED

  DM_AUDIO_FLOAT

  DM_AUDIO_DOUBLE

- DM_AUDIO_WIDTH and **dmParamsSetInt()**:

  The width of the data in bits. An integer value between 1 and 32.

- DM_AUDIO_BYTE_ORDER and **dmParamsSetEnum()**:

  DM_AUDIO_BIG_ENDIAN

  DM_AUDIO_LITTLE_ENDIAN

- DM_AUDIO_CHANNELS and **dmParamsSetInt()**:

  The number of audio channels. An integer value greater than 0.

- DM_AUDIO_RATE and **dmParamsSetFloat()**:

  The audio sampling rate in Hz. A DM_TYPE_FLOAT value greater than 0.0.

- DM_AUDIO_COMPRESSION and **dmParamsSetString()**:

  DM_AUDIO_UNCOMPRESSED or one of the values shown in Table 6-1.

**Destination Parameters**

The destination parameters, which describe the output audio data, are contained in the DMparams structure indicated by *destparams*. Any destination parameter not specified defaults to its source parameter value, except DM_AUDIO_COMPRESSION which defaults to DM_AUDIO_UNCOMPRESSED.

There is a set of four parameters for PCM mapping whose values, although they can be set for source data, are normally specified only for destination data. The parameters are based on a model where there is a PCM value that corresponds to zero voltage and a differential value that corresponds to full voltage. The set consists of the following parameters, which are shown with their DMparams setting functions and appropriate values.

- DM_AUDIO_PCM_MAP_SLOPE and **dmParamsSetFloat()**:

  The full voltage PCM value. (Default is 32767.0)

- DM_AUDIO_PCM_MAP_INTERCEPT and **dmParamsSetFloat()**:

  The zero voltage PCM value. (Default is 0.0)

- DM_AUDIO_PCM_MAP_MAXCLIP and **dmParamsSetFloat()**:

  Clip all PCM values to this maximum value. (Default is 32767.0)

- DM_AUDIO_PCM_MAP_MINCLIP and **dmParamsSetFloat()**:

  Clip all PCM values to this minimum value. (Default is -32768.0)

The function **dmACSetParams()** automatically sets their default input and output values from the input and output data format specifications. Your application needs to set them only if it has special mapping requirements, such as input data with a fixed offset like a DC bias. If your application sets any of these four parameters, it must set all of them. See afIntro(3dm) for more information on PCM mapping.

### Conversion Parameters

The conversion parameters, which modify the codec settings and other aspects of the conversion process, are contained in the DMparams structure indicated by *conversionparams*. There are five categories of conversion parameters.

1. The Processing Mode Parameter

   This parameter, whose keyword is DM_AUDIO_PROCESS_MODE, is used to determine the converter's Processing mode. It can also be used to set the processing mode when both the input and output data are uncompressed. There are two processing modes: push and pull. In *pull* mode, your application requests a given number of output frames. Decompression must use pull mode and your application uses a buffer length parameter to specify how many frames of uncompressed data the converter instance should put in the output buffer of **dmACConvert()**. In *push* mode, your application gives the converter a specified number of input frames. Compression requires push mode and your application specifies how many frames of uncompressed data are in the input buffer. The two settings for DM_AUDIO_PROCESS_MODE are

   - DM_AUDIO_PROCESS_PULL and **dmParamsGetInt()**
   - DM_AUDIO_PROCESS_PUSH and **dmParamsGetInt()**

2. Buffer Length Parameters

   The three buffer length parameters are used to determine the number of frames in the input or output buffers. Your application specifies them only during compression, decompression, or rate conversion because the input and output buffer lengths are equal at all other times. Your application must set the parameter DM_AUDIO_MAX_REQUEST_LEN prior to calling **dmACConvert()** to specify the largest buffer the converter instance will have to process. Your application then calls **dmACGetParams()** to find the value of DM_AUDIO_MIN_INPUT_LEN or DM_AUDIO_MIN_OUTPUT_LEN. If the converter instance is in pull mode, DM_AUDIO_MIN_INPUT_LEN gives the minimum number of frames the converter instance requires in the input buffer. If the instance is in push mode, DM_AUDIO_MIN_OUTPUT_LEN gives the minimum number of frames the instance requires in the output buffer. Your application then allocates buffers appropriate to these sizes. The parameters, DMparams functions, and typical values are as follows:

   - DM_AUDIO_MAX_REQUEST_LEN and **dmParamsSetInt():**

     An integer value greater than 0. This value can only be set.

- DM_AUDIO_MIN_INPUT_LEN and **dmParamsGetInt():**

  An integer value that your application can only read.

- DM_AUDIO_MIN_OUTPUT_LEN and **dmParamsGetInt():**

  An integer value that your application can only read.

3. The Dithering Parameter

   The dithering parameter, whose keyword is DM_AUDIO_DITHER_ALGORITHM, is used only when data is converted from a larger to a smaller data type. An example of such a conversion would be going from a floating point to a 16-bit integer representation. The dithering algorithm is applied to reduce the quantization error distortion inherent in reducing resolution. The two possible values are as follows:

   - DM_AUDIO_DITHER_NONE (default)
   - DM_AUDIO_DITHER_LSB_TPDF (Least Significant Bit - Triangular Probability Density Function)

4. Rate Conversion Parameters

   There are three parameters that affect the rate conversion algorithm. They are used only when the input and output sampling rates are not equal. The three parameters, their DMparams setting functions, and their possible values are shown below. See dmAudioRateConverterSetParams(3dm) for more information about them.

   - DM_AUDIO_RC_ALGORITHM and **dmParamsSetString()**:

     DM_AUDIO_RC_JITTER_FREE (default)

     DM_AUDIO_RC_POLYNOMIAL_ORDER_1

     DM_AUDIO_RC_POLYNOMIAL_ORDER_3

   - DM_AUDIO_RC_JITTER_FREE_STOPBAND_ATTENUATION and **dmParamsSetFloat()**:

     DM_AUDIO_RC_JITTER_FREE_STOPBAND_ATTENUATION_78_DB (default)

     DM_AUDIO_RC_JITTER_FREE_STOPBAND_ATTENUATION_96_DB

     DM_AUDIO_RC_JITTER_FREE_STOPBAND_ATTENUATION_120_DB

   - DM_AUDIO_RC_JITTER_FREE_TRANSITION_BANDWIDTH and **dmParamsSetFloat()**:

DM_AUDIO_RC_JITTER_FREE_TRANSITION_BANDWIDTH_1_PERCENT

DM_AUDIO_RC_JITTER_FREE_TRANSITION_BANDWIDTH_10_PERCENT

DM_AUDIO_RC_JITTER_FREE_TRANSITION_BANDWIDTH_20_PERCENT

5. The Channel Conversion Parameter

   The channel conversion or channel matrix parameter is used to mix the channels associated with a track. The matrix is a one-dimensional array composed of a two-dimensional array in row-major order, where each row represents an output channel and each column represents an input channel. See the afSetChannelMatrix(3dm) reference page for a detailed explanation. The parameter keyword and its DMparams-setting function are as follows:

   - DM_AUDIO_CHANNEL_MATRIX and **dmParamsSetFloatArray()**:

     A DMfloatarray of double-precision floating point numbers.

### Converting Data Using the Audio Conversion API

Use **dmACConvert()** to convert the audio data's format, sampling rate, and compression:

```
DMstatus dmACConvert ( DMaudioconverter converter, void *inbuffer,
                       void *outbuffer, int *in_amount,
                       int *out_amount )
```

This function performs the data format, sampling rate, and compression or decompression specified by **dmACSetParams()**. The variable *converter* is a handle to an audio converter instance previously created with **dmACCreate()** and configured with **dmACSetParams()**. The variables *inbuffer* and *outbuffer* point to the buffers that contain the audio data prior to and after conversion. As described in "Configuring a Converter Instance Using the Audio Conversion API," your application may need to determine the number of frames these buffers must hold using the DM_AUDIO_MIN_INPUT_LEN or DM_AUDIO_MIN_OUTPUT_LEN parameters. If *inbuffer* is NULL, the converter instance flushes any internal buffers to the output buffer.

The variable *in_amount* points to an integer containing the number of frames (bytes if the data is compressed) of input data available to the converter instance. This can be any value greater than 0. In pull mode, **dmACConvert()** resets this value to the number of frames (bytes) read from *inbuffer* by the converter instance. (To review the push and pull modes, see the conversion parameter discussion in "Configuring a Converter Instance Using the Audio Conversion API.") The pointer *out_amount* indicates an integer

containing the number of frames (bytes if the data is compressed) of converted data your application wants from the converter instance. The initial value is ignored in push mode. After processing the data, **dmACConvert()** resets the *out_amount* value to the number of frames (bytes) actually placed into *outbuffer*. If the conversion involved rate conversion, compression, or decompression, this value can vary significantly from the in_amount value. It can even be zero when the *in_amount* value was positive.

### Compression Parameters

The compression parameters modify individual audio converters listed in Table 6-1. The parameters and their values are listed below. For more information about using specific parameters, please refer to the relevant reference pages.

- Digital Video Interactive (DVI) has one parameter.

  DM_DVI_AUDIO_BITS_PER_SAMPLE specifies the compression algorithm. Its possible values are

  - DM_DVI_AUDIO_3BITS_PER_SAMPLE
  - DM_DVI_AUDIO_4BITS_PER_SAMPLE (default)

- ITU-T G722 has three parameters.

  DM_AUDIO_CODEC_MAX_BYTES_PER_BLOCK

  DM_AUDIO_CODEC_FRAMES_PER_BLOCK

  DM_AUDIO_CODEC_FILTER_DELAY

- ITU-T G722 has two parameters.

  DM_AUDIO_BITRATE is an integer in units of bits per second with one of the following values: 16000, 24000, 32000, 40000.

  DM_G726_NATIVE_FORMAT specifies the input or output sample data format. Its possible values are

  - AUDIO_ENCODING_ULAW
  - AUDIO_ENCODING_ALAW
  - AUDIO_ENCODING_LINEAR

- ITU-T G728 has one parameter.

  DM_G728_POSTFILTERING_FLAG selects a decoder with or without post filtering. Its possible values are

- DM_G728_POSTFILTERING_YES
- DM_G728_POSTFILTERING_NO

- Global System for Mobile Telecommunications (GSM) has three parameters.

  DM_AUDIO_CODEC_MAX_BYTES_PER_BLOCK

  DM_AUDIO_CODEC_FRAMES_PER_BLOCK

  DM_AUDIO_CODEC_FILTER_DELAY

- MPEG-1 Audio has seven parameters for encoding and decoding.

  DM_AUDIO_RATE is the input or output sampling rate given in Hz. It is a double with possible values of 32000, 44100 (default), and 48000.

  DM_AUDIO_FORMAT is the format of each input or output sample. The only supported value is DM_AUDIO_TWOS_COMPLEMENT

  DM_AUDIO_WIDTH is the width of each input or output sample. The only supported value is DM_AUDIO_WIDTH_16

  DM_AUDIO_CHANNELS is the number of channels in the input or output data. It is an integer value of 1 or 2 (default).

  DM_MPEG1_AUDIO_LAYER is a flag specifying the basic algorithm to be used. There are two possible values.

  - DM_MPEG1_AUDIO_LAYER1
  - DM_MPEG1_AUDIO_LAYER2 (default)

  DM_AUDIO_CHANNEL_POLICY indicates how multiple channels should be treated. There are three possible values.

  - DM_AUDIO_STEREO - The channels are part of a single multichannel signal.
  - DM_AUDIO_JOINT_STEREO (default) - The algorithm may attempt to exploit redundancy between channels for greater coding gain.
  - DM_AUDIO_INDEPENDENT - The separate channels are unrelated and should be processed separately.

  DM_AUDIO_BIT_RATE specifies the desired bit rate, in bits per second, for the compressed data.

Supported values with DM_MPEG1_AUDIO_LAYER1 are 32000, 64000, 96000, 128000, 160000, 192000, 224000, 256000 (default), 288000, 320000, 352000, 384000, 416000, and 448000. Supported values with DM_MPEG1_AUDIO_LAYER2 are 32000, 48000, 56000, 64000, 80000, 96000, 112000, 128000, 160000, 192000, 224000, 256000 (default), 320000, and 38400

- MPEG-1 Audio has four parameters to use with encoding only.

  DM_MPEG1_AUDIO_PSYCHOMODEL selects which psychoacoustic model to use for calculating the safe masking thresholds for quantizing noise. There are two possible values.

  - DM_MPEG1_AUDIO_PSYCHOMODEL1
  - DM_MPEG1_AUDIO_PSYCHOMODEL2

  DM_MPEG1_AUDIO_PSYCHOMODEL1_ALPHA is a float value that specifies the alpha parameter for DM_MPEG1_AUDIO_PSYCHOMODEL1. It has a possible value within (0.0, 2.0]. Default is 2.0.

  DM_MPEG1_AUDIO_BITRATE_POLICY is a flag used for the interpretation of DM_AUDIO_BIT_RATE. There are two possible values.

  - DM_MPEG1_AUDIO_FIXRATE (default)
  - DM_MPEG1_AUDIO_CONSTANT_QUALITY

  DM_MPEG1_AUDIO_CONST_QUAL_NMR is the desired mask-to-noise ratio in decibels. It is a float value with a value within (-13.0, 13.0]. The default is 0.0.

- MPEG-1 Audio has three parameters to use with decoding only.

  DM_MPEG1_AUDIO_DECIMATION_SCALE specifies the decimation factor applied to reduce the complexity of decoding. It has three possible values.

  - DM_MPEG1_AUDIO_BANDWIDTH_FULL (default)
  - DM_MPEG1_AUDIO_BANDWIDTH_HALF
  - DM_MPEG1_AUDIO_BANDWIDTH_QUARTER

  DM_MPEG1_AUDIO_SCALE_FILTERSHAPE specifies the filter shape applied to reduce the complexity of decoding. It has three possible values.

  - DM_MPEG1_AUDIO_DEFAULT_FILTER (default)
  - DM_MPEG1_AUDIO_FILTER_SHAPE1
  - DM_MPEG1_AUDIO_FILTER_SHAPE2

DM_MPEG1_AUDIO_COMBINE_CHANS_FLAG enables single channel output. It is an integer with a default value of 0, that is JOINT_STEREO mode. A value of 1 forces the decoder to produce single channel output.

- MPEG-1 Audio has three parameters to use only in query mode.

  DM_AUDIO_CODEC_FRAMES_PER_BLOCK is an integer that specifies how many sample frames are put into each compressed data block.

  DM_AUDIO_CODEC_MAX_BYTES_PER_BLOCK is an integer that indicates the maximum number of bytes that will compose a compressed data block.

  DM_AUDIO_CODEC_FILTER_DELAY is an integer that indicates the delay, in sample frames, introduced by compression and decompression processing.

### Destroying a Converter Instance Using the Audio Conversion API

The Audio Conversion library provides the function **dmACDestroy()** to destroy an audio converter instance:

```
DMstatus dmACDestroy ( DMaudioconverter converter )
```

The function frees the memory associated with the DMaudioconverter handle. The handle is not valid after this call returns.

## Summary of the Digital Media Audio Conversion Library

These are the functions of the Digital Media Audio Conversion Library API. More details about specific functions, such as the errors they return, can be found by looking at the reference pages mention in the “Description” column.

**Table 6-4** The Digital Media Audio Conversion API

| Function | Description |
|---|---|
| DMstatus **dmACConvert** ( DMaudioconverter *converter*, void **inbuffer*, void **outbuffer*, int **in_amount*, int **out_amount* ) | Convert the audio data format, sampling rate, and compression. See also dmACConvert(3dm). |
| DMstatus **dmACCreate** ( DMaudioconverter **converter* ) | Create a DMaudioconverter handle to use for audio format conversion. See also dmACCreate(3dm). |

**Table 6-4 (continued)** The Digital Media Audio Conversion API

| Function | Description |
|---|---|
| DMstatus<br>**dmACDestroy** (<br>DMaudioconverter *converter* ) | Destroy a DMaudioconverter handle used for audio format conversion. See also dmACDestroy(3dm). |
| DMstatus<br>**dmACReset** (<br>DMaudioconverter *converter* ) | Reset a DMaudioconverter handle to its default state. See also dmACReset(3dm). |
| DMstatus<br>**dmACSetParams** (<br>DMaudioconverter *converter*,<br>DMparams **sourceparams*,<br>DMparams **destparams*,<br>DMparams **conversionparams* ) | Set the DMaudioconverter parameter values. See also dmACSetParams(3dm). |
| DMstatus<br>**dmACGetParams** (<br>DMaudioconverter *converter*,<br>DMparams **sourceparams*,<br>DMparams **destparams*,<br>DMparams **conversionparams* ) | Get the DMaudioconverter parameter values. See also dmACGetParams(3dm). |

*Appendix A*

# Digital Media Conversion Libraries

This appendix contains the APIs of the individual image and audio conversion libraries. These libraries are not discussed in detail. As discussed in Chapter 6, “Digital Media Data Conversion,” most developers need only use the Image Conversion Library and the Audio Conversion Library. Those two libraries call the libraries described in this appendix as needed.

## The Color Space Library

The Color Space Library (CSL) provides developers with the ability to convert the color spaces, packings, subsamplings, and data types of images. It also enables them to perform operations on image data, such as adjusting contrast. Like the Image Conversion Library, the CSL uses parameters describing the source and destination images as well as the conversion settings to create a *color converter* The image parameters are set using a **DMparams** structure using **dmColorSetSrcParams()** and **dmColorSetDstParams()**. The conversion parameters are set with **dmColorSetConvParams()**. In most cases, only the source and destination packings need to be specified; all other values will default to appropriate values. The CSL supports the RGB, YCrCb, and Y (Luminance) color spaces.

**Table A-1** The Color Space Library API

| Function | Description |
|---|---|
| DMstatus<br>**dmColorConvert** (<br>const DMcolorconverter *converter*,<br>void **srcImage*, void **dstImage* ) | Perform the image conversion<br>See also dmColorConvert(3dm). |
| DMstatus<br>**dmColorCreate** (<br>DMcolorconverter **converter* ) | Create and initialize the color converter.<br>See also dmColorCreate(3dm). |
| DMstatus<br>**dmColorDestroy** (<br>const DMcolorconverter *converter* ) | Destroy the color converter.<br>See also dmColorDestroy(3dm). |

**Table A-1 (continued)** The Color Space Library API

| Function | Description |
|---|---|
| DMstatus<br>**dmColorGetError** (<br>const DMcolorconverter *converter*, int **error* ) | Return the value of the error flag.<br>See also dmColorGetError(3dm). |
| const char *<br>**dmColorGetErrorString** (<br>const int *error* ) | Returns a text error message.<br>See also dmColorGetErrorString(3dm). |
| DMstatus<br>**dmColorGetSrcSize** (<br>const DMcolorconverter *converter*, int **size* ) | Get the source image size in bytes.<br>See also dmColorGetSrcSize(3dm). |
| DMstatus<br>**dmColorGetDstSize** (<br>const DMcolorconverter *converter*, int **size* ) | Get the destination image size in bytes.<br>See also dmColorGetSrcSize(3dm). |
| DMstatus<br>**dmColorGetTransformMatrix** (<br>const DMcolorconverter *converter*,<br>const int *format*, double *matrix*[] ) | Get the transform matrix.<br>See also dmColorGetTransformMatrix(3dm). |
| DMstatus<br>**dmColorPrecompute** (<br>const DMcolorconverter *converter* ) | Perform any required early computations.<br>See also dmColorPrecompute(3dm). |
| DMstatus<br>**dmColorSetBrightness** (<br>const DMcolorconverter *converter*,<br>const float *brightness* ) | Set the brightness delta value.<br>See also dmColorSetBrightness(3dm). |
| DMstatus<br>**dmColorGetBrightness** (<br>const DMcolorconverter *converter*,<br>float **brightness* ) | Get brightness delta value. See also<br>dmColorSetBrightness(3dm). |
| DMstatus<br>**dmColorSetContrast** (<br>const DMcolorconverter *converter*,<br>const float *contrast* ) | Set the contrast multiplier. See also<br>dmColorSetContrast(3dm). |

**Table A-1 (continued)** The Color Space Library API

| Function | Description |
|---|---|
| DMstatus **dmColorGetContrast** ( const DMcolorconverter *converter*, float **contrast* ) | Get the contrast multiplier. See also dmColorSetContrast(3dm). |
| DMstatus **dmColorSetDefaultAlpha** ( const DMcolorconverter *converter*, const float *defaultAlpha* ) | Set the default alpha value of the source image. See also dmColorSetDefaultAlpha(3dm). |
| DMstatus **dmColorGetDefaultAlpha** ( const DMcolorconverter *converter*, float **defaultAlpha* ) | Get the default alpha value of the source image. See also dmColorSetDefaultAlpha(3dm). |
| DMstatus **dmColorSetHue** ( const DMcolorconverter *converter*, const float *hue* ) | Set the hue rotation. See also dmColorSetHue(3dm). |
| DMstatus **dmColorGetHue** ( const DMcolorconverter *converter*, float **hue* ) | Get the hue rotation. See also dmColorSetHue(3dm). |
| DMstatus **dmColorSetSaturation** ( const DMcolorconverter *converter*, const float *saturation* ) | Set the saturation multiplier. See also dmColorSetSaturation(3dm). |
| DMstatus **dmColorGetSaturation** ( const DMcolorconverter *converter*, float **saturation* ) | Get the saturation multiplier. See also dmColorGetSaturation(3dm). |
| DMstatus **dmColorSetSrcParams** ( const DMcolorconverter *converter*, DMparams **imageParams* ) | Set the source image parameters. See also dmColorSetSrcParams(3dm). |

**Table A-1 (continued)** The Color Space Library API

| Function | Description |
|---|---|
| DMstatus **dmColorSetDstParams** ( const DMcolorconverter *converter*, DMparams **imageParams* ) | Set the destination image parameters. See also dmColorSetSrcParams(3dm). |
| DMstatus **dmColorGetSrcParams** ( const DMcolorconverter *converter*, DMparams **imageParams* ) | Get the source image parameters. See also dmColorSetSrcParams(3dm). |
| DMstatus **dmColorGetDstParams** ( const DMcolorconverter *converter*, DMparams **imageParams* ) | Get the destination image parameters. See also dmColorSetSrcParams(3dm). |
| DMstatus **dmColorSetSubsamplingFilter** ( const DMcolorconverter *converter*, const int *subsamplingFilter* ) | Set the subsampling filter type. See also dmColorSetSubsamplingFilter(3dm). |
| DMstatus **dmColorGetSubsamplingFilter** ( const DMcolorconverter *converter*, int **subsamplingFilter* ) | Get the subsampling filter type. See also dmColorSetSubsamplingFilter(3dm). |

## The DVI Audio Compression Library

The DVI Audio Compression Library is based on Intel's Digital Video Interactive audio compression technology for multimedia applications. It implements the IMA (Interactive Multimedia Association) recommendations for ADPCM compression and decompression based on Intel's DVI algorithm.

**Table A-2** The DVI Audio Library API

| Function | Description |
|---|---|
| DMstatus **dmDVIAudioDecode** ( DMDVIaudiodecoder *handle*, unsigned char **ibuf*, short **obuf*, int *nsamples* ) | Do ADPCM decompression based on Intel's DVI algorithm. See also dmDVIAudioDecode(3dm). |
| DMstatus **dmDVIAudioDecoderCreate** ( DMDVIaudiodecoder **decoder* ) | Allocate a new DMDVIaudiodecoder structure. See also dmDVIAudioDecoderCreate(3dm). |
| DMstatus **dmDVIAudioDecoderDestroy** ( DMDVIaudiodecoder *handle* ) | Deallocate a DMDVIaudiodecoder. See also dmDVIAudioDecoderDestroy(3dm). |
| DMstatus **dmDVIAudioDecoderSetParams** ( DMDVIaudiodecoder *handle*, DMparams **params* ) | Set parameter values for a DMDVIaudiodecoder structure. See also dmDVIAudioDecoderSetParams(3dm). |
| DMstatus **dmDVIAudioDecoderGetParams** ( DMDVIaudiodecoder *handle*, DMparams **params* ) | Get parameter values for a DMDVIaudiodecoder structure. See also dmDVIAudioDecoderGetParams(3dm). |
| DMstatus **dmDVIAudioDecoderReset** ( DMDVIaudiodecoder *handle* ) | Fill buffers of a DMDVIaudiodecoder structure with zeroes. See also dmDVIAudioDecoderReset(3dm). |
| DMstatus **dmDVIAudioEncode** ( DMDVIaudioencoder *handle*, short **ibuf*, unsigned char **obuf*, int *nsamples* ) | Do ADPCM compression based on Intel's DVI algorithm. See also dmDVIAudioEncode(3dm). |

**Table A-2** The DVI Audio Library API

| Function | Description |
|---|---|
| DMstatus **dmDVIAudioEncoderCreate** ( DMDVIaudioencoder **encoder* ) | Allocate a new DMDVIaudioencoder structure. See also dmDVIAudioEncoderCreate(3dm). |
| DMstatus **dmDVIAudioEncoderDestroy** ( DMDVIaudioencoder *handle* ) | Deallocate a DMDVIaudioencoder structure. See also dmDVIAudioEncoderDestroy(3dm). |
| DMstatus **dmDVIAudioEncoderSetParams** ( DMDVIaudioencoder *handle*, DMparams **params* ) | Set parameter values for DMDVIaudioencoder structure. See also dmDVIAudioEncoderSetParams(3dm). |
| DMstatus **dmDVIAudioEncoderGetParams** ( DMDVIaudioencoder *handle*, DMparams **params* ) | Get parameter values for DMDVIaudioencoder structure. See also dmDVIAudioEncoderGetParams(3dm). |
| DMstatus **dmDVIAudioEncoderReset** ( DMDVIaudioencoder *handle* ) | Fill buffers of a DMDVIaudioencoder structure with zeroes. See also dmDVIAudioEncoderReset(3dm). |

## The G.711 Audio Compression Library

This library implements International Telecommunication Union Standard (ITU-T, formerly CCITT) G.711 for compression and decompression. The standard is for 64 Kb / s, 8 kHz, 16-bit pulse code modulation (PCM) audio encoding of voice frequencies.

**Table A-3** The G.711 Audio Compression Library API

| Function | Description |
|---|---|
| void **dmG711MulawEncode** ( short **samples*, unsigned char **mulawdata*, int *numsamples* ) | Convert a 16-bit linear PCM value to an 8-bit μ-law value. See also dmG711(3dm). |
| void **dmG711MulawDecode** ( unsigned char **mulawdata*, short **samples*, int *numsamples* ) | Convert an 8-bit μ-law value to a 16-bit linear PCM value. See also dmG711(3dm). |
| void **dmG711MulawZeroTrapEncode** ( short **samples*, unsigned char **mulawdata*, int *numsamples* ) | Do ITU G.711 μ-law compression with zero trap during compression. See also dmG711(3dm). |
| void **dmG711MulawZeroTrapDecode** ( unsigned char **mulawdata*, short **samples*, int *numsamples* ) | Same as **dmG711MulawDecode()**. See also dmG711(3dm). |
| void **dmG711AlawEncode** ( short **samples*, unsigned char **Alawdata*, int *numsamples* ) | Convert a 16-bit linear PCM value to an 8-bit A-law value. See also dmG711(3dm). |
| void **dmG711AlawDecode** ( unsigned char **Alawdata*, short **samples*, int *numsamples* ) | Convert an 8-bit A-law value to a 16-bit linear PCM value. See also dmG711(3dm). |
| void **dmG711MulawToAlaw** ( unsigned char **mulawdata*, unsigned char **Alawdata*, int *numsamples* ) | Convert μ -law data to A-law data. See also dmG711(3dm). |

**Table A-3 (continued)** The G.711 Audio Compression Library API

| Function | Description |
|---|---|
| void **dmG711AlawToMulaw** ( unsigned char **Alawdata*, unsigned char **mulawdata*, int *numsamples* ) | Convert A-law date to μ-law data. See also dmG711(3dm). |
| void **dmSunMulawEncode** ( short **samples*, unsigned char **mulawdata*, int *numsamples* ) | Convert a 16-bit linear PCM value to an 8-bit μ-law value using the conversion tables of Sun Microsystems. See also dmG711(3dm). |
| void **dmSunMulawDecode** ( unsigned char **mulawdata*, short **samples*, int *numsamples* ) | Convert an 8-bit μ-law value to a 16-bit linear PCM value using the conversion tables of Sun Microsystems. See also dmG711(3dm). |
| void **dmNeXTMulawEncode** ( short **samples*, unsigned char **mulawdata*, int *numsamples* ) | Convert a 16-bit linear PCM value to an 8-bit μ-law value using the conversion tables of NeXT Computers. See also dmG711(3dm). |
| void **dmNeXTMulawDecode** ( unsigned char **mulawdata*, short **samples*, int *numsamples* ) | Convert an 8-bit μ-law value to a 16-bit linear PCM value using the conversion tables of NeXT Computers. See also dmG711(3dm). |

## The G.722 Audio Compression Library

This library implements International Telecommunication Union Standard (ITU-T, formerly CCITT) G.722 for compression and decompression. The standard is for 7 kHz audio encoding within 64 Kb/s.

**Table A-4** The G.722 Audio Compression Library API

| Function | Description |
|---|---|
| DMstatus<br>**dmG722Decode** (<br>DMG722decoder *handle*,<br>unsigned char **ibuf*, short **obuf*,<br>int *nsamples* ) | Do G.722 decompression. See also dmG722Decode(3dm). |
| DMstatus<br>**dmG722DecoderCreate** (<br>DMG722decoder **decoder*, int *maxsamples*,<br>int *decodemode* ) | Allocate a new DMG722decoder structure. See also dmG722DecoderCreate(3dm). |
| DMstatus<br>**dmG722DecoderDestroy** (<br>DMG722decoder *handle*) | Deallocate a DMG722decoder structure. See also dmG722DecoderDestroy(3dm). |
| DMstatus<br>**dmG722DecoderGetParams** (<br>DMG722decoder *handle*,<br>DMparams **params* ) | Get parameter values for a DMG722decoder structure. See also dmG722DecoderGetParams(3dm). |
| DMstatus<br>**dmG722DecoderReset** (<br>DMG722decoder *handle*) | Fill buffers of a DMG722decoder structure with zeroes. See also dmG722DecoderReset(3dm). |
| DMstatus<br>**dmG722Encode** (<br>DMG722encoder *handle*, short **ibuf*,<br>unsigned char **obuf*, int *nsamples* ) | Do G.722 compression. See also dmG722Encode(3dm). |
| DMstatus<br>**dmG722EncoderCreate** (<br>DMG722encoder **encoder*, int *maxsamples*) | Allocate a new DMG722encoder structure. See also dmG722EncoderCreate(3dm). |
| DMstatus<br>**dmG722EncoderDestroy** (<br>DMG722encoder *handle* ) | Deallocate a DMG722Encoder structure. See also dmG722EncoderDestroy(3dm). |

**Table A-4 (continued)** The G.722 Audio Compression Library API

| Function | Description |
|---|---|
| DMstatus **dmG722EncoderGetParams** ( DMG722encoder *handle*, DMparams **params* ) | Get parameter values for aDMG722encoder structure. See also dmG722EncoderGetParams(3dm). |
| DMstatus **dmG722EncoderReset** ( DMG722encoder *handle*) | Fill buffers of a DMG722encoder structure with zeroes. See also dmG722EncoderReset(3dm). |

## The G.726 Audio Compression Library

This library implements International Telecommunication Union Standard (ITU-T, formerly CCITT) G.726 for ADPCM compression and decompression. The standard is for a compressed data bit stream of 40, 32, 24, or 16 Kb/s and a decompressed A-law, μ-law, or linear PCM data stream of 64 Kb/s.

**Table A-5** The G.726 Audio Compression Library API

| Function | Description |
|---|---|
| DMstatus **dmG726Decode** ( DMG726decoder *handle*, unsigned char **inBuffer*, short **outBuffer*, int *numSamples* ) | Do G.726 ADPCM decompression. See also dmG726Decode(3dm). |
| DMstatus **dmG726DecoderCreate** ( DMG726decoder **decoder*, int *bitRate*, int *outputMode* ) | Allocate a new DMG726decoder structure. See also dmG726DecoderCreate(3dm). |
| DMstatus **dmG726DecoderDestroy** ( DMG726decoder *handle* ) | Deallocate a DMG726decoder structure. See also dmG726DecoderDestroy(3dm). |
| DMstatus **dmG726DecoderSetParams** ( DMG726decoder *handle*, DMparams **params* ) | Set a DMG726decoder structure's parameter values. See also dmG726DecoderSetParams(3dm). |
| DMstatus **dmG726DecoderGetParams** ( DMG726decoder *handle*, DMparams **params* ) | Get a DMG726decoder structure's parameter values. See also dmG726DecoderGetParams(3dm). |
| DMstatus **dmG726DecoderReset** ( DMG726decoder *handle* ) | Fill a DMG726decoder structure's internal buffers with zeroes. See also dmG726DecoderReset(3dm). |
| DMstatus **dmG726Encode** ( DMG726encoder *handle*, short **ibuf*, unsigned char **obuf*, int *numSamples* ) | Do G.726 ADPCM compression. See also dmG726Encode(3dm). |

**Table A-5 (continued)** The G.726 Audio Compression Library API

| Function | Description |
|---|---|
| DMstatus **dmG726EncoderCreate** ( DMG726encoder **encoder*, int *bitRate*, int *outputMode* ) | Allocate a new DMG726encoder structure. See also dmG726EncoderCreate(3dm). |
| DMstatus **dmG726EncoderDestroy** ( DMG726encoder *handle* ) | Deallocate a DMG726Encoder structure. See also dmG726EncoderDestroy(3dm). |
| DMstatus **dmG72EncoderSetParams** ( DMG726encoder *handle*, DMparams **params* ) | Set a DMG726encoder structure's parameter values. See also dmG726EncoderSetParams(3dm). |
| DMstatus **dmG726EncoderGetParams** ( DMG726encoder *handle*, DMparams **params* ) | Get a DMG726decoder structure's parameter values. See also dmG726EncoderGetParams(3dm). |
| DMstatus **dmG726EncoderReset** ( DMG726encoder *handle* ) | Fill a DMG726encoder structure's internal buffers with zeroes. See also dmG726EncoderReset(3dm). |

## The G.728 Audio Compression Library

This library implements International Telecommunication Union Standard (ITU-T, formerly CCITT) G.728 for the audio encoding used in videoconferencing. The standard controls the coding of speech at 16 Kb/s using Low-Delay Code Excited Linear Prediction (LD-CELP).

**Table A-6** The G.728 Audio Compression Library API

| Function | Description |
|---|---|
| DMstatus **dmG728Decode** ( DMG728decoder *handle*, unsigned char **ibuf*, short **obuf*, int *nsamples* ) | Do ITU G.728 decompression (LD-CELP). See also dmG728Decode(3dm). |
| DMstatus **dmG728DecoderCreate** ( DMG728decoder **decoder* ) | Allocate a new DMG728decoder structure. See also dmG728DecoderCreate(3dm). |
| DMstatus **dmG728DecoderDestroy** ( DMG728decoder *handle* ) | Deallocate a DMG728decoder structure. See also dmG728DecoderDestroy(3dm). |
| DMstatus **dmG728DecoderGetParams** ( DMG728decoder *handle*, DMparams **params* ) | Get the parameter values of a DMG728decoder structure. See also dmG728DecoderGetParams(3dm). |
| DMstatus **dmG728DecoderSetParams** ( DMG728decoder *handle*, DMparams **params* ) | Set the parameter values of a DMG728decoder structure. See also dmG728DecoderSetParams(3dm). |
| DMstatus **dmG728DecoderReset** ( DMG728decoder *handle* ) | Fill internal buffers of a DMG728decoder structure with zeroes. See also dmG728DecoderReset(3dm). |
| DMstatus **dmG728Encode** ( DMG728encoder *handle*, short **ibuf*, unsigned char **obuf*, int *nsamples* ) | Do G.728 compression (LD-CELP). See also dmG728Encode(3dm). |

**Table A-6 (continued)** The G.728 Audio Compression Library API

| Function | Description |
|---|---|
| DMstatus **dmG728EncoderCreate** ( DMG728encoder **encoder* ) | Allocate a new DMG728encoder structure. See also dmG728EncoderCreate(3dm). |
| DMstatus **dmG728EncoderDestroy** ( DMG728encoder *handle* ) | Deallocate a DMG728Encoder structure. See also dmG728EncoderDestroy(3dm). |
| DMstatus **dmG728EncoderGetParams** ( DMG728encoder *handle*, DMparams **params* ) | Get the parameter values of a DMG728encoder structure. See also dmG728EncoderGetParams(3dm). |
| DMstatus **dmG728EncoderReset** ( DMG728encoder *handle* ) | Fill the internal buffers of a DMG728encoder structure with zeroes. See also dmG728EncoderReset(3dm). |

## The GSM Audio Compression Library

This library implements the European Global System for Mobile telecommunication (GSM) 06.10 provisional standard used for digital cellular phones. The standard, prI-ETS 300 036, describes full-rate speech transcoding which uses RPE–LTP (Regular–Pulse Excitation Long–Term Predictor)) coding at 13 Kb/s.

**Table A-7** The GSM Audio Compression Library API

| Function | Description |
|---|---|
| DMstatus **dmGSMDecode** ( DMGSMdecoder *handle*, unsigned char **ibuf*, short **obuf*, int *numSamples* ) | Do GSM decoding. See also dmGSMDecode(3dm). |
| DMstatus **dmGSMDecoderCreate** ( DMGSMdecoder **decoder* ) | Allocate a new DMGSMdecoder structure. See also dmGSMDecoderCreate(3dm). |
| DMstatus **dmGSMDecoderDestroy** ( DMGSMdecoder *handle* ) | Deallocate a DMGSMdecoder structure. See also dmGSMDecoderDestroy(3dm). |
| DMstatus **dmGSMDecoderGetParams** ( DMGSMdecoder *handle*, DMparams **params* ) | Get the parameter values of a DMGSMdecoder structure. See also dmGSMDecoderGetParams(3dm). |
| DMstatus **dmGSMDecoderReset** ( DMGSMdecoder *handle* ) | Fill internal buffers of a DMGSMdecoder structure with zeroes. See also dmGSMDecoderReset(3dm). |
| DMstatus **dmGSMEncode** ( DMGSMencoder *handle*, short **ibuf*, unsigned char **obuf*, int *numSamples* ) | Do GSM encoding. See also dmGSMEncode(3dm). |
| DMstatus **dmGSMEncoderCreate** ( DMGSMencoder **encoder* ) | Allocate a new DMGSMencoder structure. See also dmGSMEncoderCreate(3dm). |
| DMstatus **dmGSMEncoderDestroy** ( DMGSMencoder *handle* ) | Deallocate a DMGSMEncoder structure. See also dmGSMEncoderDestroy(3dm). |

**Table A-7 (continued)** The GSM Audio Compression Library API

| Function | Description |
|---|---|
| DMstatus **dmGSMEncoderGetParams** ( DMGSMencoder *handle*, DMparams **params* ) | Get the parameter values of a DMGSMencoder structure. See also dmGSMEncoderGetParams(3dm). |
| DMstatus **dmGSMEncoderReset** ( DMGSMencoder *handle* ) | Fill internal buffers of a DMGSMencoder structure with zeroes. See also dmGSMEncoderReset(3dm). |

## The MPEG-1 Audio Compression Library

This library implements the Moving Pictures Experts Group MPEG-1 audio standard. The standard is designed for encoding non-interlaced material and is optimized for single-speed CD-ROM bit rates (about 1.5 Mb / s). The compression is based on subband coding with adaptive quantization. Input data is divided into different frequency bands which are weighted by their perceptual importance. Mono and stereo sources are supported at sampling rates of 32, 44.1 and 48 kHz. Allowable bit rates range from 32 to 448 Kb / s.

**Table A-8** The MPEG-1 Audio Compression Library API

| Function | Description |
|---|---|
| DMstatus **dmMPEG1AudioDecode** ( DMMPEG1audiodecoder *decoder*, unsigned char **cmpData*, short **output*, int *fmtBytes* ) | Decode a single compressed block of data created by a call to **dmMPEG1AudioEncode()**. See also dmMPEG1AudioDecode(3dm). |
| DMstatus **dmMPEG1AudioDecoderCreate** ( DMMPEG1audiodecoder **decoder* ) | Allocate a new DMMPEG1audiodecoder structure. See also dmMPEG1AudioDecoderCreate(3dm). |
| DMstatus **dmMPEG1AudioDecoderDestroy** ( DMMPEG1audiodecoder *decoder*) | Deallocate an DMMPEG1audiodecoder structure. See also dmMPEG1AudioDecoderDestroy(3dm). |
| DMstatus **dmMPEG1AudioDecoderGetParams** ( DMMPEG1audiodecoder *decoder*, DMparams **params* ) | Get the parameter values for a DMMPEG1audiodecoder structure. See also dmMPEG1AudioDecoderGetParams(3dm). |
| DMstatus **dmMPEG1AudioDecoderSetParams** ( DMMPEG1audiodecoder *decoder*, DMparams **params* ) | Set the parameter values for a DMMPEG1audiodecoder structure. See also dmMPEG1AudioDecoderSetParams(3dm). |
| DMstatus **dmMPEG1AudioDecoderReset** ( DMMPEG1audiodecoder *handle* ) | Fill the internal buffers of an DMMPEG1audiodecoder structure with zeros. See also dmMPEG1AudioDecoderReset(3dm). |

**Table A-8 (continued)** The MPEG-1 Audio Compression Library API

| Function | Description |
|---|---|
| DMstatus **dmMPEG1AudioEncode** ( DMAudioRateConverter *encoder*, short **sampBuf*, unsigned char **output*, int *frameBytes* ) | Compress a single block of audio data using MPEG-1 audio compression algorithm. See also dmMPEG1AudioEncode(3dm). |
| DMstatus **dmMPEG1AudioEncoderCreate** ( DMMPEG1audioencoder **encoder* ) | Allocate a new DMMPEG1audioencoder structure. See also dmMPEG1AudioEncoderCreate(3dm). |
| DMstatus **dmMPEG1AudioEncoderDestroy** ( DMMPEG1audioencoder *encoder*) | Deallocate a DMMPEG1audioencoder structure. See also dmMPEG1AudioEncoderDestroy(3dm). |
| DMstatus **dmMPEG1AudioEncoderGetParams** ( DMMPEG1audioencoder *encoder*, DMparams **params* ) | Get the parameter values for an DMMPEG1audioencoder structure. See also dmMPEG1AudioEncoderGetParams(3dm). |
| DMstatus **dmMPEG1AudioEncoderSetParams** ( DMMPEG1audioencoder *encoder*, DMparams **params* ) | Set the parameter values for an DMMPEG1audioencoder structure. See also dmMPEG1AudioEncoderSetParams(3dm). |
| DMstatus **dmMPEG1AudioEncoderReset** ( DMMPEG1audioencoder *handle* ) | Fill the internal buffers of a DMMPEG1audioencoder with zeros. See also dmMPEG1AudioEncoderReset(3dm). |
| DMstatus **dmMPEG1AudioFilterStateCreate** ( DMMPEG1audiofilterstate **filterState*) | Allocate a new DMMPEG1audiofilterstate structure. See also dmMPEG1AudioFilterSateCreate(3dm). |
| DMstatus **dmMPEG1AudioFilterStateDestroy** ( DMMPEG1audiofilterstate *filterState*) | Free a DMMPEG1audiofilterstate structure. See also dmMPEG1AudioFilterSateDestroy(3dm). |
| DMstatus **dmMPEG1AudioFilterStateRestore** ( void **coder*, DMMPEG1audiofilterstate *filterState* ) | Restore a DMMPEG1audiofilterstate structure. See also dmMPEG1AudioFilterSateRestore(3dm). |

**Table A-8 (continued)** The MPEG-1 Audio Compression Library API

| Function | Description |
|---|---|
| DMstatus **dmMPEG1AudioFilterStateSave** ( void **coder*, DMMPEG1audiofilterstate *filterState* ) | Save a DMMPEG1audiofilterstate structure. See also dmMPEG1AudioFilterSateSave(3dm). |
| DMstatus **dmMPEG1AudioHeaderGetBlockBytes** ( DMMPEG1audiodecoder *decoder*, unsigned char **cmpData*, int **blockSize* ) | Get the expected length in bytes of a compressed data block. See also dmMPEG1AudioHeaderGetBlockBytes(3dm). |
| DMstatus **dmMPEG1AudioHeaderGetParams** ( unsigned char **cmpData*, DMparams **params* ) | Get decoder parameter information from the header of a compressed MPEG-1 audio data block. See also dmMPEG1AudioHeaderGetParams(3dm). |

## The Audio Rate Conversion Library

This library enables the sampling rate conversion of single-channel, 32-bit, floating point audio data.

**Table A-9** The Audio Rate Conversion Library API

| Function | Description |
|---|---|
| DMstatus **dmAudioRateConvert** ( DMaudiorateconverter *handle*, float **inbuf*, float **outbuf*, int *inlen*, int **numout* ) | Convert the data sampling rate. See also dmAudioRateConvert(3dm). |
| DMstatus **dmAudioRateConverterCreate** ( DMaudiorateconverter **converter* ) | Allocate a new DMaudiorateconverter structure. See also dmAudioRateConverterCreate(3dm). |
| DMstatus **dmAudioRateConverterDestroy** ( DMaudiorateconverter *handle*) | Deallocate an DMaudiorateconverter structure. See also dmAudioRateConverterDestroy(3dm). |
| DMstatus **dmAudioRateConverterGetParams** ( DMaudiorateconverter *handle*, DMparams **params* ) | Get the parameter values of a DMaudiorateconverter structure. See also dmAudioRateConverterGetParams(3dm). |
| DMstatus **dmAudioRateConverterSetParams** ( DMaudiorateconverter *handle*, DMparams **params* ) | Set the parameter values of a DMaudiorateconverter structure. See also dmAudioRateConverterSetParams(3dm). |
| DMstatus **dmAudioRateConverterReset** ( DMaudiorateconverter *handle*, float *resetval* ) | Fill the internal buffers of a DMaudiorateconverter structure with a constant value. See also dmAudioRateConverterReset(3dm). |

# Index

## B



## C



## D

## E

## F



## G



## H



## I



## J



## L

## Q



## R



## S



## T



## U



## V

## W



## X



## Z

## Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-1799-050.

Thank you!

## Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389