# IRIS Digital Media Programming Guide

Document Number 007-1799-040

#### **CONTRIBUTORS**

Written by Patricia Creek; Part III written by Carolyn Curtis Illustrated by Patricia Creek, Dany Galgani, Cheri Brown, David Bertrand, and Dan Young

Edited by Nancy Schweiger and Christina Cary Production by Derrald Vogt and Chris Everett

Engineering contributions by John Barco, Brian Beach, Don Bennett, David Bertrand, Mark Callow, Wiltse Carpenter, Andrew Cherenson, Doug Cook, Jonathan Devine, Grant Dorman, Dan Fink, Ron Fischer, Jeff Glover, Brian Hill, Bryan James, Bruce Karsh, Robert Keller, Eva Manolis, Ted Marsh, Spencer Murray, Paul Ning, Candace Obert, Gordon Oliver, Chris Pirazzi, Scott Porter, Mike Portuesi, Scott Pritchett, Amit Shoham, Paul Spencer, Dave Story, Archer Sully, Ann Sydeman, Alex Tang, Mike Travis, I-Ching Wang, Jim Wanslow, and Jim Wiggins.

Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson, Erik Lindholm, and Kay Maitz

© Copyright 1994, Silicon Graphics, Inc.— All Rights Reserved
This document contains proprietary and confidential information of Silicon
Graphics, Inc. The contents of this document may not be disclosed to third parties, copied, or duplicated in any form, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

#### RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and/or in similar or successor clauses in the FAR, or in the DOD or NASA FAR Supplement. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94039-7311.

Silicon Graphics, Indigo, IRIS, and the Silicon Graphics logo are registered trademarks and CHALLENGE, Cosmo Compress, Galileo Video, GL, Graphics Library, Image Vision Library, IndigoVideo, Indigo², Indigo² Video, Indy, Indy Cam, Indy Video, IRIS GL, IRIS Graphics Library, IRIS Indigo, IRIS InSight, IRIX, OpenGL, Personal IRIS, Sirius Video, Showcase, and VINO are trademarks of Silicon Graphics, Inc. Aware and the Aware logo are registered trademarks and AudioPlayback, AudioProducer, AudioPublisher, AudioSuite, Archiver, Audition, BrowsFX,

IRIS Digital Media Programming Guide Document Number 007-1799-040 MultiRate, Psycoder, and Speed-of-Sound are trademarks of Aware, Inc. Betacam and Sony are registered trademarks and Hi-8mm is a trademark of Sony Corporation. Macintosh is a registered trademark and AppleTalk and QuickTime are trademarks of Apple Computer, Inc. MII is a trademark of Panasonic, Inc. Network License System and NetLS are trademarks of Apollo Computer Inc., a subsidiary of Hewlett-Packard Company. Prosonus is a registered trademark of Prosonus. MIPS and R3000 are registered trademarks of MIPS Technologies, Inc. Open Software Foundation is a registered trademark and OSF/Motif is a trademark of the Open Systems Foundation. S-VHS is a trademark of JVC, Inc. UNIX is a trademark of AT&T Bell Labs. X Window System is a trademark of Massachusetts Institute of Technology.

IRIS Digital Media Programming Guide Document Number 007-1799-040

# Contents

List of Examples xxiii	
List of Figures xxvii	
List of Tables xxxi	
About This Guide xxxv	
What This Guide Contains xxxv	
How to Use This Guide xxxix	
Where to Start xxxix	
Style Conventions xxxix	
How to Use the Sample Programs xxxix	
Suggestions for Further Reading xl	
References for Using Digital Media with Other Libraries	d
References for Adding a User Interface to Your Program	xli

### PART ONE Digital Media Programming

1.

Programming with the IRIS Digital Media Development Environment 5
About the Digital Media Library 6
About the Digital Audio and MIDI Libraries 6
About the Video Library 7
About the IndigoVideo Library 8
About the Compression Library 8
About the Movie Library 9

2.	Programming with the Digital Media Library 13
	Digital Media Library Basics 13
	Digital Media Type Definitions 13
	Digital Media Parameters 15
	Compiling and Linking a Digital Media Library Application 15
	Debugging a Digital Media Library Application 16
	Initializing a Digital Media Application 16
	Creating and Destroying Parameter-value Lists 17
	Creating Default Audio and Image Configurations 18
	Setting and Getting Individual Parameter Values 25
	Manipulating Parameter-value Lists 29
	Synchronizing Digital Media 33
PART TWO	Digital Audio and MIDI Programming
3.	Introduction to Digital Audio and MIDI Programming 39
4.	Digital Audio System Architecture 43
	Indigo Audio System Architecture 43
	Indigo Audio Features 43
	Indigo Audio I/O Interface 44
	Indigo <sup>2</sup> and Indy Audio System Architecture 46
	Indigo <sup>2</sup> and Indy Audio I/O Interface 46
	Indy Workstation Layout 47
	4-channel Audio I/O Interface 49
	Recommendations for Audio Development System Configurations 50
	Memory 50
	Disk Space 50
	Peripherals 51
5.	Digital Audio System Software 55
	Digital Audio System Software Overview 55
	About the Digital Audio Libraries 56
	About Shared System-Wide Resources 57

Tools Available for the Audio Application Developer 60 Graphical User Interface Audio Tools 60 Online Source Code Examples 61 Third-party Audio Software and Sound Libraries Compiling and Linking an Audio Application 62 Programming with the Audio Library 67 Audio Library Basics 68 Audio Library Features 68 Audio Library Programming Model 68 Digital Audio Data Representation Handling Audio Library Errors 72 Audio Library Application Programming Concepts 73 Initializing an Audio Library Application 74 About ALports 74 Using ALconfig Structures to Configure ALports 74 Opening and Closing Audio Ports Reading and Writing Audio Data Using Audio Sample Queues Reading and Writing Samples 93 Detecting Errors in the Audio Stream 97 Querying and Controlling the Global Audio Device State 98 Techniques for Working with Global Parameters Sample Code for Querying Features and Values 107 Audio Library Synchronization Facilities Audio Sample Frame Count 112 Relating Audio Sample Frame Count to UST 116 Real-time Programming Techniques for Audio 119 Multiplexing Synchronous I/O 119 Using Scheduling Control to Give Audio High Priority 122 Preventing Memory Swapout 122 Creating Multiple Process Threads Using Shared Arenas and Semaphores 123

6.

#### 7. Programming with the Audio File Library 127

Audio File Library Basics 128

Audio File Library Programming Model 128

Handling Audio File Library Errors 128

About Audio Files 129

Creating and Configuring Audio Files 132

Creating an Audio File Setup 132

Initializing Audio File Format 133

Initializing Audio Track Data 134

Initializing Instrument Data 140

Initializing Miscellaneous Data 141

Using the Audio Utility Library to Initialize Parameter Lists 143

Opening, Closing, and Updating Audio Files 147

Opening an Audio File 147

Getting an IRIX File Descriptor for an Audio File 148

Closing and Updating Files 149

Reading and Writing Audio Track Information 150

Getting Audio File Format 150

Getting and Setting Audio Track Parameters 152

Seeking, Reading, and Writing Audio Track Frames 160

Reading and Writing Instrument Configurations 163

Handling Miscellaneous Data Chunks 168

Audio File Library Programming Tips 171

Minimizing Data and File Format Dependence 171

Preventing Concurrent Access from Multiple Threads 172

Handling Errors in Multithreaded Applications 176

Sample Audio File Program 177

#### 8. Programming with the CD Audio Library 183

- CD Audio Library Basics 184
  - CD Frames, Samples, and Subcodes 184
  - CD Tracks, Indices, and Time Codes 186
  - CD Seeking, Reading, and Playing 186
  - CD Parser 186

Opening and Closing the CD-ROM Device 187

Controlling the CD-ROM Drive Caddy 187

Navigating through a CD 187

Getting CD Locations from the End User 188

Getting CD Locations from Calculations Internal to

Your Application 188

Getting the Current CD Location 189

Seeking to a CD Location 189

Using the CD-ROM Drive 190

Playing an Audio CD from the CD-ROM Drive 190

Reading Audio Data from the CD-ROM Drive 191

Controlling the CD Parser 192

Communicating CD Status to the End User 194

CD Time Code Conversion Routines 195

CD Sample Program 196

#### 9. Programming with the DAT Audio Library 203

DAT Audio Library Basics 203

DAT Frames, Samples, and Subcodes 204

DAT Audio Program Numbers and Indices 205

DAT Run Time, Absolute Time, and Program Time 205

DAT Seeking and Reading 205

DAT Parser 205

Opening and Closing the DAT Device for Audio 206

Navigating through a DAT 206
Getting DAT Locations from the End User 207
Getting DAT Locations from Calculations Internal to Your Application 207
Seeking to a DAT Location 208
Using the DAT Drive 209
Playing a Tape in the DAT Drive 209
Making DAT Recordings for Playback on the DAT Drive 210
Reading Audio Data from the DAT Drive 211
Writing Audio Data to the DAT Drive 211

Controlling the DAT Parser 213

Communicating DAT Status to the End User 217

DAT Sample Program 217

Playing a DAT 217

### 10. Programming with the MIDI Library 223

MIDI System Architecture 224

Configuring Your System for MIDI Development 224

Connecting Devices to MIDI I/O Interfaces 227

Configuring Serial Ports for MIDI WIth the Port Setup Tool 229

MIDI Library Basics 231

Initializing MIDI Library Programs 231

Compiling and Linking MIDI Library Programs 231

MIDI Library Error Handling 231

MIDI Library Programming Model 232

Opening and Closing MIDI Ports 232

Getting the Name of an Available MIDI Port 232

Opening and Closing MIDI Input and Output Ports 233

Programming MIDI I/O 234

Hands-On MIDI Output Experience 234

About MIDI Events 235

Sending and Receiving MIDI Events 236

Printing MIDI Events 237

Processing MIDI Event Messages 238

Multiplexing MIDI I/O with File Descriptors 240
Hands-On Multiplexed MIDI I/O Experience 240
Getting a File Descriptor for a MIDI Port 240
Controlling MIDI Timing 241
Controlling MIDI Timing Mode 241
Controlling MIDI Tempo 243
Controlling MIDI Output Buffering 243
Hands-On MIDI File Player Experience 244
Synchronizing MIDI I/O with Other Media 246
Hands-On MIDI and Audio Synchronization Experience 246

#### PART THREE Video Programming

## 11. Video Basics 251

Interlacing 251

Broadcast Standards 253

Color Encoding 254

RGB 254

YUV 255

YIQ 255

YC, YC-358, YC-443, or S-Video 256

Composite Video 256

Video Signals 257

Videotape Formats 257

#### 12. Getting Started with the Video Library 261

VL Features 262

How the VL Works with Hardware 262

How the VL Works with Other Software 262

VL System Software Architecture 263

Video Daemon 264

Generic Video Tools 265

Library and Header Files 266

VL Architectural Model of Video Devices 267

VL Programming Model 269

Opening a Connection to the Video Daemon and Setting up a Data Path 271

Opening a Connection to the Video Daemon 271

Specifying Nodes on the Data Path 272

Creating and Setting Up the Data Path 273

Setting Parameters for Data Transfer to or from Memory 279

Setting Source Node Controls for Data Transfer 280

Setting Drain Node Controls for Data Transfer 284

Displaying Video Data Onscreen 293

Transferring Video Data to and from Devices 294

Creating a Buffer for the Frames 295

Registering the Ring Buffer 296

Starting Data Transfer 296

Reading Data from the Buffer 298

Ending Data Transfer 302

VL Examples 303

Simple Screen Application 304

Video-to-memory Frame Grab 307

Memory-to-video Frame Output 310

Continuous Frame Capture 314

#### 13. Using VL Controls 321

VL Control Type and Values 327

VL Control Fraction Ranges 328

VL Control Classes 328

VL Control Groupings 329

Galileo Video Controls 331

General Controls for Galileo Video 332

Galileo Video IndyCam Controls 336

Galileo Video Encoder and Color-Space Conversion Controls 337

Galileo 601 Video Digital Breakout Box Controls 338

VINO Controls 341

VINO Video Control Panel Controls 341

VINO Analog Input Controls 342

VINO IndyCam Controls 343

#### 14. VL Event Handling 347

Querying VL Events 348
Creating a VL Event Loop 350
Creating a Main Loop with Callbacks 351

#### **15. VL Blending** 361

The VL Key Generator 362
The VL Blend Node 363
VL Blending Controls 366
VL Keying 367
Galileo Video Luma Keying 368
Galileo Video Chroma Keying 370
Galileo Video Fades, Tiles, and Wipes 371
VL Blending Examples 375
Blending Video and Graphics 375
Creating a Simple Wipe Effect 376

#### PART FOUR IndigoVideo Programming

# Introduction to IndigoVideo Programming 381Using the IndigoVideo Examples 381References for Video Programming 382

#### 17. Getting Started with the IndigoVideo Library 385

IndigoVideo Basics 385 IndigoVideo 385 IndigoVideo Data Formats 386 IndigoVideo I/O 389

A Simple Program for Getting Started with IndigoVideo 392

### 18. Controlling the IndigoVideo Input Window Setting Input Parameters 399 Selecting an Input Source 400 Selecting the Input Signal Type 401 Freezing and Restarting Video Input 401 Querying Video Parameters 401 Positioning and Scaling the Video Input 402 Setting the Size of the Video Image Positioning the Video Image 405 Preventing Other Programs from Using Video 406 Combining Video and Graphics 406 Video Underlay Mode 407 Video Overlay Mode and Chroma Keying 19. **Producing IndigoVideo Output** 423 Selecting the IndigoVideo Live Output Area 423 Setting Output Parameters 426 Turning Output On and Off 426 Synchronizing Output with Input 426 Filtering Output 426 Generating Single-frame Output 426 20. Capturing Video from IndigoVideo 433 Captured Video Data Formats 434 Capturing a Single Video Frame 436 Capturing Video Frames in Burst Mode 439 Capturing Video Frames in Continuous Mode Entering Continuous Capture Mode 442

Accessing Captured Data 442

Using Data Conversion Routines 444
Converting YUV Data to RGB 444
Using 8-bit RGB Capture Data 449

Leaving Continuous Capture Mode 444

#### 21. Handling IndigoVideo Events 455

IndigoVideo Event Handling Basics 456 X Event Handling 458 IRIS GL Event Handling 462

#### 22. Using the IndigoVideo Utilities 467

Using *svcmd*, the IndigoVideo Shell-level Tool 468
Making a Movie File from IndigoVideo and Audio Input 469

#### PART FIVE Compression Programming

#### 23. Introduction to the Compression Library 475

Overview of the Compression Library 475
Compression Library Applications 476
Compression Library Features 478
Compression Library Basics 478
Compression Library Algorithms 479

Compression Library Data Formats 483

Audio Data Formats 483 Image Data Formats 483 Video Data Formats 484 Movie Data Formats 487 Header Formats 488

#### 24. Getting Started with the Compression Library 491

Overview of the Compression Library API 492

Still Image API 492 Sequential Access API 492 Buffered Access API 492

About File I/O and Error Handling 493

Using the Still Image Interface 494

Using the Sequential Frame Interface 497

Compressing a Sequence of Frames 497

Decompressing a Sequence of Frames 500

25.

26.

Using the Buffering Interface 506
Creating a Buffer 507
Managing Buffers 509
Producing and Consuming Data in Buffers 511
Creating a Buffered Record and Play Application 514
Creating Buffered Multiprocess Record and Play Applications 519
Programming with the Cosmo Compress JPEG Codec 521
Cosmo Compress Basics 521
Cosmo Compress Image Formats 522
Getting Compressed Image Information 524
Memory-to-Memory Compression and Decompression 525
Compressing and Decompressing Video Through External Connections to Cosmo Compress 527
Controlling JPEG Compressed Image Quality 532
Using Compression Library Algorithms and Parameters 537
Using the Compression Library Algorithms 537
Choosing a Compression Library Algorithm 537
Querying Compression Library Algorithms 540
Using the Compression Library Parameters 543
Compression Library Parameter Definitions 544
Setting and Querying Compression Library Parameters 551
Using Frame Type Parameters 559
Customizing the Compression Library 563
Adding Custom Algorithms to the Compression Library 563
Managing Buffers for Added Algorithms 568
Reading Data Across Buffer Discontinuities 568
Adding Custom Parameters to the Compression Library 572
rading Custom I didiffered to the Compression Library 3/2

PART SIX	Movie Programming
27.	Introduction to the Movie Library 579
	Overview of Movie Library Features and Applications 579
	Movie Library Features 579
	Movie Library Applications 580
	Using the Movie Library with Other Silicon Graphics Libraries 581
28.	Getting Started with the Movie Library 585
	Movie Library Basics 585
	Definitions 585
	Movie Library Programming Model 587
	Movie File Formats 587
	Developing a Movie Library Application 588
	Outline for Developing a Movie Library Application 588
	Compiling and Linking a Movie Library Application 589
	Debugging a Movie Library Application 590
	Setting and Getting Movie Properties 591
	Setting and Getting Movie and Track Parameters 592
	Setting and Getting Global Movie Properties 593
	Creating a Default Movie Configuration 596
	Adding Your Own Parameters to the Movie Library 598
	Setting and Getting Track Properties 601
	Setting and Getting General Track Properties 601
	Setting and Getting Audio Track Properties 603
	Setting and Getting Image Track Properties 606
29.	File I/O and Editing Movies with the Movie Library 615
	Initializing a Movie Library Application 615
	Using File Descriptors with Movies 617

Creating a New Movie 617

Opening an Existing Movie 620

Checking for the Presence of a Movie 619

30.

Adding, Locating, and Deleting Audio and Image Tracks 622 Adding an Audio or Image Track to a Movie 622 Removing an Audio or Image Track from a Movie 624 Locating an Existing Track 624 Mapping Frames from One Track to Another Track 624 Editing Movies 626 Optimizing a Movie File 627 Using a Buffer for Editing 627 Deleting Frames from a Movie Track 630 Reading and Inserting Compressed Images 631 Copying and Pasting Frames from One Movie into Another 633 Finalizing Changes and Closing Movies 636 Playing Movies with the Movie Library 639 Opening a Movie for Playback 640 Creating and Configuring a Playback Window Creating a Window for OpenGL Playback Creating a Window for IRIS GL Playback 641 Configuring the Playback Display 643 Binding a Movie to a Window for Playback 648 Binding a Window to a Movie with an Audio Track 649 Playing Multiple Movies in the Same Window Controlling Movie Playback 651 Starting and Stopping Playback 651 Controlling Audio Playback 651 Looping 653 Playing or Looping a Movie Fragment 657 Scrubbing to a Random Frame During Playback 658 Synchronizing Movie Playback 659 Getting and Setting the Playback Speed Measuring the Current Frame Rate 660 Setting and Getting a Minimum Playback Speed Threshold 660

Forcing Playback of Every Frame 661

Integrating Movies with IRIS GL Graphics 662

Controlling the Frame Display Automatically 662

Controlling the Frame Display Manually 665

Handling Events 668

Preparing an Event Mask 669

Getting a File Descriptor for the Movie Event Queue 670

Creating the Event Loop 671

Handling Movie Events 672

Handling X Window Events 676

Checking and Correcting for Slow Playback 678

#### 31. Using the Movie Library with QuickTime Movies 681

QuickTime Basics 681

QuickTime Sound 682

QuickTime Compression 682

QuickTime Frame Differencing (Keyframes) 684

Movie Library QuickTime Compatibility Requirements 685

Making a Single-fork Movie 685

Making a Self-contained Movie 686

Transferring Files Between Macintosh and

Silicon Graphics Computers 687

Adding QuickTime Capability to Your Movie Library Application 688

Using the QuickTime Compressor Library 688

Creating a QuickTime Movie 688

Reading Existing QuickTime Movies 693

#### 32. Using the Movie Library Sample Programs 701

About the Sample Programs 701

Creating Movies 705

Creating a Movie from a Sequence of Images 708

Adding or Replacing a Movie Audio Track 708

Editing Movies 709

Displaying Movie Parameters 712

A.

В.

Playing Movies 712 Creating a Simple Keyboard Interface for Playing Movies 712 Playing Multiple Movies 713 Creating a Movie Screensaver Application 714 Using the SMPTE Time Code Sample Application 717 Converting a SMPTE Time Code String to a Frame Number Converting a Frame Number to a SMPTE Time Code String 718 Converting a Time Specification to a Frame Number 719 Converting a Frame Number to a Time Code 720 Audio Specifications 723 Indigo Workstation Audio Hardware Specifications 723 Indigo Analog Audio I/O 724 Indigo Digital Audio I/O 725 Indigo Dedicated Real-time Processor 726 Indigo<sup>2</sup> Workstation Audio Hardware Specifications 726 Indigo<sup>2</sup> Analog Stereo Line-level Inputs 726 Indigo<sup>2</sup> Stereo Microphone Input 727 Indigo<sup>2</sup> Analog Stereo Line-level Outputs 727 Indigo<sup>2</sup> Analog Stereo Headphone Output/Mono Internal Speaker 727 Indigo<sup>2</sup> Digital Audio I/O 728 Aware Scalable Audio Compression Software 729 Introduction to Aware Audio Compression Software 729 Aware Software Products Features and Applications Aware Products Available in IRIS Digital Media Libraries 731 Other Digital Media Compatible Aware Audio Products 732 Accessing Aware Audio Compression from the Audio File Library 733 Valid Audio Input Data 733 Compression Defaults 733

Compression Custom Configuration 734

Accessing Aware Audio Compression from the Compression Library 736

Compression Schemes 736

Using Compression Library Parameters 737

Usage Hints 740

Aware Audio Compression Software Specifications 741

Installing a NetLS Nodelocked License 743

Glossary 745

Index 779

# List of Examples

Example 2-1	Creating and Destroying a Parameter-value List 18
Example 2-2	Setting Audio Defaults 21
Example 2-3	Setting Image Defaults 25
Example 2-4	Setting Individual Parameter Values 28
Example 2-5	Printing the Contents of a Digital Media Parameter-value List 32
Example 6-1	Configuring and Opening an ALport 76
Example 6-2	Opening Input and Output ALports 89
Example 6-3	Querying for the Existence of Other Audio Processes 108
Example 6-4	Querying for Input and Output Rates 109
Example 6-5	Querying for 4-channel Capability 111
Example 6-6	Synchronizing Audio Between Two Output Ports: <i>align.c</i> 115
Example 6-7	Calculating UST 117
Example 7-1	Creating, Filling, Querying and Freeing an AUpvlist 146
Example 7-2	Checking Audio Track Sample Format and Sample Width 154
Example 7-3	Creating a Semaphore 174
Example 7-4	Recording Stereo from an Audio Port: recordexample.c 177
Example 8-1	Copying CD Data to an Audio File: cdsample.c 196
Example 9-1	Reading DAT Samples 218
Example 10-1	Opening MIDI Input and Output Ports 233
Example 10-2	Sending a MIDI Message 235
Example 10-3	Using MIDI File Descriptors 241
Example 12-1	Sending Live Video to the Screen: simplev2s.c 304
Example 12-2	Video Frame Grabbing: simplegrab.c 307
Example 12-3	Frame Output: simplem2v.c 310
Example 12-4	Continuous Frame Capture: <i>simplecapt.c</i> 314

Example 14-1	Using VL Callbacks 352
Example 14-2	VL Event Handling: eventex.c 353
Example 15-1	Setting Up Source, Drain, and Blend Nodes 363
Example 17-1	Opening a Window to Display Live Video Input: <i>simpleinput,c</i> 393
Example 18-1	Setting up the IndigoVideo Board for PAL Input 400
Example 18-2	Selecting a Video Input Source 400
Example 18-3	Getting the Input Source Number 401
Example 18-4	Creating a Scalable Video Input Window: sizeinput.c 403
Example 18-5	Approximating the Requested Video Window Size 405
Example 18-6	Specifying a Video Window Offset 405
Example 18-7	Getting Exclusive Use of the IndigoVideo Board 406
Example 18-8	Using IndigoVideo Underlay Mode 408
Example 18-9	Using Chroma Keying to Key Out Black Pixels 410
Example 18-10	Using IndigoVideo Overlay Mode: voverlay.c 411
Example 18-11	Using the Chroma Key Map: chromamap.c 414
Example 19-1	Setting the Location of the IndigoVideo Output Window 424
Example 19-2	Aligning a Video Output Area with an IRIS GL Window 425
Example 19-3	Sending a RGB Image as a Still Video Frame 427
Example 20-1	Determining the Capture Buffer Size 436
Example 20-2	Grabbing a Single Frame of 8-bit RGB data: rgbgrab.c 437
Example 20-3	Using the SV_GET_FIELD Macro 440
Example 20-4	Capturing Frames in Burst Mode 441
Example 20-5	Initializing Continuous Capture Mode 442
Example 20-6	Accessing and Releasing Captured Frames 443
Example 20-7	Setting Top-to-Bottom pixmode for YUV 445
Example 20-8	Finding Image Data in YUV with Blanking Frames 445
Example 20-9	Grabbing YUV Frames to Save as RGB Images: vgrab.c 446
Example 20-10	Interleaving 8-bit RGB Fields with Inversion 450
Example 20-11	Displaying Interleaved 8-bit RGB Data 450
Example 20-12	Converting 8-bit RGB Capture Data to 32-bit RGB 450

Example 20-13	Setting up the IRIS GL Color Map to Display 8-bit RGB 451
Example 21-1	X Event Handling for IndigoVideo events: <i>xevents.c</i> 459
Example 21-2	Handling Video Events with IRIS GL Routines 463
Example 24-1	Using a Custom Error Handling Routine 494
Example 24-2	Compressing and Decompressing a Single Frame 496
Example 24-3	Compressing a Series of Frames 500
Example 24-4	Getting the Decompression Scheme from a Header 502
Example 24-5	Decompressing a Series of Frames 505
Example 24-6	Creating and Using an Internal Buffer 508
Example 24-7	Creating and Using an External Buffer 508
Example 24-8	Using Buffers for Playback 514
Example 24-9	Using Buffers for Nonblocking Playback 515
Example 24-10	Using Buffers for Recording 516
Example 24-11	Using Buffers for Nonblocking Recording 517
Example 24-12	Using Buffers for Multiprocess Playback 519
Example 24-13	Using Buffers for Multiprocess Recording 520
Example 24-14	Cosmo Compress Memory-to-Memory Compression 526
Example 25-1	Getting a List of Compression Library Algorithms 542
Example 25-2	Getting a List of Parameters for a Compressor/Decompressor 553
Example 25-3	Getting the Current Values of Selected Parameters 555
Example 25-4	Using Macros to Get or Set the Value of a Floating Point Parameter 556
Example 25-5	Getting and Setting Parameter Defaults 557
Example 25-6	Getting and Setting Minimum and Maximum Parameter Values 558
Example 26-1	Adding Algorithms to the Compression Library 567
Example 26-2	Decompression Buffering 568
Example 26-3	Compression Buffering 568
Example 26-4	Adding Parameters to the Compression Library 573
Example 28-1	Creating and Initializing a Default Movie Parameter-value List 597
Example 28-2	Adding a User-Defined Global Movie Parameter 599

Example 28-3	Adding a User-Defined Image Track Parameter for a New Track 600
Example 29-1	Creating a Movie 619
Example 29-2	Adding an Audio Track to a Movie 623
Example 29-3	Determining What Size Buffer to Allocate 628
Example 29-4	Reading a Compressed Image from a Movie into a Buffer 632
Example 30-1	Creating an IRIS GL Playback Window 642
Example 30-2	Binding a Window for Playing Multiple Movies 650
Example 30-3	Enabling and Muting Audio Playback 652
Example 30-4	Designating a Movie as the Primary Audio Rate Controller 653
Example 30-5	Setting and Getting the Loop Mode 655
Example 30-6	Using mvGrabIrisGL() and mvReleaseIrisGL() 663
Example 30-7	Initializing Movie Playback 667
Example 30-8	Preparing a File Descriptor Set 673
Example 30-9	Handling Movie Frame, Stop, and Error Events 675
Example 30-10	Handling X11 Expose and Resize Window Events 677
Example 31-1	Creating QuickTime Movies with the Movie Library 689
Example 31-2	Converting QuickTime Picture Data to RGBX Format 694

# List of Figures

Audio Icons 44
Audio Jacks on the Back Panel of the Indigo Workstation 45
Audio Jacks on the Back Panel of the Indigo <sup>2</sup> Workstation 47
Volume Control Buttons on the Front of the Indy Workstation 48
Audio Jacks on the Back Panel of the Indy Workstation 48
Cabling Setup for 4-channel Audio on the Indy Workstation 49
Interaction of Digital Audio System Components 55
Audio Data Flow 58
Audio Samples and Frames 70
Audio Sample Queues 91
Sample Frame Count as Returned by <b>ALgetframenumber()</b> 114
Using Fill Points 121
Audio Data Packing Formats 153
CD Audio Sample Structure 185
DAT Audio Sample Structure 204
MIDI Setup 226
Serial Ports on the Back Panel of the Indigo Workstation 227
Serial Ports on the Back Panel of the Indigo <sup>2</sup> Workstation 228
Serial Ports on the Back Panel of the Indy Workstation 228
Port Setup Icon 229
Port Setup Tool 229
Serial Port Connections 230

Figure 10-8	MIDI Port Configuration 230
Figure 11-1	Fields and Frame 252
Figure 11-2	Relationships Between Color-encoding Methods and Video Formats 256
Figure 11-3	Composite Video Waveform 257
Figure 12-1	VL System Components 263
Figure 12-2	Simple VL Path 267
Figure 12-3	Simple VL Blending 268
Figure 12-4	Zoom and Decimation 288
Figure 12-5	Clipping an Image 290
Figure 12-6	Zoom, Size, Offset, and Origin 292
Figure 12-7	vlGetNextValid(), vlGetLatestValid(), and vlPutFree() 299
Figure 15-1	Setting Up the Blend Node 364
Figure 15-2	Galileo Video Alpha Blender 364
Figure 15-3	Blending Analog Video with Part of a Graphics Screen 365
Figure 15-4	Blending Analog Video with Static Frame Data 365
Figure 15-5	Adding Another Drain to Preview the Blend 365
Figure 15-6	Luma Keying Application: Titling 368
Figure 15-7	Relationships Between Galileo Video Luma Keying Controls 369
Figure 15-8	Chroma Keying Application: TV Weather Map 370
Figure 15-9	Relationships Between Galileo Video Chroma Keying Controls 371
Figure 15-10	Galileo Video Keying Controls 374
Figure 17-1	Format of 32-bit RGB Pixels 387
Figure 17-2	Format of 8-bit RGB Pixels 387
Figure 17-3	Format of YUV Data Words 388
Figure 17-4	IndigoVideo I/O Ports 389
Figure 17-5	Connecting Video Equipment to the Indigo Video Board 390
Figure 23-1	Server-Client Compression Applications 477
Figure 24-1	Ring Buffer 506

Figure 24-2	Snapshots of Buffer State During Producing and Consuming Processes 512
Figure 24-3	Flow of Data in a Buffered Compression and Decompression Scheme 513
Figure 26-1	Buffer Architecture for Adding Algorithms 571
Figure 28-1	Typical Movie: somersault.mv 586
Figure 29-1	Movie Library File I/O Routines 616
Figure 29-2	Mapping Frames from One Track to Another 625
Figure 29-3	Inserting Frames into a Track 629
Figure 29-4	Pasting Image Frames from One Movie into Another Movie 634
Figure 30-1	Playback View Settings 643
Figure 32-1	Comments in Movie Library Sample Programs: createmovie.c++ 703
Figure 32-2	Modularity of Movie Library Sample Programs: createmovie.c++ 704
Figure 32-3	Call Graph for createmovie 707
Figure 32-4	Call Graph for <i>editmovie.c</i> 711
Figure 32-5	Call Graph for <i>moviescreen.c</i> 716
Figure GI-1	SMPTE Color Bars (75%) 750
Figure GI-2	Color Burst and Chrominance Signal 751
Figure GI-3	Component Video Signals 753
Figure GI-4	Horizontal Blanking 759
Figure GI-5	Horizontal Blanking Interval 760
Figure GI-6	Waveform Monitor Readings with and without Setup 768
Figure GI-7	SMPTE Time Code 768
Figure GI-8	Red or Blue Signal 773
Figure GI-9	Y or Green Plus Sync Signal 773
Figure GI-10	Video Waveform: Composite Video Signal With Setup (Typical NTSC) 774
Figure GI-11	Video Waveform: Composite Video Signal (Typical PAL) 775

# List of Tables

Table 2-1	Digital Media Parameter Types 14
Table 2-2	Audio Parameters 19
Table 2-3	Audio Defaults 20
Table 2-4	Image Parameters 22
Table 2-5	Image Defaults 24
Table 2-6	DM Library Routines for Setting Parameter Values 26
Table 2-7	DM Library Routines for Getting Parameter Values 27
Table 2-8	Routines for Manipulating Parameter-value Lists and Entries 29
Table 2-9	Methods for Obtaining Unadjusted System Time 33
Table 6-1	Minimum and Maximum Allowable Queue Sizes for ALports 79
Table 6-2	Input Conversions for <b>ALreadsamps()</b> 95
Table 6-3	Output Conversions for <b>ALwritesamps()</b> 96
Table 6-4	Error Parameters for <b>ALgetstatus()</b> 98
Table 6-5	Core Global Parameters for AL_DEFAULT_DEVICE 99
Table 6-6	Special Global Parameters for System-Dependent Audio Capabilities 101
Table 6-7	Global Parameter Name Strings 105
Table 7-1	Mapping of AF Library Components to AIFF-C/AIFF File Chunks 131
Table 7-2	AFfilesetup Parameters and Defaults 132
Table 7-3	AFfilesetup Instrument Parameter Constants and Defaults 133
Table 7-4	Settable Compression Parameter Values and Types 138
Table 7-5	Miscellaneous Chunk Types and Parameter Values 142
Table 7-6	Audio Utility Library Set and Get Routines 144

Table 7-7	Valid Return Values for Compression Algorithms and Parameters 157
Table 7-8	Instrument Parameter Constants and Valid Values 164
Table 10-1	MIDI Message Status Bytes 238
Table 11-1	Tape Formats and Video Formats 258
<b>Table 12-1</b>	Header Files for Video Options 266
Table 12-2	Video Library Calls for Data Transfer 271
<b>Table 12-3</b>	VL Event Masks 278
Table 12-4	Data Transfer Controls for Source Nodes 280
Table 12-5	VL_MUXSWITCH Values 281
Table 12-6	Default Sources for VINO Inputs 282
Table 12-7	VINO Timing Choices 283
Table 12-8	Dimensions for Timing Choices 283
Table 12-9	Data Transfer Controls for Drain Nodes 284
Table 12-10	Packing Types and Their Sizes and Formats 287
Table 12-11	VL_RATE Values (Items per Second) 292
Table 12-12	Buffer-Related Calls 298
Table 12-13	Calls for Extracting Data from a Buffer 299
Table 13-1	Device-Independent VL Controls 322
Table 13-2	VL Control Groupings 330
Table 13-3	Galileo Video <i>vcp</i> Controls 332
Table 13-4	Galileo Video IndyCam Controls 336
Table 13-5	Galileo Video Encoder and Color-Space Conversion Controls 337
Table 13-6	Galileo 601 Video Digital Breakout Box General Controls 338
Table 13-7	Galileo Video Digital Breakout Box Color-Space Conversion Controls 339
Table 13-8	Galileo Video DAC controls 340
Table 13-9	VINO vcp Controls 341
Table 13-10	VINO Analog Input Controls 342
Table 13-11	IndyCam Controls 343
Table 14-1	VL Event Handling Routines 348
Table 14-2	VL Event Masks 349

Table 15-1	VL Blend Controls 366
Table 15-2	Galileo Video Luma Keying Controls 369
Table 15-3	Galileo Video Chroma Keying Controls 370
Table 15-4	Controls for Fades, Tiles, and Wipes 372
Table 15-5	Galileo Video Controls Specific to Wipes 373
Table 20-1	Pixel Sizes for Video Data 435
Table 20-2	Fields in the svCaptureInfo Structure 435
Table 21-1	Video Activity Event Variable Names 456
Table 21-2	Video Parameter Change Event Variable Names 457
Table 21-3	Encoding Attribute Values 458
Table 23-1	Video Formats Not Requiring Color Conversion 487
Table 23-2	Parameters Contained in Header Data 488
Table 24-1	Typical Stream Header Contents 502
Table 24-2	Additional Video Stream Header Contents 502
Table 24-3	Cosmo Compress Image Format Parameters 523
Table 24-4	Cosmo Compress Video Field Dimensions 529
Table 24-5	Cosmo Compress Field Widths for Compression With Decimation 530
Table 24-6	Cosmo Compress Field Widths for Decompression 531
Table 25-1	Capabilities of Image and Video Algorithms 540
Table 25-2	Compression Library Parameters 549
Table 28-1	Movie Defaults 597
Table 28-2	Audio Defaults 603
Table 28-3	Image Defaults 607
Table 28-4	Image Packing Formats 610
Table 30-1	Movie Library Events 669
Table 30-2	Event Structure Fields 670
Table 32-1	SMPTE Time Code Types 717
Table B-1	Built-in Algorithms for Aware Audio Software Compression Engines 742
Table GI-1	Videotape Formats 772

### **About This Guide**

The *IRIS*<sup>®</sup> *Digital Media Programming Guide* describes the Silicon Graphics<sup>®</sup> IRIS Digital Media Development Environment software. The IRIS Digital Media Development Environment (DMdev) provides an application programming interface (API) for working with digital audio, MIDI, video, compression, and movies, using standard and optional Silicon Graphics workstation hardware and peripherals.

Silicon Graphics also supplies desktop media tools for end users, which are built on top of DMdev. Media tools are described in the online *Media Control Panels User's Guide*, which you can view from the IRIS InSight<sup>TM</sup> viewer.

#### **What This Guide Contains**

The *IRIS Digital Media Programming Guide* is divided into six parts, corresponding to the functions of the libraries:

Part I, "Digital Media Programming," has two chapters:

- Chapter 1, "Programming with the IRIS Digital Media Development Environment," gives an overview of the IRIS Digital Media Development Environment.
- Chapter 2, "Programming with the Digital Media Library," describes
  the Digital Media (DM) Library, libdmedia, a library that currently
  supports parameter setting and ring buffering for applications that use
  the DMdev. Currently, you can use the DM routines with the Movie
  Library and the Video Library.

Part II, "Digital Audio and MIDI Programming," has eight chapters;

 Chapter 3, "Introduction to Digital Audio and MIDI Programming," introduces the digital audio and MIDI libraries.

- Chapter 4, "Digital Audio System Architecture," gives a brief overview of the audio hardware and provides some recommendations for development configurations.
- Chapter 5, "Digital Audio System Software," describes the audio application programming environment and explains how audio resources are shared.
- Chapter 6, "Programming with the Audio Library," describes the structure of the Audio Library and explains how to use it to sample audio data from analog or digital input sources. Real-time programming techniques are also discussed.
- Chapter 7, "Programming with the Audio File Library," describes the structure of the Audio File Library and explains how to use it to read and write audio files.
- Chapter 8, "Programming with the CD Audio Library," describes the CD Audio Library and explains how to use it to control the CD-ROM drive for playing and sampling audio from audio compact discs.
- Chapter 9, "Programming with the DAT Audio Library," describes the DAT Audio Library and explains how to use it to control the DAT drive for playing, sampling, and recording audio from digital audio tape.
- Chapter 10, "Programming with the MIDI Library," describes connecting MIDI equipment and describes the MIDI Library, explaining how to use it for implementing and multiplexing MIDI I/O, and synchronizing MIDI and audio.

#### Part III, "Video Programming," has five chapters:

- Chapter 11, "Video Basics," explains basic video concepts that apply to both the Video Library and the IndigoVideo Library.
- Chapter 12, "Getting Started with the Video Library," describes the Video Library and explains how to use it to perform video input and output for workstations equipped with standard and optional Silicon Graphics video hardware.
- Chapter 13, "Using VL Controls," describes how to use VL controls to set video parameters for data transfer and video effects.
- Chapter 14, "VL Event Handling," describes how to handle video events using the Video Library.

Part IV, "IndigoVideo Programming," has six chapters:

- Chapter 16, "Introduction to IndigoVideo Programming," introduces the IndigoVideo library and gives an overview of the features of the IndigoVideo board.
- Chapter 17, "Getting Started with the IndigoVideo Library," describes basic concepts for using the IndigoVideo board, and presents a sample video application that displays live video input in a window.
- Chapter 18, "Controlling the IndigoVideo Input Window," explains
  how applications can position and scale the video input. It also explains
  how to select different video sources, formats, and broadcast standards.
- Chapter 19, "Producing IndigoVideo Output," explains how to encode a portion of your screen to video in real time. This chapter also covers single-frame output.
- Chapter 20, "Capturing Video from IndigoVideo," explains how to capture frames of video to memory.
- Chapter 21, "Handling IndigoVideo Events," explains how to handle video events, such as video parameters being changed by another process.
- Chapter 22, "Using the IndigoVideo Utilities," explains how to use the IndigoVideo end-user tools.

Part V, "Compression Programming," has four chapters:

- Chapter 23, "Introduction to the Compression Library," introduces the CL and describes its applications and features. It provides basic background information on compression technology and on digital audio and video data formats.
- Chapter 24, "Getting Started with the Compression Library," describes how to use the three types of interfaces supplied by the CL and how to write programs for Cosmo Compress option.
- Chapter 25, "Using Compression Library Algorithms and Parameters," explains how to use the CL algorithms and global parameters.
- Chapter 26, "Customizing the Compression Library," explains how to add your own algorithms and parameters to the CL.

Part VI, "Movie Programming," has six chapters:

- Chapter 27, "Introduction to the Movie Library," introduces the Movie Library and describes its applications and features.
- Chapter 28, "Getting Started with the Movie Library," explains how to set up, compile, and debug Movie Library applications.
- Chapter 29, "File I/O and Editing Movies with the Movie Library," explains how to perform movie file I/O and how to edit movies.
- Chapter 30, "Playing Movies with the Movie Library," describes the Movie Library playback and event-handling facilities.
- Chapter 31, "Using the Movie Library with QuickTime Movies," describes basic concepts for working with QuickTime movies, and then it explains how to add QuickTime capability to a Movie Library application. It also describes the optional QuickTime compressor Library, which provides access to QuickTime compressors for Movie Library applications.
- Chapter 32, "Using the Movie Library Sample Programs," describes the Movie Library sample programs.

Appendices at the back of this guide provide additional information:

- Appendix A, "Audio Specifications," lists relevant audio and video hardware specifications.
- Appendix B, "Aware Scalable Audio Compression Software," explains how to incorporate into your application the built-in licensable compression software by Aware<sup>®</sup>, Inc.

The Glossary at the end of this guide provides definitions for video terms.

#### How to Use This Guide

This guide is written for C language programmers. This guide assumes that you are somewhat knowledgable about digital media concepts. The discussion of each library begins by presenting the features, applications, and basic concepts pertaining to that library. Readers unfamiliar with the basic concepts can refer to the recommended references for each topic.

#### Where to Start

If you're not sure which library to use for a certain application, read Chapter 1, "Programming with the IRIS Digital Media Development Environment," 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 *List of Examples* to locate a code fragment or 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/people/4Dgifts/examples/dmedia directory, which is further organized in directories according to topic. For example, Movie Library programs are in /usr/people/4Dgifts/examples/dmedia/movie.

You must log in as *4Dgifts* to be able to compile *4Dgifts* programs. *README* files in each *4Dgifts* 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 4Dgifts 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 $^{\text{TM}}$ , IRIS Graphics Library $^{\text{TM}}$  (GL) or X Window System $^{\text{TM}}$  application, you may want to consult the following manuals:

- OpenGL Programming Guide and OpenGL Reference Guide, by Jackie Neider, Tom Davis, and Mason Woo, Addison-Wesley, 1993
- 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: Xt 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 User'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

#### **PART ONE**

# Digital Media Programming

Chapter 1, "Programming with the IRIS Digital Media Development Environment," gives an overview of the IRIS Media Libraries.

Chapter 2, "Programming with the Digital Media Library," describes the Digital Media (DM) Library, *libdmedia*, a library that currently supports parameter setting and ring buffering for applications that use the IRIS digitial media libraries. Currently, you can use the DM routines with the Movie Library and the Video Library.

# Chapter 1

# Programming with the IRIS Digital Media Development Environment

This chapter introduces the IRIS digital media libraries, briefly outlining their uses and features.

# Programming with the IRIS Digital Media Development Environment

The IRIS Digital Media Development Environment provides a digital media software development environment that includes audio, video, movie, and compression libraries.

This chapter provides an overview of the uses and features of these libraries:

- Digital Media Library, a base library that provides global type definitions and utility routines for digital media applications; it currently supports parameter setting and ring buffering
- Digital Audio and MIDI Libraries, a collection of libraries that provides an API for working with digital audio, audio files, digital compact disc, digital audio tape, and MIDI
- Video Library, a device-independent API for programming Silicon Graphics on-board video and video options
- Indigo Video Library, an API for programming the Indigo Video option for IRIS Indigo workstations equipped with Entry graphics
- Compression Library, an extensible, algorithm-independent API for compressing and decompressing audio, video, and images
- Movie Library, a file-format-independent API for reading, writing, playing, and editing movies

You can use these libraries in conjunction with other Silicon Graphics libraries, such as the ImageVision Library; see the individual library descriptions to learn which libraries are compatible.

# **About the Digital Media Library**

The Digital Media (DM) Library, *libdmedia.so*, is a library that currently supports parameter setting and ring buffering for applications that use the IRIS Digital Media software. Currently, you can use the DM routines with the Movie Library and the Video Library.

The DM Library features:

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

# **About the Digital Audio and MIDI Libraries**

Silicon Graphics offers a collection of libraries designed for developers of digital audio and MIDI software, as well as those seeking to integrate audio into their existing applications:

- Audio Library (libaudio.a)
- Audio File Library (libaudiofile.so)
- Audio Utility Library (libaudioutil.so)
- CD Audio Library (libcdaudio.a)
- DAT Audio Library (libdataudio.a)
- MIDI Library (*libmd.so*)

The digital audio libraries can be used separately or in combination. Each library is tailored to a particular set of tasks, as follows:

Audio Library Provides an API for configuring the audio system,

managing audio I/O between the application program and the audio hardware, specifying attributes of digital audio data, and facilitating real-time programming. See Chapter 6, "Programming

with the Audio Library."

Audio File Library Provides an API for reading and writing two

standard digital audio file formats, AIFF and AIFF-C. See Chapter 7, "Programming with the

Audio File Library."

Audio Utility Library Provides convenience routines for creating and configuring Audio File Library data structures.

CD Audio Library Provides an API for optional Silicon Graphics SCSI

CD-ROM drives. The drive features a special mode that allows it to read audio CD format as well as CD-ROM format. See Chapter 8, "Programming

with the CD Audio Library."

DAT Audio Library Provides an API for optional Silicon Graphics SCSI

DAT drives. See Chapter 9, "Programming with the

DAT Audio Library."

# **About the Video Library**

The Video Library (VL) is a collection of device-independent C language calls for Silicon Graphics workstations equipped with video options, such as Sirius Video TM, Indigo Video TM, or Galileo Video TM, or workstations equipped with on-board video, such as Indy TM.

The VL provides generic video tools, including simple tools for importing and exporting digital data to and from current and future Silicon Graphics video hardware, as well as to and from third-party video devices that adhere to the Silicon Graphics architectural model for video devices. Video tools are described in the *Media Control Panels User's Guide*, which you can view using the IRIS InSight viewer; similar applications are supplied in source-code

form as examples in the *4Dgifts* directory (/usr/people/4Dgifts/examples/dmedia/video/vl).

The VL provides an API that enables applications to:

- perform video teleconferencing on platforms that support it
- blend computer graphics with frames from videotape or any video source
- present video in a window on the workstation screen
- digitize video data

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

### About the IndigoVideo Library

The IndigoVideo board provides video input and output for IRIS Indigo workstations equipped with Entry graphics. The IndigoVideo Library provides a software interface to the IndigoVideo board, enabling applications to:

- display live video in a window
- capture live video to system memory
- encode graphics to video in real time
- produce high-quality single-frame video output

# **About the Compression Library**

The Compression Library, *libcl.so*, provides a flexible, extensible, and algorithm-independent software interface for compressing and decompressing audio, video, and image data.

The Compression Library features:

- algorithm independence
- hardware independence

- support of industry standard algorithms
- support of Silicon Graphics proprietary algorithms
- binary compatibility across Silicon Graphics platforms

The Compression Library provides facilities for working with audio, still images, sequential frames of data (movies), and a buffering mechanism for nonsequential compression and decompression.

You can query the Compression Library for the available algorithms, and you can add your own algorithms and parameters. A pass-through capability allows you to pass data through the routines without using an algorithm.

The Compression Library can be used with the Audio File Library, and with data used by the IRIS Movie Player and Movie Maker tools.

# **About the Movie Library**

The Movie Library, *libmovie*, is a collection of routines that provides a C language API for reading, writing, editing, and playing movies on Silicon Graphics workstations. The API provides a uniform interface to movies of various formats and lets you convert movies from one format to another.

The Movie Library features:

- the ability to read, write, and play movie files
- a file-format-independent API
- file format conversion capabilities
- support for Silicon Graphics Movie format, versions 2.0 and 3.0
- support for Apple® Computer QuickTime™ movie format
- data compression and decompression
- asynchronous playback support
- flexible playback control
- support for movies embedded in applications software

# Chapter 2

# Programming with the Digital Media Library

This chapter describes the Digital Media Library, which currently supports parameter setting and ring buffering for digital media applications.

# Programming with the Digital Media Library

The Digital Media (DM) Library, *libdmedia.so*, is a library that provides type definitions for digital media and currently supports parameter setting and ring buffering for applications that use the IRIS digital media libraries.

This chapter contains basic concepts for working with the Digital Media Library. It describes the digital media data types and explains how to use the digital media parameters.

# **Digital Media Library Basics**

It is not likely that you'll use the DM Library by itself. Typically, you call DM Library routines from an application that is written using one or more of the IRIS digital media libraries. Currently, you can use the DM routines with the Movie Library and the Video Library.

The DM Library features:

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

#### **Digital Media Type Definitions**

The DM Library provides type definitions for digital media. 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:

 $DM boolean \qquad integer for conditionals; DM\_FALSE is 0 and DM\_TRUE is 1$ 

DMfraction integer numerator divided by integer denominator
DMstatus enumerated type consisting of DM\_SUCCESS and

DM\_FAILURE

Table 2-1 lists the digital media parameter type definitions that are defined in *dmedia/dm\_params.h*.

 Table 2-1
 Digital Media Parameter Types

Parameter Type	Meaning
DM_TYPE_ENUM	Enumerated type
DM_TYPE_ENUM_ARRAY	Array of enumerated types
DM_TYPE_INT	Integer value
DM_TYPE_INT_ARRAY	Array of integers
DM_TYPE_INT_RANGE	Range of integers
DM_TYPE_STRING	String
DM_TYPE_STRING_ARRAY	Array of strings
DM_TYPE_FLOAT	Floating point value
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_PARAMS	Parameter-value list
DM_TYPE_TOC_ENTRY	Table-of-contents entry for ring buffers

#### **Digital Media Parameters**

Parameter-value lists are used to store configuration information for movies, movie tracks, ring buffers, and video paths. A parameter-value list is a list of pairs, where each pair contains the name of a parameter and the corresponding value for that parameter.

Typical ways in which you might use a parameter-value list include:

- passing a parameter-value list to a routine that configures a structure
- passing a parameter-value list that contains new parameter settings to a routine that changes the settings
- using convenience routines provided by one of the IRIS digital media libraries to set and get parameter values that apply to that library

Every parameter-value 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 and DM\_AUDIO.

The routines described in this chapter follow the general rule that ownership of data is not passed during procedure calls, except in the routines that create and destroy parameter-value 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.

#### 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>
```

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\_d.so*, by using **-ldmedia\_d** instead of **-ldmedia**, and then run your program. 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.

# **Initializing a Digital Media Application**

This section explains how to use the DM Library routines for:

- creating and destroying parameter-value lists
- creating default audio and image configurations
- setting and getting values in parameter-value lists
- manipulating parameter-value lists

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

- 1. Create an empty parameter-value 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. See "Creating Default Audio

- and Image Configurations" on page 18 for a description of this method.
- Use a generic function such as **dmParamsSetInt()** to set the values of individual parameters within an empty parameter-value list or one that has already been initialized with the standard audio or image configuration. See "Setting and Getting Individual Parameter Values" on page 25 for a description of this method.
- Use a library function such as mvSetMovieDefaults() to set a group of parameters specific to that library. See "Creating a Default Movie Configuration" in Chapter 28 for a discussion of this method.
- 3. Free the parameter-value list and its contents by calling dmParamsDestroy().

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

### **Creating and Destroying Parameter-value Lists**

Some libraries require you to allocate memory for parameter-value lists, but with the DM library, you need not allocate memory for parameter-value 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 parameter-value list, set the parameter value(s) and use them, and then destroy the structure, freeing its associated memory.

dmParamsCreate() is the only function that can create a parameter-value list, and dmParamsDestroy() is the only function that can free one. This means that parameter-value 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 parameter-value 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 parameter-value list, you must destroy it to free the associated memory. To free both the parameter-value list structure and its contents, call **dmParamsDestroy()**. Its function prototype is:

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

params is a pointer to the parameter-value list you want to destroy

Example 2-1 is a code fragment that creates a parameter-value list called *params*, then calls a Movie Library routine, **mvSetDefaults()**, to initialize the default movie parameters, and then destroys the list, freeing both the structure and its contents.

#### **Example 2-1** Creating and Destroying a Parameter-value List

```
DMparams* params;
if ( dmParamsCreate( &params ) != DM_SUCCESS ) {
    printf( "Out of memory.\n" );
    exit( 1 );
}
if ( mvSetMovieDefaults(params, MV_FORMAT_SGI_3) !=
DM_SUCCESS ) {
    printf( "Out of memory.\n" );
    exit( 1 );
}
dmParamsDestroy ( params );
```

# **Creating Default Audio and Image Configurations**

There are standard parameters that apply to images (for video and movies) and standard parameters that apply to audio (for movies). This section explains how to use the DM Library convenience routines that initialize parameter-value lists for standard audio and image configurations.

#### **Audio Parameters**

Audio uses these parameters:

- 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)

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

 Table 2-2
 Audio Parameters

Parameter	Туре	Values
DM_AUDIO_CHANNELS	Integer	1, 2, or 4
DM_AUDIO_COMPRESSION	String	DM_AUDIO_UNCOMPRESSED (default) DM_AUDIO_G711_U_LAW DM_AUDIO_G711_A_LAW DM_AUDIO_MPEG DM_AUDIO_MPEG1 DM_AUDIO_MULTIRATE DM_AUDIO_G722
DM_AUDIO_FORMAT	DMaudioformat	DM_AUDIO_TWOS_COMPLEMENT (default) DM_AUDIO_UNSIGNED DM_AUDIO_FLOAT 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

See Part II, "Digital Audio and MIDI Programming," for complete definitions of the audio parameters.

See "Setting and Getting Audio Track Properties" in Chapter 28 for a description of audio parameters that apply to Movie Library programs.

#### **Setting Audio Defaults**

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

where:	
params	is a pointer to a parameter-value 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: 1, 2, or 4

**dmSetAudioDefaults()** returns DM\_SUCCESS if there was enough memory available to set up the parameters; otherwise, it returns DM\_FAILURE.

Table 2-3 lists the parameters and values set by **dmSetAudioDefaults()**.

Table 2-3Audio 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 parameter-value 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 2-2 is a code fragment that creates a parameter-value list, fills in the audio defaults, and then frees the structure and its contents.

#### **Example 2-2** Setting Audio Defaults

#### **Image Parameters**

Images use these parameters:

- image compression scheme
- image dimensions (width and height)
- image interlacing
- image orientation (top-to-bottom vs. bottom-to-top)
- image packing format
- image rate (number of frames per second)

Table 2-4 lists the image parameters and the valid values for each (not all values are supported by all libraries).

Table 2-4Image Parameters

Parameter	Values
DM_IMAGE_HEIGHT	Integer value
DM_IMAGE_WIDTH	Integer value
DM_IMAGE_RATE	Floating point value
DM_IMAGE_COMPRESSION	DM_IMAGE_UNCOMPRESSED DM_IMAGE_RLE DM_IMAGE_RLE24 DM_IMAGE_JPEG DM_IMAGE_MPEG1 DM_IMAGE_MVC1 DM_IMAGE_MVC2 DM_IMAGE_RTR DM_IMAGE_HDCC DM_IMAGE_QT_VIDEO DM_IMAGE_QT_ANIM
DM_IMAGE_INTERLACING	DM_IMAGE_NONINTERLACED (full frame) DM_IMAGE_INTERLACED_EVEN (two fields, even field first) DM_IMAGE_INTERLACED_ODD (two fields, odd field first) DM_IMAGE_NONINTERLEAVED (obsolete, use DM_IMAGE_NONINTERLACED instead) DM_IMAGE_INTERLEAVED(obsolete, use DM_IMAGE_INTERLACED_ODD instead)

Table 2-4 (continued)	Image Parameters
Parameter	Values
DM_IMAGE_ORIENTATION	ON DM_TOP_TO_BOTTOM
	DM_BOTTOM_TO_TOP
DM_IMAGE_PACKING	DM_PACKING_RGB
	DM_PACKING_RGBX
	DM_PACKING_RGBA
	DM_PACKING_RGB332 (Indigo Entry graphics)
	DM_PACKING_RGB8
	DM_PACKING_GRAYSCALE
	DM_PACKING_YUV
	DM_PACKING_YUV411
	DM_PACKING_YUV422
	DM_PACKING_YUV422HC
	DM_PACKING_APPLE_32
	DM_PACKING_APPLE_16
	DM_PACKING_Y (equivalent to
	DM_PACKING_GRAYSCALE)
	DM_PACKING_YCbCr (equivalent to
	DM_PACKING_YUV)
	DM_PACKING_YCbCr422 (equivalent to
	DM_PACKING_YUV422)
	DM_PACKING_YCbCr422HC (equivalent to
	DM_PACKING_YUV422HC)
	DM_PACKING_YUV422DC (equivalent to
	DM_PACKING_YUV422HC)
	DM_PACKING_YCbCr422DC (equivalent to
	DM_PACKING_YUV422HC)

See "Setting and Getting Image Track Properties" in Chapter 28 for a description of image parameters that apply to Movie Library programs. See Table 12-10 in Chapter 12, "Getting Started with the Video Library," for a description of image parameters that apply to Video Library programs.

#### **Setting Image Defaults**

To initialize a parameter-value 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 parameter-value list that was returned from dmParamsCreate()

width is the width of the image

height is the height of the image

packing is the image packing format
```

Table 2-5 lists the parameters and values set by dmSetImageDefaults().

Table 2-5Image Defaults

Parameter	Default
DM_MEDIUM	DM_IMAGE
DM_IMAGE_WIDTH	width
DM_IMAGE_HEIGHT	height
DM_IMAGE_RATE	15.0
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 parameter-value 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 ( DMparams* params )
```

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

#### **Example 2-3** Setting Image Defaults

#### **Setting and Getting Individual Parameter Values**

After creating an empty parameter-value 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 parameter-value 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 2-1.

All of these functions store and retrieve entries in parameter-value lists. 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.

Table 2-6 lists the DM Library routines for setting parameter values. These routines require three arguments:

paramsa pointer to a parameter-value listparamNamethe name of the parameter whose value you want to setvaluea value of the appropriate type for the given parameter

 Table 2-6
 DM Library Routines for Setting Parameter Values

Routine	Purpose
dmParamsSetInt()	Sets the value of a parameter whose type is DMint
dmParamsSetIntArray()	Sets the value of a parameter whose type is DMintarray
dmParams SetIntRange()	Sets the value of a parameter whose type is DMintrange
dmParamsSetEnum()	Sets the value of a parameter whose type is DMenum
dm Params Set Enum Array ()	Sets the value of a parameter whose type is DMenumarray
dmParamsSetFloat()	Sets the value of a parameter whose type is DMfloat
dm Params Set Float Array ()	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 DMfract
dm Params Set Fract Array ()	Sets the value of a parameter whose type is DMfractionarray
dm Params Set Fract Range ()	Sets the value of a parameter whose type is DMfractionrange
dmParamsSetParams()	Sets the value of a parameter whose type is DMparam
dmParamsSetString()	Sets the value of a parameter whose type is DMstring

Table 2-6 (continued)	DM Library Routines for Setting Parameter Values
Routine	Purpose
dmParamsSetStringArray()	Sets the value of a parameter whose type is DMstringarray
dmParamsSetTocEntry()	Sets the value of a parameter whose type is DMTocEntry

These routines return either DM\_SUCCESS or DM\_FAILURE.

Table 2-7 lists the DM Library routines for setting parameter values. These routines require two arguments:

params a pointer to a parameter-value 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, parameter-value lists, and table-of-contents entries, the pointer that is returned points into the internal data structure of the parameter-value 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 parameter-value list, it should be copied.

 Table 2-7
 DM Library Routines for Getting Parameter Values

Routine	Purpose
dmParamsGetInt()	Returns an integer value for the given parameter
dm Params Get Int Array ()	Returns a pointer to a value of type DMintarray for the given parameter
dmParamsGetIntRange()	Returns a pointer to a value of type DMintrange for the given parameter
dmParamsGetEnum()	Returns an integer value for the given parameter
dm Params Get Enum Array ()	Returns a pointer to a value of type DMenumarray for the given parameter
dmParamsGetString()	Returns a pointer to a value of type const char for the given parameter

Table 2-7 (continued)	OM Library Routines for Getting Parameter Values
Routine	Purpose
dmParamsGetStringArray()	Returns a pointer to a value of type DMstringarray for the given parameter
dmParamsGetFloat()	Returns a value of type double for the given parameter
dm Params Get Float Array ()	Returns a pointer to a value of type DMfloatarray for the given parameter
dm Params Get Float Range ()	Returns a pointer to a value of type DMfloatrange for the given parameter
dmParamsGetFract()	Returns a value of type DMfraction for the given parameter
dmParamsGetFractArray()	Returns a pointer to a value of type DMfractionarray for the given parameter
dm Params Get Fract Range ()	Returns apointer to a value of type DMfractionrange for the given parameter
dmParamsGetParams()	Returns a pointer to a value of type DMparams for the given parameter
dmParamsGetTocEntry()	Returns a value of type DMTocEntry for the given parameter

Example 2-4 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 2-4** 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 Parameter-value Lists**

This section explains how to manipulate parameter-value lists.

Table 2-8 lists the routines that perform operations on parameter-value lists and the entries within them.

**Table 2-8** Routines for Manipulating Parameter-value Lists and Entries

Routine	Purpose
dmParamsCopyAllElems()	Copy the entire contents of one list to another
dmParamsCopyElem()	Copy one parameter-value pair from one list to another
dmParamsGetElem()	Get the name of a given parameter
dmParamsGetElemType()	Get the data type of a given parameter
dm Params Get Num Elems ()	Get the number of parameters in a list
dmParamsIsPresent()	Determine if a given parameter is in the list
dmParamsRemoveElem()	Remove a given parameter from the list

The sections that follow explain how to use each routine.

#### **Determining the Number of Elements in a Parameter-value List**

To perform any task that requires your application to loop through the contents of a parameter-value 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 parameter-value list, call dmParamsGetNumElems(). Its function prototype is:

```
int dmParamsGetNumElems ( 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.

#### Copying the Contents of One Parameter-value List into Another

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

```
 \begin{tabular}{ll} DMs tatus $dmParamsCopyAllElems (DMparams* from Params, DMparams* to Params) \end{tabular}
```

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.

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

If a parameter appears in more than one parameter-value 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 ( DMparamsfromParams, 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 ( DMparams* params, 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:

```
\label{eq:dmparams} $\operatorname{DMparams}^* \operatorname{params}^* \operatorname{params}^*, \ \operatorname{int} \\ \operatorname{index} \ )
```

See Table 2-1 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:

```
\label{local_problem} DM boolean \ dmParamsIsPresent \ ( \ DM params* \ params, \ const \ char* \ paramName \ )
```

DM\_TRUE is returned if *paramName* is in *params*; otherwise, DM\_FALSE is returned.

#### Removing an Element from a Parameter-value 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 2-5 prints the contents of a parameter-value list.

**Example 2-5** Printing the Contents of a Digital Media Parameter-value 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 ) );
            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" );
    }
}
```

# Synchronizing Digital Media

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

The Digital 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 IRIS 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 2-9 describes how UST information is provided by each of the libraries.

**Table 2-9** Methods for Obtaining Unadjusted System Time

Library	UST Method
Digital Media Library	dmGetUST()
Audio Library	ALgetframenumber() and ALgetframetime()
MIDI Library	mdTell() and mdSetTimestampMode()
Video Library	ustime field in the DMediaInfo structure
Compression Library	ustime field in the CLimageInfo structure

The DM Library routine, **dmGetUST()**, returns a high-resolution, unsigned 64-bit number to processes using the digital media subsystem. Typically, you use the appropriate routine for the library that handles the type of media being processed, as listed in Table 2-9, rather than **dmGetUST()**. However, **dmGetUST()** is useful for correlating UST to system time for events that are not related to a media stream, such as pushing a button or making a network connection.

#### **PART TWO**

# **Digital Audio and MIDI Programming**

Chapter 3, "Introduction to Digital Audio and MIDI Programming," introduces the digital audio and MIDI libraries.

Chapter 4, "Digital Audio System Architecture," gives a brief overview of the audio hardware and provides some recommendations for development configurations.

Chapter 5, "Digital Audio System Software," describes the audio application programming environment and explains how audio resources are shared.

Chapter 6, "Programming with the Audio Library," describes the structure of the Audio Library and explains how to use it to sample audio data from analog or digital input sources. Real-time programming techniques are also discussed.

Chapter 7, "Programming with the Audio File Library," describes the structure of the Audio File Library and explains how to use it to read and write audio files.

Chapter 8, "Programming with the CD Audio Library," describes the CD Audio Library and explains how to use it to control the CD-ROM drive for playing and sampling audio from audio compact discs.

Chapter 9, "Programming with the DAT Audio Library," describes the DAT Audio Library and explains how to use it to control the DAT drive for playing, sampling, and recording audio from digital audio tape.

Chapter 10, "Programming with the MIDI Library," describes connecting MIDI equipment and describes the MIDI Library, explaining how to use it for implementing and multiplexing MIDI I/O, and synchronizing MIDI and audio.

# Chapter 3

# Introduction to Digital Audio and MIDI Programming

This chapter outlines the features of the digital audio system and describes the audio I/O interface.

# Introduction to Digital Audio and MIDI Programming

Silicon Graphics offers a collection of libraries designed for developers of digital audio and MIDI software, as well as those seeking to integrate audio into their existing applications.

Part II, "Digital Audio and MIDI Programming," describes in detail the application programming interfaces (APIs) for these libraries, which are included in the IRIS Digital Media Development Environment:

- Audio Library (libaudio.a)
- Audio File Library (libaudiofile.so)
- Audio Utility Library (libaudioutil.so)
- CD Audio Library (libcdaudio.a)
- DAT Audio Library (libdataudio.a)
- MIDI Library (libmd.so)

Each chapter presents the digital audio and MIDI libraries from a taskoriented perspective. Chapters are organized to cover topics in roughly the order you are concerned about them as you write audio or MIDI programs. To illustrate the use of the various component libraries, sample code fragments and demonstration programs are used throughout.

Digital audio programs typically access analog or digitally recorded sound data that is either input directly to the workstation audio hardware or stored on disk, digital audio tape, or CD. The application then manipulates the data and outputs the result to analog or digital line-out jacks, to disk, or to tape. MIDI programs read, process, and produce MIDI data streams, which are in turn interpreted by MIDI devices such as synthesizers and drum machines that are distributed across a MIDI network. The libraries described in this part of this guide provide all the necessary features to create audio and MIDI applications for Silicon Graphics workstations that support audio.

Reference documentation on the digital audio and MIDI routines is contained in online reference pages. These provide a concise, thorough description of each library function and are available through the use of the *man* or *Xman* command.

This guide assumes that you're somewhat familiar with principles of digital audio and MIDI. This section lists additional references that cover background material and topics beyond the scope of this part.

Although some background material is provided in the chapters on digital audio and MIDI, you may want to do some more in-depth reading. The following texts may provide useful supplementary information:

- *AES, Journal of the Audio Engineering Society,* edited by Daniel R. von Recklinghausen, Audio Engineering Society.
- *The Art of Digital Audio*, by John Watkinson, Focal Press, 1988.
- Computer Music Journal, edited by Steven Travis Pope, MIT Press.
- Elements of Computer Music, F. Richard Moore, Prentice-Hall, 1990.
- MIDI Sequencing for Musicians, compiled by the staff of Keyboard Magazine, H. Leonard Publishing Corp., 1989.
- *MIDI Sequencing in C*, by Jim Conger, M & T Books, 1989.
- *MIDI 1.0 Detailed Specification* and *Standard MIDI Files 1.0*, International MIDI Association, 5316 W. 57th St., Los Angeles, CA 90056.
- Music Through MIDI, by Michael Boom, Microsoft Press, 1991.
- Musical Applications of Microprocessors, by Hal Chamberlin, Hayden Books, 1985.

If you plan on using the MIDI C++ classes, you may want to use the following books as references:

- The Annotated C++ Reference Manual, by Margaret Ellis and Bjarne Stroustrup, AT&T Bell Laboratories, 1990—the official C++ language reference manual.
- *C++ Primer*, by Stanley Lippman, AT&T Bell Laboratories, 1989—An introductory-level, tutorial-style presentation of C++.
- *The C++ Programming Guide*—an online manual provided with the Silicon Graphics C++ library.

# Chapter 4

# Digital Audio System Architecture

This chapter outlines the features of the digital audio system and describes the audio I/O interface.

# Digital Audio System Architecture

Before you start to program, you should familiarize yourself with your workstation's audio hardware and the peripherals with which you will be working. This chapter describes the audio capabilities and the audio I/O interfaces available on IRIS Indigo, Indigo<sup>2</sup>, and Indy workstations. This chapter also provides recommendations for minimal and optimal configurations of memory, hard disk, and other peripherals useful for audio development and testing.

See Appendix A, "Audio Specifications," and your workstation owner's guide for complete details on audio hardware features. See the online release notes for *audiodev*, the audio development environment of the IRIS Media Libraries, for information about system software requirements.

# Indigo Audio System Architecture

The standard audio hardware supplied with Indigo workstations supports 24-bit digital stereo and 16-bit analog stereo sound. A dedicated real-time processor works in tandem with the CPU to ensure that audio timing isn't degraded by other system demands.

## Indigo Audio Features

Indigo audio features include:

- built-in speaker
- stereo line-level analog input and output
- stereo headphone output
- microphone input with phantom power
- AES/EBU digital audio input and output

- sampling rates include 8000, 11025, 16000, 22050, 32000, 44100, and 48000 Hz.
- independent input and output rates
- output rate can be synchronized to the digital input rate
- low-latency operation for highly interactive applications

# Indigo Audio I/O Interface

The audio hardware interface on the back panel of Indigo workstations includes these 3.5-mm audio input and output jacks, which are labeled with icons (see Figure 4-1):



(((<del>+))</del>





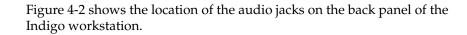


Figure 4-1 Audio Icons

- monaural microphone input jack for mic-level audio input
- stereo line-level input (line in) jack for analog audio input from a tape deck, CD player, or similar source
- stereo line-level output (line out) jack for analog audio output, for example, to a tape deck or amplifier
- stereo headphone output jack
- stereo digital I/O jack for digital audio input and output

An internal switching mechanism selects one active input source from the three available inputs. All three outputs are always enabled; each transmits a copy of the same output signal, but the volume is adjusted on the headphone/speaker output. Using the headphone jack preempts output to the internal speaker, which normally outputs the sum of the left and right signals.

Digital input and output signals are simultaneously transmitted over a stereo cable. The Audio Engineering Society (AES) standard supports mono and stereo streams of 20-bit or 24-bit samples. Each of the digital input and output streams contains two interleaved channels (left and right) of audio samples.



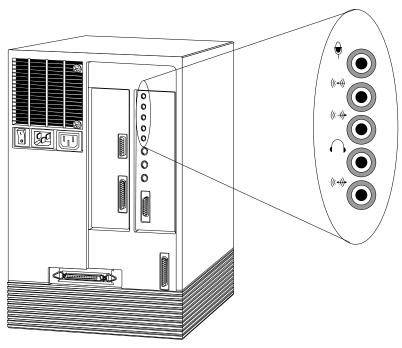


Figure 4-2 Audio Jacks on the Back Panel of the Indigo Workstation

# Indigo<sup>2</sup> and Indy Audio System Architecture

The audio hardware supplied standard with the Indigo<sup>2</sup> and Indy workstations provides the same basic audio capabilities as that of the Indigo workstation, plus:

- stereo microphone input
- 4-channel mode that supports full-speed, simultaneous 4-channel analog input and 4-channel analog output

# Indigo<sup>2</sup> and Indy Audio I/O Interface

The audio hardware interface on Indigo<sup>2</sup> and Indy workstations includes these 3.5-mm audio input and output jacks (see Figure 4-3 for the Indigo<sup>2</sup> back panel layout, and Figure 4-5 for the Indy back panel layout):

- microphone/line-in<sub>2</sub> jack for mono and stereo mic-level audio input
- stereo line-in jack for analog audio input from a tape deck, CD player, or similar source
- stereo line-out jack for analog audio output, for example, to a tape deck or amplifier
- stereo headphone/line-out<sub>2</sub> output jack
- stereo digital in/out jack for digital audio input and output

As in the Indigo workstation, all three outputs are enabled, and an internal switching mechanism selects one active input source from the three available inputs. In addition, a software-controllable internal switching mechanism permits 4-channel audio I/O through the standard I/O interface. See "4-channel Audio I/O Interface" on page 49 for details on 4-channel audio.

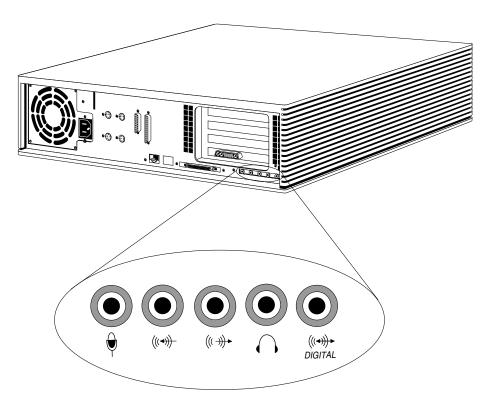


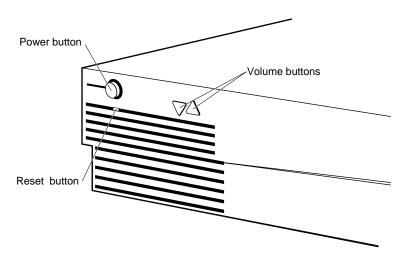
Figure 4-3 shows the location of the audio jacks on the back panel of the  $\rm Indigo^2$  workstation.

**Figure 4-3** Audio Jacks on the Back Panel of the Indigo<sup>2</sup> Workstation

# **Indy Workstation Layout**

The Indy workstation features a slightly different layout for its audio I/O interface. Two triangular pushbuttons on the front of the Indy workstation let you adjust the volume of the internal speaker/headphone output up or down, as desired. Pressing both buttons at the same time mutes the speaker/headphone output.

Figure 4-4 shows the volume control buttons on the front of the Indy workstation.



**Figure 4-4** Volume Control Buttons on the Front of the Indy Workstation

Figure 4-5 shows the location of the audio jacks on the back panel of the Indy workstation.

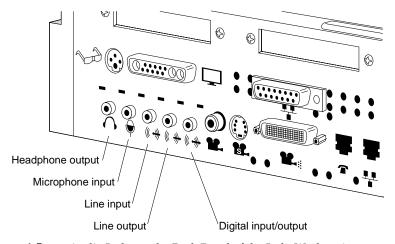
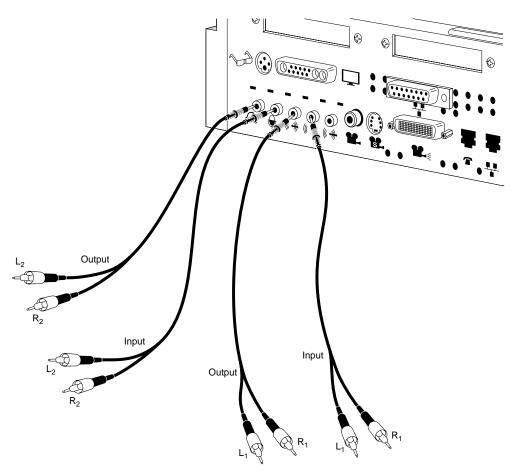


Figure 4-5 Audio Jacks on the Back Panel of the Indy Workstation

## 4-channel Audio I/O Interface

A software-controllable internal switching mechanism permits 4-channel audio I/O through the standard I/O interface. When a system is operating in 4-channel mode, the electrical properties of the microphone jack can be configured to accept either line-level or mic-level input, and the electrical properties of the headphone jack can be configured to produce line-level output.

Figure 4-6 shows an Indy workstation cabling setup for 4-channel audio.



**Figure 4-6** Cabling Setup for 4-channel Audio on the Indy Workstation

Cables like the ones shown in Figure 4-6 can be purchased from audio accessory dealers. One end of the cable has 3.5-mm audio plugs that plug into the Indigo<sup>2</sup> or Indy workstation jacks; the other end independently terminates each of the two independent signals with RCA phono plugs.

**Note:** Do not confuse these cables with "Y" connectors that route the same signal to multiple connections.

When the system is configured (either from *apanel* or from the Audio Library) to use 4-channel mode,  $(L_1, R_1)$  samples are input to the line-in jack and  $(L_2, R_2)$  samples are input to the microphone/line-in<sub>2</sub> jack. Similarly, in 4-channel mode,  $(L_1, R_1)$  samples are output from the line-out jack, and  $(L_2, R_2)$  samples are output from the headphone/line-out<sub>2</sub> jack.

# **Recommendations for Audio Development System Configurations**

The primary considerations in setting up your system for digital audio software development are memory and disk space. Because of the large sizes of audio sample files, disk space in particular is crucial.

### **Memory**

A minimum of 32 MB is recommended for digital audio development. The more memory installed, the more responsive your workstation will be when handling large amounts of sample data, as well as during compilation.

### **Disk Space**

Be sure to allow an adequate amount of disk space. These statistics should help give you an idea of the kind of disk space required for your application:

- mono 8-bit, CCITT/TSB G.711  $\mu$ -law encoded 8 kHz (speech quality) audio = 8 kBytes/sec
- mono 16-bit (15-bit range, 14-bit resolution), CCITT/TSB G.722 compressed 16 kHz (high-quality speech with more computationally expensive compression) audio = 8 kBytes/sec

- stereo 16-bit 44.1 kHz (CD-quality digitized analog input) audio = 176 kBytes/sec
- stereo 24-bit 48 kHz (highest-quality digital, 4-byte word) audio = 384 kBytes/sec
- 4-channel 16-bit 44.1 kHz (CD-quality digitized analog input) audio = 352 kBytes/sec
- 4-channel 24-bit 48 kHz (highest-quality digital, 4-byte word) audio = 768 kBytes/sec

A minimum of 600 MB is suggested; 800 MB or more is recommended, especially if your development work involves storing large amounts of high-quality sample data on disk.

# **Peripherals**

If you do not already have a CD-ROM drive, you may want to purchase one. Prosonus<sup>®</sup>, Aware<sup>®</sup>, Inc., and other companies supply CD-ROM libraries of audio sample data (see "Third-party Audio Software and Sound Libraries" in Chapter 5 for information on ordering these CD-ROM libraries). You can also use the drive for sampling from audio CDs (obtain permission before using copyrighted material).

A DAT drive is recommended both for general data archiving and for transferring audio from hard disk.

# Chapter 5

# Digital Audio System Software

This chapter describes the digital audio system software features and explains shared audio resources.

# Digital Audio System Software

This chapter describes the components of the digital audio system software: digital audio libraries, device drivers, and system-wide resources, and explains how these components interact. This chapter also describes other resources available to application developers, such as end-user audio tools, third-party audio software and sound libraries, and sample programs.

# **Digital Audio System Software Overview**

Figure 5-1 diagrams the interaction between an audio application and the audio libraries, the device drivers, the IRIX file system, the audio hardware, and the optional SCSI devices.

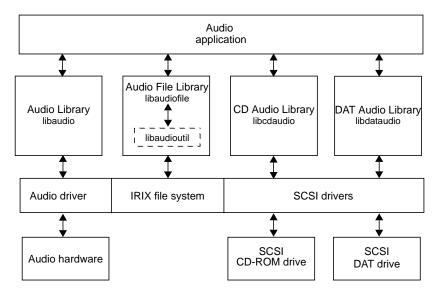


Figure 5-1 Interaction of Digital Audio System Components

# **About the Digital Audio Libraries**

The digital audio libraries can be used separately or in combination. Each library is tailored to a particular set of tasks, as follows:

provides an API for configuring the audio system, Audio Library

> managing audio I/O between the application program and the audio hardware, specifying attributes of digital audio data, and facilitating real-time programming. See Chapter 6, "Programming with the Audio Library."

Audio File Library provides an API for reading and writing two standard

digital audio file formats, AIFF and AIFF-C. See

Chapter 7, "Programming with the Audio File Library."

CD Audio Library provides an API for optional Silicon Graphics SCSI CD-

> ROM drives. The drive features a special mode that allows it to read audio CD format as well as CD-ROM format. See Chapter 8, "Programming with the CD

Audio Library."

DAT Audio provides an API for optional Silicon Graphics SCSI Library

DAT drives. See Chapter 9, "Programming with the

DAT Audio Library."

## **About Shared System-Wide Resources**

Audio applications share CPU resources with other processes, and they share audio resources with other audio applications running concurrently.

## **How Audio Applications Share CPU Resources**

CPU resources are managed by the IRIX kernel, which gives some resources higher priority than others. Programming style can affect CPU usage, so to get the best performance from your application, use native data formats whenever possible (to avoid internal conversion), and free system resources as soon as they are no longer needed (see the individual chapters on each library for details). You can also request exclusive resources or upgrade the priority of your application by using the special IRIX real-time programming techniques described in "Real-time Programming Techniques for Audio" in Chapter 6.

#### **How Audio Applications Share Audio System Resources**

Figure 5-2 shows how the IRIS audio utilities *apanel*, *soundeditor*, and *soundfiler* share the system's audio resources. Similarly, your audio application must share the audio resources with other audio applications running concurrently.

## **How Outputs from Multiple Audio Applications Are Combined**

In Figure 5-2, three audio applications are running simultaneously. A recording engineer is using *soundeditor* to combine live input from a microphone with a prerecorded sound file stored on the disk. She is using *apanel* to monitor the input level and *soundfiler* to audition sound files through her headphones.

Note that while the input is selected from among three possible inputs, all of the outputs are added together and clipped to generate the final output, which is presented to all three outputs. This means that an audio application is responsible for determining if other audio applications are running concurrently, and limiting its output signal accordingly to avoid unnecessary clipping.

Figure 5-2 shows how the IRIS audio utilities *apanel*, *soundeditor*, and *soundfiler* share the system's audio resources.

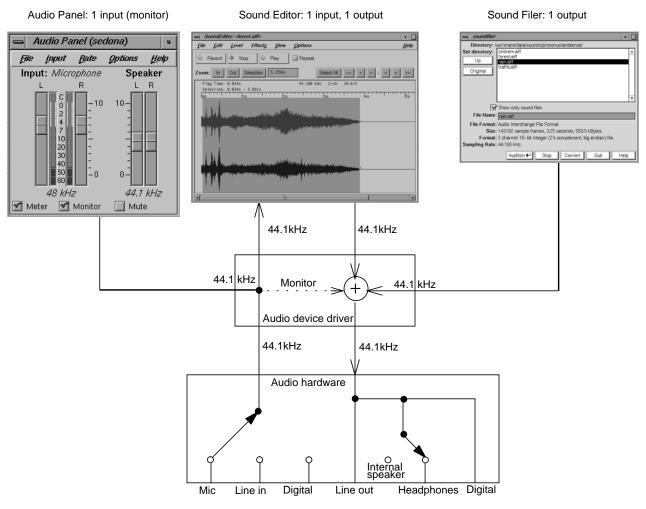


Figure 5-2 Audio Data Flow

#### How Global Audio Settings Are Established and Maintained

The system-wide digital audio hardware and software settings are initialized to reasonable defaults when the system is powered on and whenever it is rebooted.

In Figure 5-2, the input rate and output rate are set at 44.1 kHz and remain fixed unless changed from *apanel*. *soundfiler* and *soundeditor* both allow the user to control the volume from *apanel*. *soundfiler* changes the input and output rates when needed, and *soundeditor* has the ability to change the rates but asks for confirmation before making any changes.

The values of the global audio settings are known collectively as the audio system *state*. Certain audio settings can be initialized and modified in software. The AL has routines for querying which elements of state can be controlled by software, and for getting and setting the values of the global state parameters. It is good programming practice to query for the existence of other audio processes before changing global settings.

#### **Programming Guidelines for Managing System-Wide Resources**

Keep these guidelines in mind when writing audio applications:

- Determine the availability of special features before attempting to use them.
- Monitor the existence of concurrent audio applications and process output accordingly.
- Manage system-wide settings that rely on personal preference, such as volume, through a global audio control program such as *apanel*; otherwise, query for the existence of other audio processes before changing settings such as data rates that can affect other applications.
- Manage memory allocation for efficient use of system-wide resources.

# Tools Available for the Audio Application Developer

This section describes additional tools that you may find helpful for developing audio applications.

## **Graphical User Interface Audio Tools**

End-user audio tools are provided for playing, recording, and manipulating digital audio signals. These audio tools were created using the digital audio libraries and therefore support AIFF and AIFF-C file formats. These tools are provided as part of the standard system software and feature online help.

See the *Media Control Panels User's Guide* for a complete description of these tools:

apanel the audio control panel for selecting inputs, input and

output levels, and sampling rates

cdman for playing audio CDs on a CD-ROM through your

workstation's audio outputs, and for recording CD audio

tracks to disk

datman for playing and recording digital audio tapes using the

optional internal DAT drive, and for recording DAT audio

tracks to disk

soundeditor a simple editor for viewing, manipulating, and combining

multiple tracks of recorded samples, as live input or from a

sound file

soundfiler an audio file librarian for organizing and previewing

sample sound files and converting between different sound

file formats

In addition, the system Toolchest contains a tool for performing confidence tests on system components, including the audio system, and the CD-ROM and DAT drives. See the owner's guide for your workstation for more information about confidence tests.

## Online Source Code Examples

Source code examples are located online in /usr/people/4Dgifts/examples/ *dmedia*, in directories labeled *audio*, *cd*+*dat*, *dmplay*, *dmrecord*, and *midi*. README files in these directories explain how to use and compile these programs. When a program from one of these directories is included in this guide, it is referred to as the 4Dgifts programname.c program. Because the online source for these programs can get updated more frequently than the printed version of this guide, you should consider the online source code as the most recent version if there is a discrepancy between them.

# Third-party Audio Software and Sound Libraries

This section describes third-party audio software and libraries that are made available to the developer as part of the IRIS digital media libraries. Contact the companies directly for licensing and use rights.

#### **Aware Audio Compression Software and Audio Products**

Aware, Inc. scalable audio compression software is provided with the and can be accessed from Audio File Library routines or Compression Library routines. Two Aware codecs (compressor-decompressors) that provide ISO/MPEG and Aware  $MultiRate^{TM}$  lossless and near-lossless compression are built into the Audio File Library as compression parameters, and additional Aware audio compression software can be accessed through other parameters in the Audio File and Compression Libraries. Aware also offers other licensable audio products and a CD-ROM library; see Appendix B, "Aware Scalable Audio Compression Software," for details.

For more information about Aware products, contact Aware at:

Aware, Inc. One Memorial Drive Cambridge, MA 02142 Phone: (617) 577-1700 Fax: (617) 577-1710

Email: sales@aware.com

#### The Prosonus Sound Library

The Prosonus Sound Library, which is included with the end user media tools, contains more than 10 MB of professional quality music and sound samples. These files are located in <code>/usr/lib/sounds/prosonus</code>, and they represent a small subset of the music, sound effects, and instrument samples created by Prosonus. All included files are sampled at 44.1 kHz and stored in AIFF format. The complete Prosonus Sound Library is separately available on CD-ROM from Prosonus.

For more information about Prosonus products, contact Prosonus at:

Prosonus 11126 Weddington Street North Hollywood, CA 91601 Phone: (800) 999-6191 or (818) 766-5221 Fax: (818) 766-6098

Prosonus files are license-free when used in private presentations. They may be shared via NFS with other Silicon Graphics computers but may not be copied to other systems. If you intend to ship the Prosonus files with a product intended for resale or broadcast, copyrights and royalties may apply. Please contact Prosonus for questions concerning licensing and resale of Prosonus files.

# **Compiling and Linking an Audio Application**

This section lists compiling and linking commands for digital audio and MIDI programs.

To compile an Audio Library program, enter:

cc -g ALsample.c -o ALsample -laudio

To compile an Audio File Library program, enter:

cc -g AFLsample.c -o AFLsample -laudiofile -laudioutil -lm

The Audio File Library also requires linking with *libm.a*, the math library, and with *libaudioutil.so*, the Audio Utility Library.

To compile a CD Audio Library program, enter:

```
cc -g CDsample.c -o CDsample -lcdaudio -lds -libmediad
```

The CD Audio Library also requires linking with *libds.a*, the SCSI device library and *libmediad*, the media library daemon.

To compile a DAT Audio Library program, enter:

```
cc -g DATsample.c -o DATsample -ldataudio
```

Programs making use of more than one of these libraries must link to all of the ones they use (the linking order is often specific):

```
cc -g prog.c -o prog -laudio -laudiofile -lcdaudio -lds -libmediad
```

Depending on the application you are writing, you may also have to link with other libraries such as the GL shared library, the math library, and the C shared library. You can use fast **malloc()** routines by including *malloc.h* and linking with *libmalloc.a*.

The audio and MIDI libraries are compatible with both ANSI C and the standard C. To compile code that is not ANSI-compliant, add **-cckr** to the command line.

# **Programming with the Audio Library**

This chapter begins by presenting basic digital audio concepts. It describes the Audio Library (AL) programming model and how to use the AL for audio I/O, and then it discusses how to apply real-time programming techniques to audio applications.

# Programming with the Audio Library

The Audio Library (AL) provides a uniform application programming interface (API) for audio input to and output from Silicon Graphics workstations that feature high-quality digital audio systems.

The AL comprises routines that provide these basic capabilities:

- creating digital audio input and output connections
- · reading and writing digital audio data
- querying and controlling digital audio data attributes
- querying and controlling the configuration of the audio system
- · handling errors

## In this chapter:

- "Audio Library Basics" on page 68 discusses fundamental audio concepts and explains the features, programming model, error handler, and audio sampling methods of the Audio Library.
- "Initializing an Audio Library Application" on page 74 explains how to create and configure audio ports.
- "Reading and Writing Audio Data" on page 90 explains how to read and write audio samples.
- "Querying and Controlling the Global Audio Device State" on page 98 explains how to query and set global audio parameters.
- "Audio Library Synchronization Facilities" on page 112 explains how to synchronize audio ports with one another and with other media.
- "Real-time Programming Techniques for Audio" on page 119 explains how to use IRIX real-time programming facilities in conjunction with AL routines for providing optimal audio performance.

## **Audio Library Basics**

This section discusses the basic concepts and data structures used by the AL— with particular attention devoted to the programming model, sample data formats, error handling, and programming concepts.

## **Audio Library Features**

Features of the AL include:

- Binary compatibility—AL programs written on one Silicon Graphics workstation equipped with an audio system are executable on other audio-equipped workstations across the product line.
- Shared audio resources—more than one audio application can be active at a time, and multiple programs can have input and output streams open concurrently.
- Real-time performance—a special group of AL functions useful specifically for writing real-time code.

## **Audio Library Programming Model**

The AL programming model has two basic objects:

Audio device

The audio hardware used by the AL, which is shared among audio applications. The audio device 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 samples 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)

## **Digital Audio Data Representation**

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 Sample Rates**

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.

## **Digital Audio Sample Frames**

A sample frame is a set of audio samples that are coincident in time. A sample frame for mono data is a single sample. A sample frame for stereo data consists of a left-right sample pair. A sample frame for 4-channel data has two left-right sample pairs  $(L_1, R_1, L_2, R_2)$ .

Stereo samples are interleaved; left-channel samples alternate with right-channel samples. 4-channel samples are also interleaved, but each frame has two left-right sample pairs.

Figure 6-1 shows the relationship between the number of channels and the frame size of audio sample data.

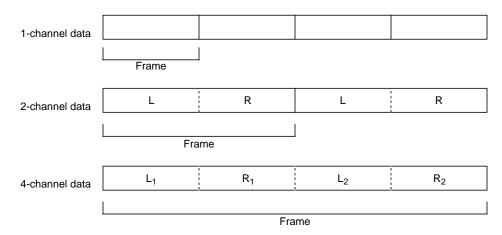


Figure 6-1 Audio Samples and Frames

#### **Digital Audio Sample Formats**

The AL uses 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 AL and 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.

#### **Digital Audio Input and Output Sample Resolutions**

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<sup>2</sup>, 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.

## **Handling Audio Library Errors**

This section describes techniques for error handling in AL applications.

When the AL encounters an error, it:

- 1. Checks to see whether an error handler is set, and if so, calls the specified routine.
- 2. Sets an error code, and returns a failure from the function call.

The default error handler prints a message to *stderr*. Although these error messages may be helpful for debugging during the development phase, you should turn off the default error handler in order to provide more effective error handling by using the IRIX *oserror*(3C) system call to retrieve function return codes.

To turn off the default error handler, call **ALseterrorhandler()**. Its function prototype is:

Substituting zero for *efunc* turns off the error handler.

Most AL routines set error codes when they fail. Throughout this guide, the return values and relevant error codes are listed along with the description of each routine. You can retrieve these error codes by calling *oserror*(3C). Based on these return codes, programs can adapt or recover, and/or alert the user by displaying a dialog box type of notifier or by printing information to the shell window from which the application was launched.

## **Audio Library Application Programming Concepts**

Typically, your AL program must:

- initialize data structures
- set up buffers for passing data between your application and the CPU
- query for available features
- configure and open audio connections
- pass data to and from the ALport and operate on that data
- process errors
- close audio connections
- free system resources

The sections that follow explain these concepts in detail.

## **Initializing an Audio Library Application**

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 AL data structures that provide audio I/O capability.

## About ALports

The AL provides an opaque data structure called an ALport for audio I/O connections. An ALport provides a one-way (input or output) mono, stereo, or 4-channel audio data connection 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.

An ALport consists of a sample queue and static and dynamic state information. For audio input, the hardware places audio samples in an input port's queue at a constant rate, and your application program reads the samples from the queue. Similarly, for audio output, your application writes audio samples to an output port's queue, and the audio hardware removes the samples 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 samples
- stereo data
- a two's complement sample format
- a 16-bit sample width

These settings provide an ALport that is compatible with CD- and DATquality 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()**.
- 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 (page 77).
  - Call **ALsetqueuesize()** to change the sample queue size (page 79).
  - Call **ALsetsampfmt()** to change the sample data format (page 80).
  - Call **ALsetwidth()** to change the sample data width (page 82).
  - Call **ALsetfloatmax()** to set the maximum amplitude of floating point data (not necessary for integer data formats) (page 84).
- 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 6-1 demonstrates how to configure and open an output ALport that accepts floating point mono samples.

**Example 6-1** 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 6-1,

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

### Setting and Getting the Number of Channels for an ALconfig

An ALport can be configured for one, two, or four audio channels. The channel setting remains in effect as long as the port is open.

**Note:** Configuring an ALport to use four channels does not depend on the hardware configuration of the system on which the application is running. See "Querying and Controlling the Global Audio Device State" on page 98 for information on configuring the hardware for 4-channel mode.

To set the number of channels for an ALconfig structure, call **ALsetchannels()**. Its function prototype is:

```
int ALsetchannels ( ALconfig config, long channels )
```

#### where:

config is the ALconfig for which you want to set the channels

channels is the number of channels to configure: 1, 2, or 4

Any ALport that you open with this *config* will have the number of channels that you set in *channels*.

**ALsetchannels()** returns 0 upon successful completion; otherwise, it returns -1 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\_CHANNELS channels is not 1, 2, or 4

To retrieve the channel setting of a given ALconfig structure, call **ALgetchannels()**. Its function prototype is:

```
long ALgetchannels ( ALconfig config )
where:
config is the ALconfig structure being queried
```

**ALgetchannels()** returns the *channel* setting of *config*, upon successful completion; otherwise, it returns –1 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

### Setting and Getting the Sample Queue Size for an ALconfig

Selecting the proper size for the sample queue is very important, because continuous sound output depends on the ability of the application to fill the queue at least as fast as the hardware empties it. For example, if the queue is too small, the application may take too long supply new samples, resulting in audible breaks that sound like pops or clicks. The size of the queue determines the maximum delay that can be tolerated while waiting for the application to get more samples at the given sample rate. To determine how much space to allocate for the sample queue, consider the data type and rate. For example, the default queue size of 100,000 samples provides buffer space for slightly more than one second of 48 kHz stereo audio data, and a little more than three seconds of 32 kHz mono data. To better understand these phenomena, see Figure 6-2 on page 91 for an illustration of a sample queue and read the associated discussion.

**Tip:** The main point to be concerned about is how full to keep the queue, regardless of its size. If the queue is full, more time passes before samples are played. The ideal situation is to keep enough samples in the queue to allow for the longest possible delay that will be experienced in retrieving the next batch of samples. See "Real-time Programming Techniques for Audio" on page 119 for an explanation of how to set the fill threshold for a queue.

The noninclusive values for minimum and maximum allowable queue sizes for ALports on Indigo, Indigo<sup>2</sup>, and Indy workstations are listed in Table 6-1.

 Table 6-1
 Minimum and Maximum Allowable Queue Sizes for ALports

ALport Type	Minimum Size	Maximum Size
Mono	510	131,069
Stereo	1019	262,139
4-channel on Indigo	2038	524, 278
4-channel on Indigo <sup>2</sup> or Indy	1019	262,139

To specify an ALconfig with a sample queue size other than the default for an ALport, call **ALsetqueuesize()**. Its function prototype is:

```
int ALsetqueuesize ( ALconfig config, const long size )
```

### where:

config is the ALconfig structure for which you want to change the

sample queue size

size is the number of sample locations to allocate for the queue

Any Alport that you open with this *config* will have a queue size of *size*.

**ALsetqueuesize()** returns 0 upon successful completion; otherwise, it returns –1 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\_QSIZE *size* is either negative or larger than the

maximum allowable queue size

To retrieve the size of the sample queue in a given ALconfig structure, call **ALgetqueuesize()**. Its function prototype is:

```
long ALgetqueuesize ( ALconfig config )
```

## where:

config is the ALconfig structure being queried

**ALgetqueuesize()** returns the *queuesize* of *config* upon successful completion; otherwise, it returns –1 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

#### Setting and Getting the Sample Data Format for an ALconfig

The AL allows you to choose between three sample formats:

- two's complement (default)
- floating point
- double-precision floating point

To set the sample format type of a given ALconfig structure, call **ALsetsampfmt()**. Its function prototype is:

```
int ALsetsampfmt ( Alconfig config, long sampleformat )
```

where:

config is the ALconfig structure for which you want to change the

sample format

sampleformat must be one of three symbolic constants:

AL\_SAMPFMT\_TWOSCOMP two's complement linear

PCM format, for which the width is specified by

ALsetwidth()

AL\_SAMPFMT\_FLOAT 32-bit IEEE double-

precision floating point scaled linear PCM format

AL\_SAMPFMT\_DOUBLE 64-bit IEEE double-

precision floating point scaled linear PCM format

Any ALport that you open with this *config* will have a sample format of *sampleformat*.

**ALsetsampfmt()** returns 0 upon successful completion; otherwise, it returns –1 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\_SAMPFMT sampleformat is not one of

AL\_SAMPFMT\_TWOSCOMP, AL\_SAMPFMT\_FLOAT, or AL\_SAMPFMT\_DOUBLE

To retrieve the sample format of a given ALconfig structure, call **ALgetsampfmt()**. Its function prototype is:

```
long ALgetsampfmt ( ALconfig config )
```

where:

config is the ALconfig structure being queried

**ALgetsampfmt()** returns the *sampleformat* setting of *config* upon successful completion; otherwise, it returns –1 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

#### Setting and Getting the Integer Sample Width for an ALconfig

The sample width represents the degree of precision to which the full-scale range of an audio signal can be sampled. You can specify the width of two's complement integer sample data, but you can't specify the width of floating point samples. Thus, setting the sample width has no effect when the sample format is AL\_SAMPFMT\_FLOAT or AL\_SAMPFMT\_DOUBLE; however, the width setting does have an effect if the sample format is subsequently changed to AL\_SAMPFMT\_TWOSCOMP.

The sample width also determines which data type the AL uses when reading and writing samples. The sample widths available for two's complement data, and their associated resolutions and data types, are:

8-bit samples representing a total of 2<sup>8</sup> quantized signal values. The AL

treats 8-bit samples as packed, signed characters (chars).

16-bit samples representing a total of 2<sup>16</sup> quantized signal values. The AL

treats 16-bit samples as packed, signed short integers

(shorts)

24-bit samples representing a total of 2<sup>24</sup> quantized signal values. The AL

treats 24-bit samples as right-justified, sign-extended,

signed 32-bit integers (longs).

For all sample widths, sample values map linearly to intermediate signal amplitudes.

To specify the sample width setting of two's complement data for an ALconfig structure, call **ALsetwidth()**. Its function prototype is:

```
int ALsetwidth ( ALconfig config, long samplesize )
```

#### where:

config is the ALconfig structure for which you want to change the

sample width

*samplesize* is a symbolic constant denoting the sample width:

AL\_SAMPLE\_8 1-byte sample width of range –128 to

127

AL\_SAMPLE\_16 2-byte sample width of range -32768 to

32767

AL\_SAMPLE\_24 4-byte sample width of range -8388608

to 8388607

Any ALport that you open with this *config* will have a sample width of *samplesize*.

**ALsetwidth()** returns 0 upon successful completion; otherwise it returns –1 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\_WIDTH samplesize is not one of AL\_SAMPLE\_8,

AL\_SAMPLE\_16, or AL\_SAMPLE\_24

To retrieve the current sample width setting of an ALconfig structure, call **ALgetwidth()**. Its function prototype is:

```
long ALgetwidth ( ALconfig config )
```

where:

config is the ALconfig structure being queried

**ALgetwidth()** returns the *samplesize* of *config* upon successful completion; otherwise, it returns –1 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

## Getting and Setting the Floating Point Data Range

If you configure an ALport to use floating point data (a sample format of either AL\_SAMPFMT\_FLOAT or AL\_SAMPFMT\_DOUBLE), you need to define a maximum value in order to set the upper and lower bounds of the samples that pass through that port. Setting the floating point maximum value specifies a symmetrical range that is centered about zero.

**Tip:** To have more control over the scaling, you can program your application to perform its own floating point-to-integer conversion before passing samples through the ALport.

To set the maximum value of floating point data, call **ALsetfloatmax()**. Its function prototype is:

```
int ALsetfloatmax ( ALconfig config, double maximum_value )
```

where:

config is the ALconfig structure for which you want to set the

floating point maximum value

maximum\_value is an IEEE double-precision floating point value,

which defines the range of floating point data for the

ALreadsamps() or ALwritesamps() functions

Samples read into any ALport that you open with this *config* are scaled to the range [-maximum\_value, maximum\_value]. Samples output from this ALport should be in the range [-maximum\_value, maximum\_value] to avoid limiting. The default maximum value is 1.0.

**Note:** The number of quantization steps that can be represented by floating point samples is a function of the value of  $maximum\_value$ . If  $maximum\_value$  is too small, you will not be able to represent  $2^{16}$  evenly spaced amplitude levels.

ALsetfloatmax() has no function when the sample format is AL\_SAMPFMT\_TWOSCOMP; however, the *maximum\_value* setting takes effect if the sample format is subsequently changed to AL\_SAMPFMT\_FLOAT or AL\_SAMPFMT\_DOUBLE.

**ALsetfloatmax()** returns 0 upon successful completion; otherwise, it returns –1 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\_FLOATMAX maximum\_value is zero

To retrieve the floating point maximum value, call **ALgetfloatmax()**. Its function prototype is:

```
double ALgetfloatmax ( Alconfig config )
```

where:

config is the ALconfig structure being queried

**ALgetfloatmax()** returns the *maximum\_value* of *config* 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\_CONFIG config is either invalid or null

### Retrieving the Setup of an Existing ALport

You can retrieve an ALconfig whose settings match those of an existing ALport. This is an easy way to create an ALconfig to use for changing the dynamic settings of an ALport, as described next in "Modifying the Audio Data Attributes of an Open ALport" on page 85.

To retrieve a new ALconfig structure that is a clone of an existing ALconfig structure already in use by an existing audio port, call **ALgetconfig()**. Its function prototype is:

```
ALconfig ALgetconfig ( ALport port )
```

where:

port is the audio port whose ALconfig structure is being cloned

You should call **ALfreeconfig()** to deallocate the ALconfig when it is no longer needed.

**ALgetconfig()** 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\_PORT port is either invalid or null

AL\_BAD\_OUT\_OF\_MEM insufficient memory available to allocate

ALconfig structure

#### Modifying the Audio Data Attributes of an Open Alport

In general, you don't change the settings for an ALport while it is open, but sometimes you might need to modify the audio data attributes of an ALport while it is open. For example, to create continuous output from multiple sound files that have different sample widths, such as 8-bit and 16-bit data, an application might need to change the sample width of the output port to

match the output data, without closing and reopening the port, in order to prevent interruptions in the output.

To change the data attributes of an ALport instantaneously, use **ALsetsampfmt()**, **ALsetfloatmax()**, and **ALsetwidth()** as needed to define the settings of an ALconfig, which you then pass to the **ALsetconfig()** routine. The only settings that can be modified with this method are the sample format, the sample width, and the maximum floating point value. You can't use this method to change the audio device, the queue size, or the number of channels in an ALport.

**ALsetconfig()** changes an audio port's ALconfig structure to match that of a given ALconfig. Its function prototype is:

```
int ALsetconfig ( ALport port, ALconfig config )
```

#### where:

port is the audio port for which you want to change the ALconfig

settings

config is the ALconfig from which the new settings are copied

**ALsetconfig()** returns 0 upon successful completion; otherwise, it returns –1 and sets an error code that you can retrieve by calling *oserror*(3C). Possible errors include:

AL\_BAD\_PORT port is either invalid or null
AL\_BAD\_CONFIG config is either invalid or null

AL\_BAD\_DEVICE port and config have conflicting device

settings

AL\_BAD\_QSIZE port and config have conflicting values

for queuesize

AL\_BAD\_CHANNELS port and config have conflicting values

for channels setting

### Freeing Resources Associated with an ALconfig

To minimize memory consumption, you should free the memory associated with an ALconfig that is no longer needed. An ALconfig is no longer needed if the application is not going to open any more ports with it.

To deallocate an ALconfig structure, call **ALfreeconfig()**. Its function prototype is:

int ALfreeconfig ( ALconfig config )

where:

config is the ALconfig to deallocate. Freeing an ALconfig structure

does not affect any port(s) that have already been opened

using that ALconfig

**ALfreeconfig()** returns 0 on successful completion; otherwise, it returns –1 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

## **Opening and Closing Audio Ports**

An ALport provides a one-way (input or output) mono, stereo, or 4-channel audio data connection 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.

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()**.

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 samples 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 6-2 opens an input port and an output port and then closes them.

## **Example 6-2** 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 samples to and from ALports.

## **Using Audio Sample Queues**

Audio samples are placed in the sample queue of an ALport to await input or output (see Figure 6-2). The audio system uses one end of the sample queue; the audio application uses the other end.

During audio input (left side of Figure 6-2), the audio hardware continuously writes audio samples 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 samples 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 (right side of Figure 6-2), the application writes audio samples to the tail of the queue. The audio hardware continuously reads samples 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 samples in the queue as fast as the hardware removes them, the queue empties, causing the hardware to send 0-valued samples 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  $(2 \times 48,000 = 96,000)$  samples to the output port at the rate of at least 1 set of samples 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 samples into a queue that already contains 70,000 samples, there isn't enough space in the queue to store all the new samples, and the program will *block* (wait) until enough of the existing samples have been removed to allow for all 40,000 new samples 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.

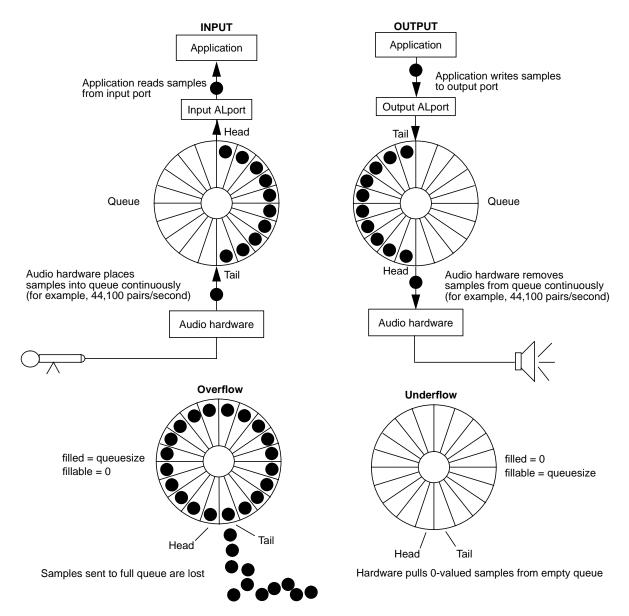


Figure 6-2 shows how input and output ports use audio sample queues.

Figure 6-2 Audio Sample Queues

#### Monitoring the Audio Sample Queue Status to Provide Nonblocking I/O

This section explains how to use the AL routines for monitoring the status of an ALport's sample queue.

The AL maintains the following status information about the queue:

filled the number of queue locations that contain valid data

fillable the number of empty locations in the queue

The sum of the empty locations and the full locations is the total size of the queue:

```
filled + fillable = queuesize
```

Checking the *filled* and *fillable* statuses before reading and writing prevents blocking and helps prevent overflow and underflow errors.

**ALgetfillable()** and **ALgetfilled()** provide instantaneous information on the state of an audio port's queue.

To prevent blocking during output, you can determine how many samples will fit into the queue by calling **ALgetfillable()** before writing any samples, and then write only that many samples to the queue.

To get the number of empty queue locations in a given ALport, call **ALgetfillable()**. Its function prototype is:

```
long ALgetfillable ( ALport port )
```

where:

port is the audio port whose queue is being examined

The value returned indicates how many samples can still be written without blocking.

To prevent blocking during input, you can determine how many samples are in the queue by calling **ALgetfilled()** before reading any samples, then read only that many samples from the queue. You can also periodically check **ALgetfilled()** to find out whether all of your output data has drained before you shut down a port by calling **ALcloseport()**.

To find out how many queue locations in a given audio port currently have valid samples in them at a given instant, call **ALgetfilled()**. Its function prototype is:

```
long ALgetfilled ( ALport port )
where:
port is the audio port whose queue is being examined
```

The value returned indicates how many samples can still be read without blocking if *port* is an input port or how many samples have yet to be played if it is an output port.

### More Methods for Working with Queues

Besides using these routines, you can use **ALgetstatus()** to check for underflow and overflow errors, as described in "Detecting Errors in the Audio Stream" on page 97.

"Real-time Programming Techniques for Audio" on page 119 discusses how to use several other routines that allow an application to view and modify the dynamic state of an audio port. These routines are most useful in developing real-time audio applications.

## **Reading and Writing Samples**

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

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

#### Reading Samples from an Input ALport

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

where:

port is an audio port configured for input

is a pointer to a buffer into which you want to transfer the

samples read from input. *samples* is treated as one of the following types, depending on the configuration of the

ALport:

char \* for integer samples of width AL\_SAMPLE\_8

short \* for integer samples of width AL\_SAMPLE\_16

long \* for integer samples of width AL\_SAMPLE\_24

float \* for floating point samples

double \* for double-precision floating point samples

samplecount is the number of samples to read

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

**Note:** When the application is reading samples 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 samples 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, **ALreadsamps()** will provide 4 samples per frame, but the second pair of samples will be set to 0.

Table 6-2 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 6-2
 Input Conversions for ALreadsamps()

	Hardware Configuration		
Input	Indigo, and Indigo <sup>2</sup> or Indy in Stereo Mode	Indigo <sup>2</sup> 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	$\left(L_1+R_1\right)/2$	$\left(\text{Clip }(L_1+L_2),\text{Clip }(R_1+R_2)\right)/2$	
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 Samples to an Output ALport

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

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

#### where:

port is an audio port configured for input

samples is a pointer to a buffer from which you want to transfer the

samples to the audio port

samplecount is the number of samples you want to read

**Note:** When the application is writing samples 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 samples will be transferred.

Table 6-3 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 6-3** Output Conversions for **ALwritesamps()** 

		Hardware Configuration	
Output	Frame as Written into Port	Indigo, and Indigo <sup>2</sup> or Indy in Stereo Mode	Indigo <sup>2</sup> or Indy in 4-channel Mode
Mono Port	(L <sub>1</sub> )	(L <sub>1</sub> , L <sub>1</sub> )	$(L_1, L_1, 0, 0)$
Stereo Port	$(L_1, R_1)$	$(L_1, R_1)$	$(L_1, R_1, 0, 0)$
4-channel Port	$(L_1, R_1, L_2, R_2)$	$(Clip (L_1 + L_2), Clip (R_1 + R_2))$	$(L_1, R_1, L_2, R_2)$

## **Detecting Errors in the Audio Stream**

Errors in an input or output audio stream may occur if an application is unable to read samples from or write samples to a queue fast enough to satisfy the demands of the real-time hardware.

This section explains how to use two AL routines that let you identify errors and define custom error-reporting functions.

If a program cannot provide samples to an output port fast enough to keep up with the hardware, an audible break in the output may be heard. Likewise, if an application does not read input samples as fast as the hardware puts them in the queue, some samples will be lost.

The audio system detects such discontinuities in audio sample streams, and information concerning these breaks can be gathered by the application. This information can be used to dynamically tune the application execution, to increase the priority of a process, or merely to alert the user to errors.

**ALgetstatus()** provides access to information regarding the most recent error in the audio stream associated with a port. Its function prototype is:

where:

port is the audio port being queried

PVbuffer is an array of longs, the even elements of which should

contain the error parameters you want to read

bufferlength is the number of elements in the PVbuffer array

The odd element directly following each parameter will then be written with the current values associated with each corresponding parameter.

**ALgetstatus()** lets you determine the number of errors associated with the stream, the type of the last error, the length of the last error, and the location of the error relative to the total lifetime of the port.

The location of the error marks the point in the port's lifetime, that is, the time since the port was opened, at which the error was detected. This value

is a 48-bit number representing the number of sample frames sent through the port. The value is generated by concatenating the least significant 24 bits of the values associated with AL\_ERROR\_LOCATION\_LSP and AL\_ERROR\_LOCATION\_MSP.

Table 6-4 lists and describes the error parameters.

**Table 6-4** Error Parameters for **ALgetstatus()** 

Error Parameter	Description
AL_ERROR_LENGTH	Current length in sample frames of the current error. Consecutive values of this variable may differ if the current error has not completed. Only the least significant 24 bits of this variable are valid.
AL_ERROR_LOCATION_LSP	Least significant portion (LSP) of the location of the beginning of the current error. Only the least significant 24 bits of this variable are valid.
AL_ERROR_LOCATION_MSP	Most significant portion of the location of the beginning of the current error (in sample frames). Only the least significant 24 bits of this variable are valid.
AL_ERROR_NUMBER	Number of errors associated with the port since it was opened.
AL_ERROR_TYPE	Type of error that has most recently occurred on the port. Supported types are AL_ERROR_INPUT_OVERFLOW and AL_ERROR_OUTPUT_UNDERFLOW.

## **Querying and Controlling the Global Audio Device State**

This section explains how to use the AL routines for querying and modifying the global audio device state. Your application should query for the availability of special audio features because different workstations have different capabilities, and because programming in this way makes it easy to update your application when new features are added.

Because the audio device is a shared resource, it is especially important to query whether other audio applications are running, so that your application does not inadvertently change a setting upon which another application relies. If no other audio applications are running, your program can use the AL routines described in this section to modify the settings of the state variables, but an application should always verify that it is the only audio application in use before changing any system-wide settings.

There is a core set of parameters that exists on every system and special parameters for capabilities such as 4-channel mode and stereo mic mode that don't exist on all configurations. To query for the availability of a noncore parameter, you have to query for both its existence and whether it supports the settings that you require. It is not necessary to query for the existence of core parameters.

Table 6-5 lists the core set of global parameters, describes their roles, and provides valid values.

 Table 6-5
 Core Global Parameters for AL\_DEFAULT\_DEVICE

Global Parameter	Description and Valid Values
AL_INPUT_SOURCE	Selects the active input source: AL_INPUT_LINE—line-level input jack AL_INPUT_MIC—microphone input jack AL_INPUT_DIGITAL—serial digital input jack
AL_LEFT_INPUT_ATTEN	Controls the left input attenuation level for both the line-in level and the microphone level. Range = $0-255$ , $0 = \text{no}$ attenuation, $255 = \text{maximum}$ attenuation.
AL_RIGHT_INPUT_ATTEN	Controls the right input attenuation level for both the line-in level and the microphone level. Range = $0-255$ , $0$ = no attenuation, $255$ = maximum attenuation.
AL_INPUT_RATE	Indicates the sample rate at the analog (line or microphone) inputs. A positive value indicates a specific sampling rate in Hz. The AL rounds unsupported values to the nearest supported value.
	A negative value indicates a logical value, including AL_RATE_AES_1, meaning to match the analog sampling rate to the rate at which data is arriving at the digital input.
	Note that AL_INPUT_RATE does not apply when the digital input jack is in use. The digital input data stream has its own sample rate, which is determined strictly by the device generating the digital data.

 Table 6-5 (continued)
 Core Global Parameters for AL\_DEFAULT\_DEVICE

Global Parameter	Description and Valid Values
AL_OUTPUT_RATE	Indicates the sample rate at the analog and digital outputs. A positive value indicates a specific sampling rate in Hz. The AL rounds unsupported values to the nearest supported value.
	A negative value indicates a logical value, such as AL_RATE_INPUT_RATE, meaning to match the output sample rate to the rate used by the currently active input, or AL_RATE_AES_1, meaning to match the output sample rate to the rate at which samples are arriving at the digital input.
AL_LEFT_SPEAKER_GAIN	Controls the left speaker and headphone volume levels; does not affect line-level and digital outputs. Range = $0-255$ , $0 = no$ gain, $255 = maximum$ gain. Zero gain does not necessarily mean zero volume.
AL_RIGHT_SPEAKER_GAIN	Controls the right speaker and headphone volume levels; does not affect line-level and digital outputs. Range = $0-255$ , $0$ = no gain, $255$ = maximum gain. Zero gain does not necessarily mean zero volume.
AL_INPUT_COUNT	Read-only value that indicates the number of system-wide open input ALports.
AL_OUTPUT_COUNT	Read-only value that indicates the number of system-wide open output ALports.
AL_UNUSED_COUNT	Read-only value that indicates the number of system-wide unopened ALports.
AL_MONITOR_CTL	Controls monitoring. When monitoring is enabled, audio input is passed through to the output. Input and output sample rates must be precisely matched to prevent distortion. AL_MONITOR_ON enables monitoring; AL_MONITOR_OFF disables monitoring.
AL_SPEAKER_MUTE_CTL	AL_SPEAKER_MUTE_ON mutes speaker and headphones; AL_SPEAKER_MUTE_OFF unmutes speaker and headphones. Any change to AL_LEFT_SPEAKER_GAIN or AL_RIGHT_SPEAKER_GAIN shuts off speaker muting.

Table 6-6 lists and describes special parameters that are available on some systems. You should query for the existence of these parameters *and* whether they support the required values before using them.

 Table 6-6
 Special Global Parameters for System-Dependent Audio Capabilities

Global Parameter	Description and Valid Values
AL_CHANNEL_MODE	Configures the audio hardware. AL_STEREO configures the hardware for stereo audio; AL_4CHANNEL configures the hardware for 4-channel audio on systems that support it.
AL_MIC_MODE	Selects the microphone mode. AL_MONO selects the mono microphone; AL_STEREO selects stereo mic input on systems that support it.
AL_LEFT2_INPUT_ATTEN	Controls the attenuation for the $L_2$ line-level or miclevel input.
AL_RIGHT2_INPUT_ATTEN	Controls the attenuation for the $\ensuremath{R}_2$ line-level or miclevel input.
AL_LEFT_MONITOR_ATTEN	Controls the attenuation for the left half of the monitor signal. Range = $0-255$ , $0 = \text{no}$ attenuation, $255 = \text{maximum}$ attenuation.
AL_RIGHT_MONITOR_ATTEN	Controls the attenuation for the right half of the monitor signal. Range = $0-255$ , $0 = no$ attenuation, $255 = maximum$ attenuation.
AL_DIGITAL_INPUT_RATE	Read-only value; sample rate at which data is arriving at the digital input. The rate is that signified by the nonaudio bits of the incoming digital signal; it is not actually measured. A positive value indicates a specific sampling rate in Hz.
	A negative value indicates a logical value, including AL_RATE_UNDEFINED, meaning that the audio system could not determine the digital input data rate, or the device generating the digital data has marked the data as having an indeterminate rate.
	Note that the digital input data stream contains its own clock signal; thus, its notion of a given rate will differ slightly from an internally generated version of the same rate.

## **Techniques for Working with Global Parameters**

The AL routines for working with parameters are:

ALqueryparams() determines possible hardware parameters

ALgetparams() gets current settings of hardware parameters

ALsetparams() sets hardware parameters

ALgetminmax() gets bounds of hardware parameters

ALgetdefault() gets default values of hardware parameters

ALgetname() returns name for an audio device state variable

All of these routines expect a *device* argument of type long, representing the particular audio device whose state you want to know or change. The only currently supported device is AL\_DEFAULT\_DEVICE.

Several of these routines expect parameter-value buffer (*PVbuffer*) arguments. A *PVbuffer* is simply an array of long integers, where the integers are logically organized as pairs of elements. The first element of each pair is a parameter constant defined in the include file *audio.h*. The second element of each pair stores a value associated with the parameter. The second location can be used to pass a value for a parameter into a routine or to return a value for a given parameter from a routine.

**Tip:** You don't have to pass an array containing all of the possible parameters; create an array that contains only the values of interest.

Some methods for using these routines are:

- If you need a complete list of all available parameters, call
   ALqueryparams(). To be certain that you have a large enough buffer to
   contain the parameter-value pairs, you can pass a zero in place of the
   buffer, then call malloc() to allocate a buffer the size of the returned
   value.
- If you are interested only in certain values, create an array that is twice the size of the number of parameters you are querying, and fill the even locations with the parameters of interest, then:
  - call ALgetparams() to determine the current settings of the state variables.

- fill in the even entries with the values that you want to change, and then call **ALsetparams()** to change the values.
- Some parameters might exist but might not allow the needed settings, so call **ALgetminmax()** to get the parameter bounds and check to be sure that the values you want to use exist.

#### **Getting a List of Available Parameters**

**ALqueryparams()** asks the audio device to supply a list of descriptors and corresponding descriptions for all the currently available global state variables. Its function prototype is:

where:

*device* is the a

is the audio device (AL\_DEFAULT\_DEVICE)

**PVbuffer** 

is an array of longs, into which **ALqueryparams()** writes a descriptor/description pair for each state variable associated with *device*. The even (0, 2, 4, ...) entries receive the descriptors. The odd entries (1, 3, 5, ...) receive one of two description values (negative values indicate read-only parameters):

± AL\_RANGE\_VALUE means that the associated

device state variable can assume a range of values in which the relative magnitude of a value has a meaning; that is, larger values mean an increase in whatever the parameter

controls

± AL\_ENUM\_VALUE means that the associated

device state variable assumes values from an enumerated type—the range is limited, but there is no inherent relationship

between values

bufferlength is the number of elements in the PVbuffer array

**ALqueryparams()** returns a long value representing the buffer size necessary to hold all parameters and their values. If your *PVbuffer* is of smaller dimensions than this value, you have not received a complete set of descriptor/description pairs for *device*. See Table 6-5 for a list of currently supported core global parameters. See Table 6-6 for a list of special global parameters that are not supported on all systems.

**ALsetparams()** lets you modify the values of many of these global parameters, though you should take care in using these functions. See the description of **ALsetparams()** at the end of this section for details.

#### **Getting the Bounds of Global Parameters**

**ALgetminmax()** obtains maximum and minimum values for a given global parameter. Its function prototype is:

where:

device is the audio device (AL\_DEFAULT\_DEVICE)

param is the parameter whose range you want to know

minparam is a pointer to a variable into which the minimum value will

be written

maxparam is a pointer to a variable into which the maximum value will

be written

#### **Getting the Defaults of Global Parameters**

**ALgetdefault()** returns the default value for a given audio hardware device state parameter. Its function prototype is:

```
long ALgetdefault ( long device, long parameter )
```

where:

device is the audio device (AL\_DEFAULT\_DEVICE)

parameter is the parameter whose default value you want to obtain

## **Getting the Names Corresponding to the Global Parameters**

**ALgetname()** returns a pointer to a null-terminated string that can be used to label an audio hardware device state parameter. Treat this string as a read-only string. Its function prototype is:

Table 6-7 lists the global parameter name strings.

 Table 6-7
 Global Parameter Name Strings

Global Parameter	Name String
AL_INPUT_SOURCE	"Line/MIC/AES"
AL_LEFT_INPUT_ATTEN	"Left Input Atten"
AL_RIGHT_INPUT_ATTEN	"Right Input Atten"
AL_INPUT_RATE	"Input Rate"
AL_OUTPUT_RATE	"Output Rate"
AL_LEFT_SPEAKER_GAIN	"Left Output Gain"
AL_RIGHT_SPEAKER_GAIN	"Right Output Gain"
AL_INPUT_COUNT	"Input Count"
AL_OUTPUT_COUNT	"Output Count"
AL_UNUSED_COUNT	"Unused Count"
AL_MONITOR_CTL	"Monitor Control"
AL_LEFT_MONITOR_ATTEN	"Left Monitor Atten"
AL_RIGHT_MONITOR_ATTEN	"Right Monitor Atten"
AL_SPEAKER_MUTE_CTL	"Speaker Mute Control"
AL_MIC_MODE	"Microphone Mode"

 Table 6-7 (continued)
 Global Parameter Name Strings

Global Parameter	Name String
AL_CHANNEL_MODE	"System Channel Mode"
AL_DIGITAL_INPUT_RATE	"Digital Input Rate"

#### **Getting Current Parameter Settings**

**ALgetparams()** gets the current value(s) of the device parameters referenced in the *PVbuffer*. Its function prototype is:

#### where:

device is the audio device (AL\_DEFAULT\_DEVICE)

*PVbuffer* is an array of pairs of longs, the even (0, 2, 4, ...) entries of

which should contain the global parameters whose values

you want to obtain

bufferlength is the number of elements in the PVbuffer array

**ALgetparams()** fills the odd (1, 3, 5, ...) entries in the *PVbuffer* array with the current values associated with each corresponding parameter.

See Table 6-5 for a description of the currently supported core global parameters. See Table 6-6 for a list of special global parameters that are not supported on all systems.

#### Modifying the Values of the Global Parameters

**ALsetparams()** sets the current value(s) of one or more audio hardware device parameters. Its function prototype is:

## where:

device is the audio device (AL\_DEFAULT\_DEVICE)

*PVbuffer* is an array of pairs of longs, the even (0, 2, 4, ...) entries of

which should contain the global parameters whose values you want to change to the corresponding values listed in the

odd (1, 3, 5, ...) entries.

bufferlength is the number of elements in the PVbuffer array

See Table 6-5 for a description of the currently supported core global parameters. See Table 6-6 for a list of special global parameters that are not supported on all systems.

When an application program modifies a global state parameter such as the output sample rate, it may affect other processes on the system that are also engaged in audio processing. For example, if one application is playing a 44.1 kHz recording through an output port, and a second application changes the global output sample rate from 44.1 kHz to 16 kHz, the output of the original application will be distorted.

## Sample Code for Querying Features and Values

This section provides sample code fragments that demonstrate the proper methods to use when querying for certain attributes.

#### **Determining Whether Other Audio Applications Are Running**

To determine whether other audio applications are running, query the system for open input or output ports. To determine the total number of ports available on your system, add the values returned for AL\_INPUT\_COUNT, AL\_OUTPUT\_COUNT, and AL\_UNUSED\_COUNT.

Example 6-3 demonstrates querying for other active audio output.

**Example 6-3** Querying for the Existence of Other Audio Processes

```
^{\star} 'Nonrude' behavior is defined as follows: before modifying global values, first check
 * to see whether any other output ports are currently active; if any other processes have
 * open output ports, don't modify anything.
 * /
rude = 0;
. . .
/*
* Need to determine whether audio is in use. If not, then we
 * can just go ahead and be "rude."
pvbuf[0] = AL_OUTPUT_COUNT;
pvbuf[2] = AL_MONITOR_CTL;
if (ALgetparams(AL_DEFAULT_DEVICE, pvbuf, 4) < 0) {</pre>
    if (oserror() == AL_BAD_DEVICE_ACCESS) {
    fprintf(stderr,"%s: Can't play -- could not access audio hardware.\n");
    return -1;
if ((pvbuf[1] == 0) && (pvbuf[3] == AL_MONITOR_OFF)) {
    rude = 1;
    }
```

#### **Determining the Input and Output Rates**

Querying the system for an input or output rate must be done carefully in order to obtain a valid result. Example 6-4 contains two routines, **get\_input\_rate()** and **get\_output\_rate()**, each of which returns a rate either in Hz or AL\_RATE\_UNDEFINED if the rate cannot be determined. A minimal **main()** program calls the routines. See *ratequery.c* in /usr/people/4Dgifts/examples/dmedia/audio for another example of rate querying.

#### **Example 6-4** Querying for Input and Output Rates

```
#include <audio.h>
. . .
/*
\mbox{\scriptsize \star} These routines expect to be run with the AL error handler shut off.
 * (call ALseterrorhandler(0)).
. . .
int
get_input_rate()
    long buf[6];
    buf[0] = AL_INPUT_RATE;
    buf[2] = AL_INPUT_SOURCE;
    buf[4] = AL_DIGITAL_INPUT_RATE;
    ALgetparams (AL_DEFAULT_DEVICE, buf, 6);
    if (buf[1] == AL_RATE_AES_1 || buf[3] == AL_INPUT_DIGITAL) {
         * We are clocked off of the digital input. Find the
         * real input rate, if we can.
         * /
        if (ALgetdefault(AL_DEFAULT_DEVICE,AL_DIGITAL_INPUT_RATE) >= 0) {
            return buf[5];
        }
    else if (buf[1] > 0) {
         * Input rate is in Hz and we're using an analog input -- return rate.
        return buf[1];
    }
    return AL_RATE_UNDEFINED;
}
int
get_output_rate()
    long buf[4];
    buf[0] = AL_OUTPUT_RATE;
    buf[2] = AL_DIGITAL_INPUT_RATE;
    ALgetparams(AL_DEFAULT_DEVICE, buf, 4);
```

```
if (buf[1] > 0) {
         * Output rate is in Hz -- return it.
        return buf[1];
    else {
         * Output rate is a logical rate -- track down what it means.
        if (buf[1] == AL_RATE_AES_1) {
            /*
             * We are clocked off of the digital input. Find the
             * real input rate, if we can. If we can't, return AL_RATE_UNDEFINED
            if (ALgetdefault(AL_DEFAULT_DEVICE,AL_DIGITAL_INPUT_RATE) >= 0) {
                return buf[3];
        }
        else if (buf[1] == AL_RATE_INPUTRATE) {
            return get_input_rate();
    return AL_RATE_UNDEFINED;
}
main()
    int x;
   ALseterrorhandler(0);
   x = get_output_rate();
    if (x == AL_RATE_UNDEFINED) {
       printf("can't get output rate\n");
    else {
       printf("output rate = %d\n",x);
   x = get_input_rate();
    if (x == AL_RATE_UNDEFINED) {
        printf("can't get input rate\n");
    }
    else {
       printf("input rate = %d\n",x);
}
```

#### **Determining Whether 4-channel Capability Exists**

Although you can open a 4-channel ALport on any system, you cannot change the system's electrical configurations if it does not support 4-channel mode.

To determine whether a system has 4-channel capability, use **ALgetminmax()**, then verify that the maximum value is 4.

Example 6-5 demonstrates how to query for 4-channel hardware capability.

#### **Example 6-5** Querying for 4-channel Capability

```
* Query to see if we are on a machine with 4-channel
* HW capability. If so, switch into 4-channel mode.
* If AL_CHANNEL_MODE both exists (ALgetminmax doesn't
* fail) AND has a maximum of 4, then we're OK.
* If we wanted to be really nice, we could check,
* by querying AL_INPUT_COUNT and AL_OUTPUT_COUNT, to
* see if any other apps were doing audio. If so, we
* might not want to switch to 4-channel mode, lest
* we introduce artifacts into their audio streams.
* /
if (ALgetminmax(AL_DEFAULT_DEVICE, AL_CHANNEL_MODE,
   &min, &max) >= 0 && max == 4) {
   long buf[2];
   buf[0] = AL_CHANNEL_MODE;
   buf[1] = 4;
   ALsetparams(AL_DEFAULT_DEVICE, buf, 2);
}
* Even if we don't have 4-channel HW capability,
* the AL will let us use a 4-channel buffer, so
* we can continue at this point without regard to
* HW type.
* /
```

## **Audio Library Synchronization Facilities**

The AL provides two different facilities for synchronization:

- The AL allows for multiple audio ports (ALports) to be synchronized in a sample accurate manner, by using the absolute sample frame count.
- The AL allows audio data to be related to other media based on common time line, by using the unadjusted system time (UST).

The AL provides a method of determining the absolute sample count of the current sample frame under program control (that is, the sample frame which can be read/written with a call to the Audio Library) and a method of relating UST values to the count of samples which have entered or exited the audio device.

As mentioned in Chapter 2, "Programming with the Digital Media Library," the digital media libraries provide a single time line, UST, through which media may be related. This time value is the number of nanoseconds since the operating system was started. As an absolute time value, UST is not particularly useful. However, it is extremely useful for relating different media types and for evaluating the relative timing of events.

#### Audio Sample Frame Count

Absolute sample frame count is the basis for timing within the AL. Whenever audio is input or output on a device, a count is kept of the sample frames elapsed. This sample frame count is the absolute number of sampling periods elapsed since input or output started. If the audio sample rate is set to 44100 kHz, the sample frame count advances at the nominal rate of 44100 counts per second, regardless of the channel setting for the port (see **ALsetchannels()** for more details on setting the number of channels for a port).

The sample frame count increases regardless of whether an application is reading or writing audio samples using the **ALreadsamps()** or **ALwritesamps()** function calls, respectively. As long as an audio port (ALport) is open, the sample frame count advances.

The AL function **ALgetframenumber()** provides a way for an application to query the absolute sample frame count associated with the current sample frame to be written (in the case of an output port) or read (in the case of an input port).

#### The function prototype for **ALgetframenumber()** is:

where:

port is the audio port of interest

framenum is a pointer to a 64-bit number in which to hold the resultant

frame count value

If **ALgetframenumber()** succeeds, 0 is returned; otherwise a −1 is returned.

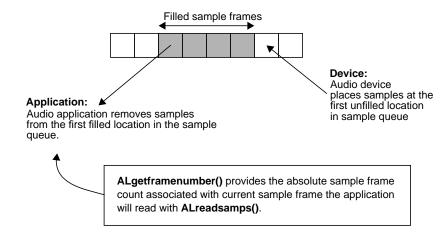
Since the sample frame count is an absolute value of sample frames entering or exiting an audio device, two audio ports (ALports) can be synchronized by reading/writing samples at the identical sample frame count. This "port-to-port" synchronization is guaranteed to be sample accurate.

In general, **ALgetframenumber()** does not return equal values for the sample frame count for different ports. In order to synchronize two audio ports, you will need to make the sample frame count of the two ports match by reading/writing samples from/to one of the sample queues. Example 6-7 demonstrates synchronizing two audio ports.

**Note:** The absolute sample frame count is valid only if the port in question does not overflow (in the case of input) or underflow (in the case of output). When your port underflows or overflows, the value of the sample frame count continuously changes, and you cannot reliably place samples in the queue at a desired location. In order to reestablish a valid value for sample frame count (and hence synchronization) your application must recover from the underflow or overflow (read or write samples as appropriate) and then query for the value of sample frame count again.

Figure 6-3 shows the relationship of the sample frame count returned by **ALgetframenumber()** to sample frames in the queue associated with an input or output audio port (ALport).

#### Input Port Sample Queue:



## **Output Port Sample Queue:**

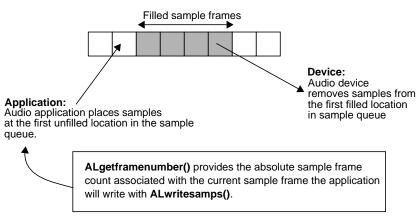


Figure 6-3 Sample Frame Count as Returned by ALgetframenumber()

In Example 6-6, the first two **ALwritesamps()** calls are used to bring the audio ports out of an underflow condition. This ensures that subsequent calls to **ALgetframenumber()** will result in valid sample frame counts.

**Example 6-6** Synchronizing Audio Between Two Output Ports: *align.c* 

```
/* align.c - synchronize audio of two output audio ports */
#include <stdio.h>
#include <dmedia/audio.h>
main(void)
    ALport port_1, port_2;
    short buf_1[10000], buf_2[10000];
    short zilch[10000];
    unsigned long long count_1, count_2, delta_count;
    int i;
    /* get two output ports with default configurations */
    port_1 = ALopenport("port_1", "w", NULL);
    port_2 = ALopenport("port_2", "w", NULL);
    if (port_1 == NULL || port_2 == NULL) {
            printf("oops...no audio ports\n");
            exit(-1);
    }
    /* set up the output sample buffers */
    for (i = 0; i < 10000; i++) {
            buf_1[i] = i;
            buf_2[i] = -i;
            zilch[i] = 0;
    }
    /* bring the output ports out of underflow state */
    ALwritesamps(port_1, zilch, 10000);
    ALwritesamps(port_2, zilch, 5000);
    ALgetframenumber(port_1, &count_1);
    ALgetframenumber(port_2, &count_2);
    /* count_1 should be > count_2 at this point */
    delta_count = count_1 - count_2;
    printf("frame count difference = %lld\n", delta_count);
```

```
/* write delta_count frames of zeroes to port_2 */
ALwritesamps(port_2, zilch, delta_count*2);

ALgetframenumber(port_1, &count_1);
ALgetframenumber(port_2, &count_2);
delta_count = count_1 - count_2;
printf("frame count difference = %lld\n", delta_count);

while (1) {
    ALwritesamps(port_1, buf_1, 10000);
    ALwritesamps(port_2, buf_2, 10000);
    ALgetframenumber(port_1, &count_1);
    ALgetframenumber(port_2, &count_2);
    if (count_1 != count_2) {
        printf("lost synchronization of audio port.\n");
    }
}
ALcloseport(port_1);
ALcloseport(port_2);
```

## **Relating Audio Sample Frame Count to UST**

The IRIS digital media libraries provide a time line called unadjusted system time (UST) for relating media to one another. The UST is a 64-bit count of the number of nanoseconds elapsed since the workstation operating system was started.

The AL provides a way for application programs to relate the number of audio sample frames input to or output from a device to UST values, by providing a pair of values (*UST*, *sample frame count*) simultaneously. The UST value is the time when the samples in the frame entered the audio device (in the case of input) or exited the audio device (in the case of output). That is, the UST is the time at which the samples physically "hit the jacks." The audio system software accounts for any latency in hardware and intermediate buffering.

The AL function **ALgetframetime()** provides both UST and sample frame count for an audio port (ALport) to an application. The function prototype for **ALgetframetime()** is:

where:

port is the audio port of interest

ustime is a pointer to a 64-bit number to hold the value of UST

*fnum* is a pointer to a 64-bit number to hold the value of sample

frame count

If **ALgetframetime()** succeeds, it returns 0 to the application; otherwise, it returns a –1 and sets an error number which can be retrieved with *oserror*(3C).

When an application program calls the **ALgetframetime()** function, the AL provides the most recent pair of (*UST*, *sample frame count*) that it has calculated. In general, the value of sample frame count returned by **ALgetframetime()** is *not* the same as the sample frame count value returned by **ALgetframenumber()**. However, a UST value corresponding to the sample frame count returned by **ALgetframenumber()** can be calculated from (*UST*, *sample frame count*) pairs.

Example 6-7 demonstrates calculating the UST value for the next sample to be read from an input port.

### **Example 6-7** Calculating UST

```
/* getust.c - get ustime for first sample in input port */
#include <stdio.h>
#include <audio.h>
```

```
main(void)
    Alport port;
    long long count_1, count_2, ustime_1, ustime_2;
    double nrate;
    nrate = 1e+9/44100.0; /* nanosecs per sample @ 44.1 kHz*/
    port = ALopenport("my_input", "r", NULL);
    if (port == NULL) exit(-1);
    ALgetframenumber(port, (unsigned long long*)&count_2);
    ALgetframetime(port, (unsigned long long*)&count_1,
        (unsigned long long*)&ustime_1);
    ustime_2 = ustime_1 - (count_1 - count_2)*nrate;
 /* ustime_2 corresponds to the first sample frame in port */
    printf("ust(1) = %lld msc(1) = %lld\n",
           ustime_1, count_1);
    printf("ust(2) = %lld msc(2) = %lld\n",
           ustime_2, count_2);
    ALcloseport(port);
}
```

This example code could have calculated the sample frame rate from multiple (*UST*, *sample frame count*) pairs and used that value instead of calculating it from the nominal audio frame rate.

**Note:** The sample frame value returned by **ALgetframenumber()** is valid only if the port does not overflow/underflow. In the case of underflow or overflow, the (*UST*, *sample frame count*) pair will continue to be valid (though you may wish to request a new, more recent, pair). Note, however, that two back-to-back invocations of **ALgetframetime()** are not guaranteed to result in unique (*UST*, *sample frame count*) pairs.

For a more involved use of UST and sample frame count, see *recordmidi.c++* in /usr/people/4Dgifts/examples/dmedia/midi/syncrecord. This code demonstrates synchronization of audio and MIDI using the UST to relate the two streams of data and is discussed further in "Hands-On MIDI and Audio Synchronization Experience" in Chapter 10.

## **Real-time Programming Techniques for Audio**

The Audio Library provides several routines that modify or return information about the dynamic state of an audio port. These routines, together with the <code>select()</code> or <code>poll()</code> IRIX system calls, make it possible to write applications that can multiplex audio processing tasks with other processing such as user interfaces, audio signal processing, or graphics. Other IRIX system calls, such as <code>prct()</code>, <code>schedct()</code>, and <code>sproc()</code>, also help audio applications to achieve efficient real-time performance. This section discusses these routines and techniques for using them effectively. See the online book, <code>Topics in IRIX Programming</code>, for a description of the IRIX real-time programming facilities.

## Multiplexing Synchronous I/O

The **select()** system call makes it possible for an application to multiplex synchronous I/O tasks. An application passes **select()** three (optionally null) lists of file descriptors, along with an optional timeout parameter. **select()** blocks until one or more of the following conditions occur:

- one or more of the file descriptors in the "read list" are ready for reading
- one or more of the file descriptors in the "write list" are ready for writing
- an exceptional condition is pending for one of the file descriptors in the "exception list"
- a timeout occurs (if specified)

When **select()** returns, it replaces the original file descriptor lists with subsets containing the file descriptors for which requested events have occurred. See the *select*(2) man page for details.

The AL provides a mechanism to control the behavior of **select()** such that you can wake a process before an output queue runs out of samples or before an input sample queue overflows. The functions described in this section control the behavior of **select()**.

#### Getting a File Descriptor for an ALport

**ALgetfd()** returns an IRIX file descriptor for a port that may be used with the **select()** call. Its function prototype is:

```
int ALgetfd ( ALport port )
```

where:

port is the audio port whose file descriptor you want. This

descriptor can then be used to construct the arguments for

a call to **select()** or **poll()** 

When using **select()**, an input port's file descriptor is used in a *read fdset* and an output port's file descriptor is used in a *write fdset*.

When using **poll()**, an input port's file descriptor is used with the POLLIN event flag and an output port's file descriptor is used with the POLLOUT event flag.

These **select()** and **poll()** system calls are used to give up application control of the CPU until the audio port is emptied or filled past a previously set fill point (see the description of **ALsetfillpoint()** below).

#### Setting and Getting the Fill Point for a Queue

**ALsetfillpoint()** allows an application to set a threshold level for an input or output port that controls the behavior of the **select()** function. Its function prototype is:

```
int ALsetfillpoint ( ALport port, long fillpoint )
```

where:

port is the audio port whose fill point you want to set

fillpoint is the fill point value, in number of samples

For an input port, the fill point is the number of locations in the sample queue that must be filled in order to trigger a return from **select()**. For an output port, the fill point is the number of locations that must be free in order to wake up from **select()**.

When used in conjunction with **ALgetfd()** and **select()** or **poll()**, **ALsetfillpoint()** lets you programmatically relinquish control from an audio application to other processes.

**Note:** ALreadsamps() and ALwritesamps() may alter the fill point, so you should (re)set it just before you call **select()** or **poll()**.

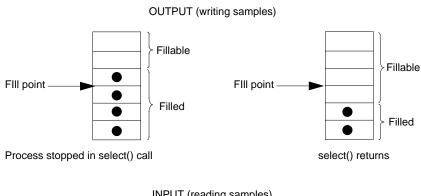
**ALgetfillpoint()** returns the current fill point of a port. Its function prototype is:

long ALgetfillpoint ( ALport port )

where:

port is the audio port being queried

Figure 6-4 shows how the relationship between the number of samples and the fill point affects the behavior of the **select()** call during input and output.



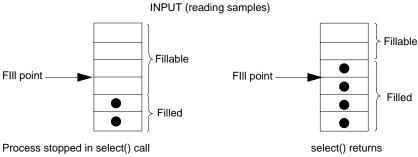


Figure 6-4 Using Fill Points

## **Using Scheduling Control to Give Audio High Priority**

IRIX provides control of process scheduling through the use of the **schedctl()** function. This function allows the program to change its execution priority. To maintain real-time audio processing, an application may need to be placed at a high priority relative to other jobs in the system. See the *schedctl(2)* manual page and for more information on usage. See "Using Shared Arenas and Semaphores" on page 123 for an example program that demonstrates how to use **schedctl()**.

## **Preventing Memory Swapout**

**prctl()** is an IRIX function that gives you control of certain attributes of a process. By using the *PR\_RESIDENT* argument, you can make your audio process immune to kernel memory swapout, thus helping to ensure uninterrupted audio input and output. See the *prctl(2)* man page for more details.

You can also use **mpin()** or **plock()** to lock user pages into memory. See the man pages for those functions for more information.

#### **Creating Multiple Process Threads**

The **sproc()** system call lets you split a process into two *threads*. **sproc()** is an IRIX system call similar to **fork()**, except that it makes use of shared memory. The shared memory features of **sproc()** allow sharing of data, file descriptors, and address space between the two process threads. When using **sproc()** in an application with audio, you can create one thread that services audio and another thread that handles the user interface. Using **sproc()** permits the use of graphical user interfaces without interrupting the audio data stream. See "Using Shared Arenas and Semaphores" on page 123 for an example program that demonstrates how to use **sproc()** in conjunction with an IRIS IM menu (IRIS IM is Silicon Graphics' port of the industry-standard OSF/Motif).

## **Using Shared Arenas and Semaphores**

Another real-time programming technique is to use an IRIX *shared arena*. In essence, a shared arena is a memory-mapped file that you can access just like regular memory.

This section provides some hints for working with shared arenas; more information is available in *Topics in IRIX Programming*.

#### Shared arenas allow:

- shared memory between unrelated processes
- shared synchronization tools: *locks* for controlling access, and semaphores for process communication

Create a shared arena by calling **usinit()**. (The "us" prefix stands for user space.) The first process that calls **usinit()** creates an arena with the given file name; subsequent calls to **usinit()** invoking the same file name attach to the existing arena.

Using shared memory can create data dependency situations such as different process writing to the same memory location at the same time, or one process trying to read from a memory location before another has finished writing to that location. Areas where a potential data dependency exists are called *critical regions*.

Critical regions can be protected with locks, which keep trying until access is gained, or semaphores, which sleep until access is available. Semaphores can be used to allow multiple processes into a critical region at the same time. Processes waiting on a semaphore are queued on a first-come, first-served basis. To acquire (decrement) a semaphore, call **uspsema()**; to release (increment) call **usvsema()**. When uspsema causes the semaphore count to go negative, the process will block until some other process calls **usvsema()**.

The *motifexample.c* program in */usr/people/4Dgifts/examples/dmedia/audio* demonstrates the Audio Library programming concepts presented in this chapter and some Audio File programming concepts that are discussed in Chapter 7, "Programming with the Audio File Library."

Several real-time programming techniques are used in *motifexample.c*:

- The **sproc()** system call creates two separate threads: a user interface thread, and an audio thread. The *PR\_SALL* argument specifies the sharing of all data. Everything that pertains to handling audio is kept in the separate audio process.
- Scheduling control gives the audio process high-priority, nondegrading scheduling.
- Memory swapout is prevented by using mpin() to lock samples in memory.
- A shared memory arena is used to share data.
- Semaphores provide interprocess communication for handling commands from the application.
- Polling is used to monitor two kinds of events: commands from the application and the need for more samples in the queue.

# Chapter 7

# Programming with the Audio File Library

This chapter describes how to use the Audio File Library to store and retrieve audio on disk in AIFF or AIFC files.

# Programming with the Audio File Library

The Audio File (AF) Library, *libaudiofile.so*, provides a uniform programming interface for reading and writing audio files. Currently, the AF Library supports the Apple Computer Inc. Audio Interchange File Format (AIFF) and the Audio Interchange File Format with Compression (AIFF-C).

The AF Library currently supports read-only and write-only file access (but not both simultaneously). Therefore, to alter an existing file, you must create a new file and copy data from the original file. Sample code that demonstrates how to copy the logical components of a file, and other concepts, is available online in /usr/people/4Dgifts/examples/dmedia/soundfile.

The Audio File Library comprises routines that handle four basic tasks:

- · creating and configuring new audio files
- reading and writing track information to and from audio files
- reading and writing instrument configurations to and from audio files
- reading and writing miscellaneous data to and from audio files

#### In this chapter:

- "Audio File Library Basics" on page 128 discusses the basics of programming with the AF Library.
- "Creating and Configuring Audio Files" on page 132 explains how to initialize AF Library data structures.
- "Opening, Closing, and Updating Audio Files" on page 147 explains how to create and use audio files.
- "Reading and Writing Audio Track Information" on page 150 explains how to work with audio file tracks.
- "Audio File Library Programming Tips" on page 171 contains important programming tips for making AF Library programs format independent and multithread/multiprocessor safe.

## **Audio File Library Basics**

This section explains fundamental AF Library concepts.

## **Audio File Library Programming Model**

The AF Library has two basic data structures:

- AFfilesetup, an audio file setup that stores initialization parameters used when creating a new audio file handle
- AFfilehandle, an audio file handle that provides access to the audio file

The basic steps required for setting up an audio file for writing are:

- 1. Initialize an AFfilesetup, by calling **AFnewfilesetup()**.
- 2. Configure the AFfilesetup for your data, as described in "Creating an Audio File Setup" on page 132.
- Open an audio file for reading or writing, as described in "Creating an Audio File Setup" on page 132 by calling either AFopenfile() or AFopenfd(). These routines return an AFfilehandle whose data configuration matches the settings in the AFfilesetup.

#### **Handling Audio File Library Errors**

The AF Library provides an error handling mechanism that directs error messages to *stderr*. You can replace the default AF Library error handler with one of your own.

**AFseterrorhandler()** lets you replace the default error handler function with one of your own. Its function prototype is:

AFerrfunc AFseterrorhandler(AFerrfunc errfunc)

where *errfunc* is a pointer to an alternate error handling routine of type AFerrfunc that is declared as:

void errfunc(long arg1, const char\* arg2)

#### **About Audio Files**

This section explains basic concepts for working with audio files. It describes data structures used by the Audio File Library and in particular, the structure of AIFF-C files and the higher-level abstraction that the AF Library API uses to read and write AIFF-C (and AIFF) files.

The AF Library breaks audio files into the following four functional components:

Audio file format Allows applications to identify audio file formats and

format versions.

Audio tracks Contain audio sample data, parameters that

characterize the data format (such as sample rate, channel configuration, and compression type), and *marker* structures that store sample frame locations in

the track for looping and other purposes.

Instrument Contain instrument parameters for configuring digital samples when playing back audio track data, and loop

markers for repeating tracks or portions of a track.

Miscellaneous Include text strings (author, copyright, name,

data annotation, and so on) and other non-audio information

(such as MIDI data and application-specific data).

The two portions of an audio file you will make most use of are *audio tracks* and *instrument configurations*.

#### **Audio File Formats**

Audio file format is typically indicated by header information preceding the actual data that describes the nature of the data in that file. The file format of an audio file constrains the data format of each of its tracks to one of a set of track formats supported by that file format, but you do not necessarily know which one. You must therefore set and query the track format for each of an audio file's tracks independently of its file format. It is often possible and desirable to write your application so that it queries only the data format(s) of the track(s) (instead of querying the file format) of the audio files it opens.

#### Audio Tracks, Sample Frames, and Track Markers

Audio tracks contain the recorded samples that produce sound when sent to the audio hardware. These samples are stored linearly for mono recordings and as interleaved left-right pairs (left channel in even locations, right channel in odd locations) for stereo recordings. These pairs are called *sample frames* (this term is also used for mono tracks, but a sample frame is the same thing as a sample when mono data is used).

Audio tracks also contain *track markers*, which can be set to point to arbitrary locations in the audio track. These markers, which are identified by a long integer ID number and (optionally) a name string, point to locations between sample frames.

#### **Audio Track Format Parameters**

Data format information, including sample rate, sample format, sample width, and sample compression type is stored as part of the audio track. Several kinds of compression are supported (you can also choose not to use compression). The AF Library automatically compresses samples being written to a file and decompresses samples read from a file. The ability of the AF Library to perform compression/decompression of audio data in real time is dependent on system overhead. To guarantee real-time performance, you should make use of scheduling control as described in "Using Scheduling Control to Give Audio High Priority" in Chapter 6.

#### **Instrument Configurations and Loops**

Instrument configurations contain a set of parameters that define the aspects of a sampler, including detuning, key velocity, and gain. They also contain *loop markers*, which identify the beginning and ending points of loops that allow all or part of the audio track to be repeated. These loop markers point to previously created audio track markers, which in turn refer to locations in the audio track that comprise the beginning and ending of the loop. AIFF and AIFF-C files support two kinds of loops, *sustain* and *release*, each with a beginning and ending marker, which can be used in audio tracks and track markers.

## AIFF-C and the AF Library API

Silicon Graphics has adopted AIFF-C as its standard digital audio file format. AIFF-C is based on Apple Computer's Audio Interchange File Format (AIFF), which conforms to the *EA IFF 85 Standard for Interchange Format Files* developed by Electronic Arts. Unlike the older AIFF standard, AIFF-C files can store compressed sample data as well as two's complement linear PCM sample data.

AIFF-C provides a standard file format for storing sampled sounds on magnetic media. The format can store any number of channels of sampled sound at a variety of sample rates and sample widths. The format is extensible, allowing future support of new compression types and application-specific data, while maintaining backward compatibility.

An AIFF-C file is composed of a series of different kinds of data *chunks*. For the most part, the AF Library API handles low-level chunk manipulation. For complete information on the types of chunks supported by AIFF-C, see the *Audio Interchange File Format with Compression (AIFF-C) Specification*.

Both AIFF and AIFF-C files consist of similar component structures. The chunks in an AIFF-C file are grouped together inside a special *container chunk*. The *EA IFF 85* specification defines several types of container chunks, but the kind used by AIFF-C is of type 'FORM'.

Table 7-1 shows the mapping between the Audio File Library API functional components and the low-level AIFF-C/AIFF data chunks.

**Table 7-1** Mapping of AF Library Components to AIFF-C/AIFF File Chunks

AF Library Functional Component	AIFF-C/AIFF Chunks
File format information	'FVER', 'FORM'
Audio tracks	'SSND', 'COMM', 'MARK', 'AESD', 'COMT' <sup>a</sup>
Instrument configurations	'INST'
Miscellaneous data	'AUTH', 'NAME', '(c) ', 'ANNO',' MIDI ', 'APPL'

a. 'COMT' chunks are not currently supported by the AF Library.

## **Creating and Configuring Audio Files**

This section explains how to initialize an AF Library application, including how to create, configure, and free AF Library data structures for working with audio files.

## **Creating an Audio File Setup**

The AFfilesetup structure stores initialization parameters used when creating a new audio file. When you open an audio file for reading or writing the AF Library returns another structure, an AFfilehandle, which provides access to the audio file and is used as an argument by all AF Library routines.

**AFnewfilesetup()** creates and initializes an AFfilesetup structure that you configure for your data, and then use to open an audio file:

Affilesetup Afnewfilesetup(void)

**AFnewfilesetup()** returns a default AFfilesetup structure.

Table 7-2 lists the AFfilesetup configuration parameters and their defaults.

**Table 7-2** AFfilesetup Parameters and Defaults

Parameter	Default
File format	AF_FILE_AIFFC
Audio track	AF_DEFAULT_TRACK
Audio track sample format, sample width	AF_SAMPFMT_TWOSCOMP, 16-bit
Audio track channels (interleaved)	2 (stereo)
Audio track compression	AF_COMPRESSION_NONE
Audio track markers	Four markers with IDs: 1,2,3,4
Instrument	AF_DEFAULT_INST
Instrument Parameters	(See Table 7-3)
Loops	Two loops with IDs: 1,2; default mode is AF_LOOP_MODE_NOLOOP

Table 7-3 lists the AFfilesetup instrument parameters and their defaults.

 Table 7-3
 Affilesetup Instrument Parameter Constants and Defaults

Instrument Parameter Constant	Default
AF_INST_MIDI_BASENOTE	60
AF_INST_MIDI_HINOTE	127
AF_INST_MIDI_HIVELOCITY	127
AF_INST_MIDI_LONOTE	0
AF_INST_MIDI_LOVELOCITY	1
AF_INST_NUMCENTS_DETUNE	0
AF_INST_NUMDBS_GAIN	0
AF_INST_SUSLOOPID	1 (loop ID for sustain loop)
AF_INST_RELLOOPID	2 (loop ID for release loop)

Your application should free an AFfilesetup that is no longer needed. **AFfreefilesetup()** deallocates an AFfilesetup structure. Its function prototype is:

void Affreefilesetup(Affilesetup setup)

where *setup* is an AFfilesetup previously created by a call to **AFnewfilesetup()**. This does not affect any file previously opened using the same AFfilesetup structure.

Before using the new AFfilesetup to open an audio file, you might need to modify the default AFfilesetup in order to create the configuration you want. The sections that follow explain how to change the default AFfilesetup configuration.

## **Initializing Audio File Format**

You need to set the file format in an AFfilesetup structure before passing the structure to **AFopenfile()**.

**AFinitfilefmt()** configures the file format parameter in an AFfilesetup structure. Its function prototype is:

```
void AFinitfilefmt(AFfilesetup setup, long fmt)
```

where *setup* is the AFfilesetup structure, and *fmt* is an integer constant which specifies an audio format supported by the AF Library. Two valid format types are currently available:

- AF\_FILE\_AIFFC (AIFF-C format)
- AF\_FILE\_AIFF (AIFF format)

A new audio file that is opened by calling **AFopenfile()** with this AFfilesetup as an argument will then be formatted accordingly.

## **Initializing Audio Track Data**

This section explains how to change the default settings for audio track parameters in an AFfilesetup structure before passing the structure to **AFopenfile()**.

**Note:** Each of the functions in this section contains a *trackid* argument, which identifies an audio track in the AFfilesetup structure being initialized. In the current release of the AF Library, the value of *trackid* must always be AF\_DEFAULT\_TRACK.

#### **Initializing Audio Track Sample Rate**

The AF Library requires that you specify the sample rate for a new file before you pass the AFfilesetup structure to **AFopenfile()**.

**AFinitrate()** configures the sample rate in Hz for an audio track in an AFfilesetup structure. Its function prototype is:

```
void AFinitrate(AFfilesetup setup, long trackid, double rate)
```

where *setup* is the AFfilesetup structure, *trackid* is a long integer that identifies an audio track in *setup*, and *rate* is a positive double-precision integer that specifies the sample rate in Hz. For example, to configure *setup* for a CD-quality AIFF-C file, initialize the *rate* for AF\_DEFAULT\_TRACK to 44100.0.

#### Initializing Audio Track Sample Format and Sample Width

**AFinitsampfmt()** initializes the sample format and width parameters for an audio track in an AFfilesetup structure. Its function prototype is:

where *setup* is the AFfilesetup structure, *trackid* is a long integer that identifies an audio track in *setup*, and *fmt* is a long integer constant that denotes a sample format. Currently, only one format is supported: AF\_SAMPFMT\_TWOSCOMP. *width* is a positive long integer value from 1 to 32 that specifies the width (in bits) of the sample data. See "Getting Audio Track Sample Format and Sample Width" on page 152 for more details about sample format and sample width.

**Note:** If the audio track in an AIFF-C file is configured for compression, *fmt* and *width* should match the data format specified by the compression algorithm. See Table 7-4 for a list of compression algorithms.

#### **Initializing Audio Track Channels**

**AFinitchannels()** configures the number of interleaved audio channels for an audio track within an AFfilesetup structure. This information is then used by **AFopenfile()** when it is called with the AFfilesetup structure as an argument. Its function prototype is:

where *setup* is the AFfilesetup structure, *trackid* is a long integer that identifies an audio track in *setup*, and *channels* is a long integer representing the number of interleaved audio channels. Valid values for *channels* are 1 (mono) or 2 (stereo); the default value is 2.

#### **Initializing AES Data**

AES channel status bytes are embedded in AES audio samples to provide additional information about that data, such as whether an emphasis has been added to a sample. For example, on early CD recordings, high frequencies were sometimes emphasized to compensate for the nature of CD players. You might want to reverse compensate for that emphasis if you are

loading AES stream data directly from a CD player through the AES serial input of your workstation for playback on a different source, such as DAT. See the *AES3-1985 (ANSI S4.40-1985)* document for more information about AES channel status bytes.

**AFinitaeschanneldata()** sets a flag, which is off by default, in an AFfilesetup structure to indicate that space should be reserved for the 24 AES channel status bytes that are embedded in all AES data. Its function prototype is:

```
void AFinitaeschanneldata(AFfilesetup setup, long trackid)
```

where *setup* is the AFfilesetup structure, and *trackid* is a long integer that identifies an audio track in *setup*.

**AFsetaeschanneldata()** sets the values of the AES channel status bytes. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *trackid* is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK), and *buf* is a 24-element array that specifies the AES channel status bytes. If no header space has been reserved in the file (by calling **AFinitaeschanneldata()** before creating the file). **AFsetaeschanneldata()** 

**AFinitaeschanneldata()** before creating the file), **AFsetaeschanneldata()** ignores the data and returns without error.

#### **Initializing Audio Track Compression**

**AFinitcompression()** and **AFinitcompressionparams()** let you configure an audio track in an AFfilesetup structure to store compressed audio data. All compression encoding is handled automatically by **AFwriteframes()**; therefore your application program need only work with linear PCM data.

**Note:** AIFF files do not support compression. It is an error to try to open an AIFF file using an AFfilesetup whose compression setting is other than AF\_COMPRESSION\_NONE.

**AFinitcompression()** lets you select from among several built-in default codec (compressor-decompressor) configurations that are preconfigured. If you use **AFinitcompression()** to select one of the default codecs that are built

in to the Audio File Library, you don't have to worry about setting the individual compression parameters, because they are automatically set to the proper values for each default configuration.

**AFinitcompressionparams()** lets you chose the codec configuration and set the associated codec-specific compression parameters yourself, although it does supply the defaults listed in Table 7-4. If you use

**AFinitcompressionparams()**, you have to create and fill in an Audio Utility Library parameter-value list (AUpvlist), as described in "Using the Audio Utility Library to Initialize Parameter Lists" on page 143.

You may also select from additional audio codecs from Aware, Inc. that provide ISO/MPEG I and Aware MultiRate I audio compression, which are built in to the Audio File Library and can be accessed under license from Aware, Inc. by using the parameters in Table 7-4. See Appendix B, "Aware Scalable Audio Compression Software," for more information.

#### The function prototypes are:

long AFinitcompressionparams( AFfilesetup setup, long track,

long compression,

AUpvlist pvlist, long numitems)

where:

*setup* is the AFfilesetup structure that was previously created by

calling **AFnewfilesetup()**.

track is a positive long integer that identifies an audio track in

*setup*. Because AIFF-C files contain only one audio track per file, you should use the constant AF\_DEFAULT\_TRACK to

access the track.

compression is a positive integer symbolic constant that indicates the

type of audio compression being used. See Table 7-4 for a

list of valid compression values.

pvlist is an Audio Utility Library parameter-value list (AUpvlist)

structure, filled with parameters and values related to the compression scheme *compression*. Currently, the only compression schemes that have any parameters are those

supplied by Aware, Inc.

*numitems* is the number of valid entries in the *pvlist*.

Table 7-4 lists the valid *compression* values that you can set for AIFF-C files; *compression* must be AF\_COMPRESSION\_NONE for AIFF files.

 Table 7-4
 Settable Compression Parameter Values and Types

Parameter Value	Compression Type
AF_COMPRESSION_NONE	No compression.
AF_COMPRESSION_G722	64 Kbps ADPCM for 16 kHz 16-bit.
AF_COMPRESSION_G711_ULAW	64 Kbps PCM encoding for 8 kHz 16-bit.
AF_COMPRESSION_G711_ALAW	64 Kbps PCM encoding for 8 kHz 16-bit.
AF_COMPRESSION_AWARE_MPEG <sup>a</sup>	Aware implementation of ISO/MPEG I-audio, Layers I and II. The default setting for this parameter is AF_COMPRESSION_AWARE_DEFAULT_MPEG_II).
AF_COMPRESSION_AWARE_MULTIRATE <sup>a</sup>	Aware MultiRate I lossless or near-lossless algorithm The default setting for this parameter is AF_COMPRESSION_AWARE_DEFAULT_MULTIRATE).
AF_COMPRESSION_AWARE_DEFAULT_MPEG_I <sup>b</sup>	Aware implementation of ISO/MPEG I-audio layer I, joint-stereo, fixed rate at 192 Kbps per channel.
AF_COMPRESSION_AWARE_DEFAULT_MPEG_II <sup>b</sup>	Aware implementation of ISO/MPEG I-audio layer II, joint-stereo, fixed rate at 128 Kbps per channel.
AF_COMPRESSION_AWARE_DEFAULT_MULTIRATE <sup>b</sup>	Aware MultiRate I operating in high-resolution near-lossless (near perfect reconstruction) mode.
AF_COMPRESSION_AWARE_DEFAULT_LOSSLESS <sup>b</sup>	Aware MultiRate I operating in lossless (perfect reconstruction) mode.

a. These values are intended for use with AFinitcompressionparams().

b. These values are intended for use with AFinitcompression().

## **Initializing Audio Track Markers**

Audio track marker structures store sample frame locations in the track for looping and other purposes. Markers are identified by a long integer ID number and (optionally) a name string. Markers point to a location between two samples in the audio track: position 0 is before the first sample, position 1 is between the first and second sample, and so on. You can assign positions to the markers by calling **AFsetmarkpos()**. By default, **AFnewfilesetup()** allocates space for four markers, which is sufficient to store the beginning and end points for both a sustain loop and a release loop.

**AFinitmarkids()** initializes a list of unique marker IDs corresponding to marker structures in a given audio track. Its function prototype is:

where *setup* is the AFfilesetup structure, *trackid* is a long integer that identifies an audio track in *setup*, *markids* is an array of unique positive long integers that will be used as handles for the marker structures in the file opened with *setup*, *nmarks* is a long integer that specifies the number of marker IDs in the *markids* array, that is, the total number of marker structures that will be allocated for the audio track. AIFF-C (and AIFF) files can contain up to 65535 markers in a track.

**AFinitmarkname()** specifies a name string for a marker structure. Marker names default to empty strings. Its function prototype is:

where *setup* is the AFfilesetup structure, *trackid* is a long integer that identifies an audio track in *setup*, *markid* is a positive long integer that identifies a marker structure configured previously by **AFinitmarkids()**, *name* is a string that will be written into the marker structure when an audio file is created by passing *setup* to **AFopenfile()**.

# **Initializing Instrument Data**

This section explains how to initialize the instrument parameters in an AFfilesetup structure before passing the structure to **AFopenfile()**.

**AFinitinstids()** initializes a list of unique instrument IDs that are used to reference the instrument configurations in an AFfilesetup. Its function prototype is:

where *setup* is the AFfilesetup structure, *instids* is an array of positive long integers that are used as handles for the instrument configurations in an audio file, and *ninsts* is the number of entries in *instids*.

**Note:** Currently, the AF Library supports only one instrument configuration per file, which is the maximum allowed by both AIFF and AIFF-C formats; therefore, *ninsts* should be set to either 0 or 1 and *instids* contains at most one element, whose value must be AF\_DEFAULT\_INST. If you set *ninsts* to 0 (meaning that no instrument configuration will be in the audio file you plan to open), **AFinitinstids()** will ignore the *instids* argument, and *instids* can be made a null pointer in this case.

**AFinitloopids()** initializes a list of unique instrument loop IDs that correspond to the loops supplied for a specified instrument in an audio file. Its function prototype is:

where *setup* is the AFfilesetup structure, *instid* is a long integer that identifies an instrument configuration in an audio track. In the current release of the AF Library, the value of *instid* should always be AF\_DEFAULT\_INST. *loopids* is an array of unique, positive long integers that will identify individual loops within an audio file opened using *setup*. *nloops* is a long integer that indicates the number of elements in *loopids*.

The values set in *loopids* can be used by other AF Library functions to set the start point, end point, and play mode for each loop (see "Reading and Writing Instrument Configurations" on page 163).

**Note:** In the current release of the AF Library, both AIFF and AIFF-C files must contain exactly 2 loops: a "sustain" loop and a "release" loop. *nloops* is currently ignored, since its value is always 2.

# **Initializing Miscellaneous Data**

Use these functions to initialize miscellaneous data chunks in an AFfilesetup structure, including file name, author, copyright, and annotation strings, MIDI data, and application-specific data.

**AFinitmiscids()** initializes a list of unique miscellaneous chunk IDs that are then used to reference various file format-dependent data chunks in an audio file. Its function prototype is:

where *setup* is the AFfilesetup structure, *miscids* is an array of unique, positive long integers used to reference the miscellaneous data chunks in an audio file opened using *setup*, *nmisc* is the number of elements in *miscids*, that is, the total number of miscellaneous chunks in the file configuration. The default number of miscellaneous IDs in an AFfilesetup structure is 0.

**AFinitmisctype()** initializes a miscellaneous data chunk with a given ID to one of a variety of supported chunk types in AIFF and AIFF-C files. Its function prototype is:

where *setup* is the AFfilesetup structure, *miscid* is a positive long integer that identifies a miscellaneous chunk in *setup*, and *type* is a long integer constant that defines the chunk type.

Table 7-5 lists the valid parameters for each chunk type.

 Table 7-5
 Miscellaneous Chunk Types and Parameter Values

Parameter Value	Miscellaneous Chunk Type
AF_MISC_AIFF_ANNO	Annotation string
AF_MISC_AIFF_APPL	Application-specific data
AF_MISC_AIFF_AUTH	Author string
AF_MISC_AIFF_COPY	Copyright string
AF_MISC_AIFF_MIDI	MIDI data
AF_MISC_AIFF_NAME	Name string

A single AIFF or AIFF-C file may contain any number of ANNO, APPL, or MIDI chunks, but only one of each of the other (NAME, AUTH, and (c)) miscellaneous chunks.

**AFinitmiscsize()** initializes the amount of space reserved for miscellaneous chunks of data in an AFfilesetup structure. This space is then reserved (written as a zero-filled area) in the header structure of an audio file that is opened using the specified AFfilesetup structure.

Use **AFwritemisc()** to write the data after the file has been opened. The application program is responsible for managing the contents of the header space reserved for each chunk. Its function prototype is:

where *setup* is the AFfilesetup structure, *miscid* is a positive long integer that identifies a miscellaneous chunk in *setup*, and *size* is a non-negative long integer that specifies the number of bytes to reserve for the chunk data identified by *miscid*. It is not necessary to add a trailing "zero pad byte" normally required by chunks in AIFF/AIFF-C files with odd numbers of data bytes (see the description for **AFreadmisc()**); the AF Library handles this transparently.

# Using the Audio Utility Library to Initialize Parameter Lists

The Audio Utility Library, *libaudioutil.so*, provides routines for getting and setting parameters, parameter types, and parameter values contained in lists. Currently, these routines are used only when initializing and querying parameters for the built-in licensable audio compression software from Aware Inc., which is accessible from AF routines. Licenses can be verified by using the **AUchecklicense()** routine.

These routines use the Audio Utility Library parameter-value list (AUpvlist) data structure, which is an array of structures, each of which contains a list of parameters, parameter types, and parameter values.

### Creating and Configuring an Audio Utility Parameter-value List

Use **AUpvnew()** to create an empty AUpvlist with the specified number of blank structures. Its function prototype is:

AUpvlist AUpvnew(int numitems)

where:

numitems

is an integer number of list items to use when creating a new AUpvlist—one list item contains the parameter, parameter

type, and parameter value entries.

**AUpvnew()** returns an empty AUpvlist structure. If an error occurs—either because *numitems* is less than or equal to zero, or because of a memory allocation error—a null pointer, AU\_NULL\_PVLIST, is returned.

### Freeing an Audio Utility Parameter-value List

When an AUpvlist is no longer needed, you should free the memory associated with it by calling **AUpvfree()**. Its function prototype is:

int AUpvfree(AUpvlist pvlist)

where *pvlist* is the structure for which memory should be freed.

## **Getting and Setting Parameter Values**

Use the AUpvlist structure when setting and getting a parameter, its type, and its value. The "set" routines fill in the structure entries for the designated list item with the specified information; the "get" routines return the requested information in pointers corresponding to the item being queried.

Table 7-6 lists and describes the AU Library get and set routines.

**Table 7-6** Audio Utility Library Set and Get Routines

Routine	Description
AUpvgetmaxitems()	Returns the number of list entries allocated for <i>pvlist</i> when it was created by AUpvnew()
AUgetparam()	Gets the parameter of the <i>item</i> th entry in <i>pvlist</i> and returns it in <i>param_ptr</i>
AUpvgetval()	Gets the value of the <i>item</i> th entry in <i>pvlist</i> and returns it in <i>val_ptr</i>
AUpvgetvaltype()	Gets the value type of the <i>item</i> th entry in <i>pvlist</i> and returns it in <i>type_ptr</i>
AUpvsetparam()	Sets the parameter of the <i>item</i> th entry in <i>pvlist</i> to <i>param</i>
AUpvsetval()	Sets the value of the $item$ th entry in $pvlist$ to the value stored in $val\_ptr$
AUpvsetvaltype()	Sets the type of the value of the <i>item</i> th entry in <i>pvlist</i> to <i>type</i>

## The function prototypes of the routines in Table 7-6 are:

```
int AUpvgetmaxitems(AUpvlist pvlist)
int AUpvgetparam(AUpvlist pvlist, int item, int *param_ptr)
int AUpvgetval(AUpvlist pvlist, int item, void *val_ptr)
int AUpvgetvaltype(AUpvlist pvlist, int item, int *type_ptr)
int AUpvsetparam(AUpvlist pvlist, int item, int param)
int AUpvsetval(AUpvlist pvlist, int item, void *val_ptr)
int AUpvsetvaltype(AUpvlist pvlist, int item, int type)
```

pvlist	is an Audio Utility Library parameter-value list data type	
	created by a previous call to <b>AUpvnew()</b> .	

item is an integer zero-based index into an AUpvlist. The index should be a non-negative value that is less than *numitems*-1.

is a symbolic constant describing the type of parameter. Currently supported types are:

• AU\_PVTYPE\_LONG—values are longs

AU\_PVTYPE\_DOUBLE—values are double-precision floating points

param is an integer that will become the parameter or the parameter-value pair.

is a pointer to a void type. Data is read from this pointer, interpreted according to the type associated with this entry, and stored in the AUpvlist.

param\_ptr is a pointer to an integer that is filled with the value of the

parameter portion of a parameter-value pair.

value\_ptr is a pointer to a void type. Data representing the value portion of a parameter-value pair is copied to this address

as interpreted by this entry's type.

#### Verifying a License

Use **AUchecklicense()** to verify whether a license for a particular audio product is available. Its function prototype is:

#### where:

product

is a constant symbol for the product license that is being queried. Currently defined licenses are:

- AU\_LICENSE\_AWARE\_MPEG\_ENCODER
- AU\_LICENSE\_AWARE\_MPEG\_DECODER
- AU\_LICENSE\_AWARE\_MULTIRATE\_ENCODER
- AU\_LICENSE\_AWARE\_MULTIRATE\_DECODER

}

is a pointer to an integer describing a NetLS error, which will be set only if the return value is AU\_LICENSE\_ERR.
 is a pointer to a character pointer, which is changed to point to an informative string only if the return value is AU\_LICENSE\_ERR. The string contains the NetLS error that occurred and contact information on how to obtain

On successful completion, **AUchecklicense()** returns AU\_LICENSE\_OK. If product is unknown, then AU\_BAD\_PRODUCT is returned. If a NetLS error occurs, then AU\_LICENSE\_ERR is returned and \*errorval and \*message are set, describing the error. See Appendix B for more information about NetLS.

Example 7-1 contains a listing of a portion of code from the *aifcinfo.c* demo program that is provided in */usr/people/4Dgifts/examples/dmedia/soundfile*. This portion of code creates an AUpvlist with 3 items, fills those items with the pertinent information, then frees the memory associated with the AUpvlist when it is no longer required.

```
Example 7-1 Creating, Filling, Querying and Freeing an AUpvlist
```

support or a license.

```
AUpvnew(&pvlist, 3);

AUpvsetparam(pvlist, 0, AF_AWARE_PARAM_LAYER);
AUpvsetvaltype(pvlist, 0, AU_PVTYPE_LONG);
AUpvsetparam(pvlist, 1, AF_AWARE_PARAM_BITRATE_POLICY);
AUpvsetvaltype(pvlist, 1, AU_PVTYPE_LONG);
AUpvsetparam(pvlist, 2, AF_AWARE_PARAM_BITRATE_TARGET);
AUpvsetvaltype(pvlist, 2, AU_PVTYPE_LONG);

AFgetcompressionparams(file, AF_DEFAULT_TRACK,
    &track_desc->compressiontype, pvlist, 3);
AUpvgetval(pvlist, 0, &track_desc->aware_desc.layer);
AUpvgetval(pvlist, 1,
    &track_desc->aware_desc.bitratepolicy);
AUpvgetval(pvlist, 2,
    &track_desc->aware_desc.bitratetarget);
AUpvfree(pvlist);
```

# **Opening, Closing, and Updating Audio Files**

Before opening a new audio file using **AFopenfile()**, create and configure an appropriate AFfilesetup structure (as described in "Creating and Configuring Audio Files" on page 132). Audio files can be opened either for reading or writing (but not both simultaneously). In order to change an existing file, you must copy the contents of the file to a new file, writing edits as you go. See the sample source code in <code>/usr/people/4Dgifts/examples/dmedia/soundfile</code> for a demonstration of this process.

# **Opening an Audio File**

**AFopenfile()** allocates and initializes an AFfilehandle structure for a named file. The audio track logical read/write pointer used by **AFreadframes()** and **AFwriteframes()** is initialized to point to the location of the first sample in the audio file. Its function prototype is:

```
Affilehandle AFopenfile(char *name, char *mode, Affilesetup setup)
```

where *name* is a character string that names the file to be opened, and *mode* identifies whether the file is being opened for read or write access. Valid values for *mode* are:

- "r" read-only access
- "w" write-only access

*setup* is an AFfilesetup structure previously created using **AFnewfilesetup()** and configured using various AF Library initialization functions described in previous sections. *setup* is ignored when *mode* is set to "r".

**AFopenfile()** returns an AFfilehandle structure for the named file. If an error occurs, **AFopenfile()** returns the value AF\_NULL\_FILEHANDLE.

# Getting an IRIX File Descriptor for an Audio File

Another way of opening a file is to call the IRIX system function **open()** to open the file, and then get a handle to the file descriptor from the AF Library.

**AFopenfd()** returns an AFfilehandle structure for a file that has already been opened. Its function prototype is:

```
Affilehandle Afopenfd(int fd, char *mode, Affilesetup setup)
```

where *fd* is an IRIX file descriptor previously returned by **open()**, *mode* identifies whether the file is being opened for read or write access (see **AFopenfile()**), and *setup* is an AFfilesetup structure previously created using **AFnewfilesetup()** and configured using various AF Library initialization functions described in previous sections. *setup* is ignored when *mode* is set to "r".

**AFopenfd()** returns an AFfilehandle structure for the named file. If an error occurs, **AFopenfd()** returns the value AF\_NULL\_FILEHANDLE.

**AFgetfd()** returns the IRIX file descriptor associated with the audio file referred to by the given AFfilehandle structure. Its function prototype is:

```
int AFgetfd(AFfilehandle file)
```

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()**.

The file descriptor returned by **AFgetfd()** is intended for use in a **select()** loop. It is not intended to allow reading, writing, and seeking in an audio file without the knowledge of the Audio File Library. Doing so causes unpredictable results unless you save and restore the file position whenever you modify it.

The AF does not reposition the file to the correct place before reading from (using **AFreadframes()**) or writing to (using **AFwriteframes()**) it. If you modify the file position of the file descriptor given by **AFgetfd()**, you should save the file position and restore it to its previous position before reading or writing data to the file. Alternately, you can use one of two different file descriptors opened to the same file. The file must be re-opened in order to get a separate file descriptor (*dup*(2) will not work because it gives you two file descriptors that share the same file offset).

In addition, if you attempt to write to the file, no matter how the AFfilehandle was opened, the results are undefined.

# **Closing and Updating Files**

**AFclosefile()** releases a file's resources back to the system. It also updates the headers of files opened for write access. The AFfilehandle structure deallocated by **AFclosefile()** should not be used by any subsequent AF Library function calls. Its function prototype is:

```
long AFclosefile(AFfilehandle file)
```

where *file* is the AFfilehandle structure to be deallocated. This structure was returned by **AFopenfile()** when the file being closed was created.

**AFclosefile()** returns a negative value if an error occurs while closing a file and updating the header fields. If compression was used to write a file, a negative value indicates that some sample frames were lost due to the filter delay of the compressor. If no error occurs, the return value is 0.

**AFsyncfile()** updates the complete contents of an audio file opened for writing without actually closing the file. This is useful for maintaining consistent header information between writing samples to the file's audio track. Its function prototype is:

```
long AFsyncfile(AFfilehandle file)
```

where *file* is the AFfilehandle structure to be updated. This structure was returned by **AFopenfile()** when the file being closed was created.

**AFsyncfile()** returns a negative value if an error occurs while trying to update *file*. If the update is successful, or if *file* was opened as read-only, **AFsyncfile()** returns 0.

# **Reading and Writing Audio Track Information**

This section describes functions that read and manipulate audio track data and parameters in an audio file. Your application should query for audio file characteristics before opening a file and reading and writing data.

# **Getting Audio File Format**

This section describes functions that query the file format from either a file handle or from an IRIX file descriptor of an opened audio file.

**AFgetfilefmt()** returns an integer value indicating the format of the file and returns a separate version number for AIFF-C files. Its function prototype is:

```
long AFgetfilefmt(AFfilehandle file, long *version)
```

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, and *version* is used to return a file format version number in the form of a non-negative long integer. AIFF files do not use version numbers, so a value of 0 is always returned as the AIFF version number.

**AFgetfilefmt()** returns a non-negative long integer indicating the format of the file. Currently supported values include:

- AF\_FILE\_AIFFC (AIFF-C format)
- AF\_FILE\_AIFF (AIFF format)

but your application should allow for the possibility of other (or unknown) file formats being returned.

**AFidentifyfd()** returns the file format of a given IRIX file descriptor. Its function prototype is:

```
long AFidentifyfd(int fd)
```

where *fd* is an IRIX file descriptor previously returned by **open()**.

**AFidentifyfd()** returns a long integer value representing the audio file format (see **AFgetfilefmt()** for the return values for supported formats). If **AFidentifyfd()** does not recognize the format, AF\_FILE\_UNKNOWN is

returned. If the format is not one supported by the AF Library, AF\_FILE\_UNSUPPORTED is returned.

To determine whether a file is a sound file that can be opened by the AF, check for an unrecognizable format rather than a recognizable format. For example, rather than testing whether the file format is either AF\_FILE\_AIFF or AF\_FILE\_AIFC, use this code:

```
if (filefmt == AF_FILE_UNSUPPORTED ||
    filefmt == AF_FIILE_UNKNOWN)
    {
       printf("file is not supported by the AF library!");
       exit(0);
}
```

Applications that branch depending on the file format should still check for unrecognized formats:

```
switch (AFidentifyfd(fd))
{
case AF_FILE_AIFF: do_aiff_thing(); break;
case AF_FILE_AIFC: do_aiffc_thing(); break;
case AF_FILE_UNKNOWN:
case AF_FILE_UNSUPPORTED:
        printf("this file is not supported by AF library!!");
        exit(0);
default:
        printf("program cannot handle this file format!");
        exit(0);
}
```

**Tip:** Sometimes, instead of checking the file format, you should check the sampling format and other track parameters from the audio file track, as described in "Getting and Setting Audio Track Parameters" on page 152. For example, a program that simply reads 16-bit AF\_SAMPFMT\_TWOSCOMP audio data out of an AIFF file should be able to correctly read that type of data out of a file whose file format is not AIFF, as long as it does not also intend to read AIFF-specific chunks from the data (for example, certain MISC and INST chunks). Such a program has no need to call **AFidentifyfd()** or **AFgetfilefmt()** to get the file format.

# **Getting and Setting Audio Track Parameters**

Most audio track parameters (except markers) must be initialized before a new audio file is opened and cannot be modified after that point, but you should query an audio file for its track parameters.

#### **Getting Audio Track Sample Rate**

**AFgetrate()** returns the sample rate of an audio track in an opened audio file. Its function prototype is:

```
double AFgetrate(AFfilehandle file, long trackid)
```

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, and *trackid* is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK).

**AFgetrate()** returns a double-precision floating point value that describes in Hz the audio sampling rate of the audio track.

#### **Getting Audio Track Sample Format and Sample Width**

**AFgetsampfmt()** retrieves the sample format and sample width for an audio track in an opened audio file. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *trackid* is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK), *sampfmt* is a pointer to a long integer denoting the format of the sample data (for AIFF and AIFF-C files, this value is always AF\_SAMPFMT\_TWOSCOMP), and *width* is a pointer to a long integer that denotes the sample width in bits (for AIFF and AIFF-C files, this value is between 1 and 32).

**Tip:** Do not assume that AF\_SAMPFMT\_TWOSCOMP is the only value that can be returned by **AFgetsampfmt()**. Write your application so that it rejects files with sample formats it does not support.

Sample width may or may not have meaning, depending on the value of *sampfmt*. For AF\_SAMPFMT\_TWOSCOMP data, you can use the sample width value to determine the data type used to pass samples to **AFwriteframes()** and from **AFreadframes()**: 1–8 bit samples are packed into chars, 9–16 bit samples are packed into shorts, and 17–32 bit samples are packed into longs. Data formats whose sample width is not a multiple of eight are augmented by zero-bit-padding on the right (see Figure 7-1).

There is a special case for reading 24-bit integer data. The AF automatically converts 3-byte data into 4-byte quantities in a manner that is compatible with the Audio Library (AL) by sign-extending the left-most bits of 17 to 24-bit data.

Figure 7-1 shows the data packing for two complement integer data (AF\_SAMPFMT\_TWOSCOMP).

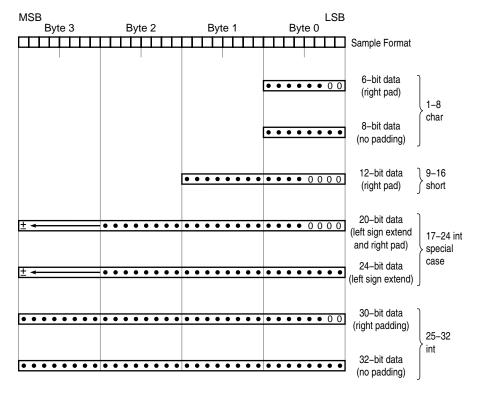


Figure 7-1 Audio Data Packing Formats

**Tip:** Don't assume that the maximum size of integers in files opened by the AF Library is 32 bits or that the number of bits will be a multiple of 8. Even for AIFF files, the sample width is not necessarily a multiple of 8. Generally, this can be ignored, because audio samples that do not take up an integral number of bytes are left-justified inside the next larger integral number of bytes (with the remaining bits set to 0). But you should write your application so that it does not assume the sample width is a multiple of 8, as demonstrated in Example 7-2.

Example 7-2 checks for the audio track sample format, and then classifies integer data according to its sample width.

**Example 7-2** Checking Audio Track Sample Format and Sample Width

```
#include <dmedia/audiofile.h>
Affilehandle h = Afopenfile(....);
if (!h) return;
AFgetsampfmt(h, AF_DEFAULT_TRACK, &sampfmt, &sampwidth);
if (sampfmt != AF_SAMPFMT_TWOSCOMP)
   printf("This program can't read audio files of this "
           sample format");
           exit(0);
/* round sampwidth up to nearest number of bytes */
int nbytes = ((sampwidth-1) / 8) + 1;
switch (nbytes)
   {
   case 1: do_8_thing(); break;
   case 2: do_16_thing(); break;
   case 3: do_24_thing(); break;
   case 4: do_32_thing(); break;
   default:
       printf("This program can't read audio files of this "
               sample width d\n", sampwidth);
               exit(0);
               }
```

#### **Getting Audio Track Channels**

The number of channels in an audio track is initially set by **AFinitchannels()** before the file is created.

**AFgetchannels()** returns the number of interleaved audio channels in the audio track of an opened audio file. Its function prototype is:

```
long AFgetchannels(AFfilehandle file, long trackid)
```

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, and *trackid* is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK). **AFgetchannels()** returns 1 if *trackid* is monaural, 2 if it is stereo, or any other positive integer (even for AIFF/AIFF-C files).

**Tip:** Your application should be able to handle audio files containing an arbitrary number of channels. For example, the application could reject a file that has more than the supported number of channels, or it could combine channels selectively or use certain channels while ignoring others.

## **Getting AES Data**

**AFgetaeschanneldata()** retrieves AES channel status information from an opened audio file. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *trackid* is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK), and *buf* is a 24-element array that receives the AES channel status bytes.

**AFgetaeschanneldata()** returns a 1 if there is AES channel data, or a 0 if there is no data.

**Tip:** There is no guarantee whether a given file format will contain AES data, so your application should call **AFgetaeschanneldata()** to determine whether AES channel bytes are encoded in an audio file.

#### **Getting Audio Track Compression**

This section describes routines that let you get compression information for an audio track from an AFfilehandle structure.

When reading or writing a file (even an AIFF-C file) containing compressed data, first call **AFgetsampfmt()** to get the native sample format of the codec, and check that it is able to be read/written using that format. The native sample format of a codec is the sample format of the data it produces on decompression or expects on compression.

**Tip:** Your application should reject compressed files with native sample formats it does not support. Check for an unrecognized format rather than a defined format. The currently defined codecs all convert the compressed data to and from 16-bit AF\_SAMPFMT\_TWOSCOMP data, but you should not assume that a certain format is guaranteed for future codecs. For example, if you know that the file is AF\_COMPRESSION\_G711\_ULAW, then the native format for that codec is 16-bit AF\_SAMPFMT\_TWOSCOMP. However, you should call **AFgetsampfmt()** in any case, to allow for the possibility of future codecs whose native sample format is something other than 16-bit signed integer or which have more than one native sample format (some may be configurable or may vary depending on what kind of data was originally compressed).

**AFgetcompression()** and **AFgetcompressionparams()** return the compression type used in the audio track of an opened audio file. In addition, **AFgetcompressionparams()** scans a requested number of items and returns codec-specific parameters for the audio track. **AFgetcompression()** returns a long integer representing the compression algorithm used for the audio track's data; **AFgetcompressionparams()** returns this value in the *compression* pointer.

### The function prototypes are:

#### where:

file

1S t.

is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**.

trackid	is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF_DEFAULT_TRACK).
compression	is a pointer to a positive long integer that will be filled in with the symbolic constant that indicates the type of audio compression being used for the specified audio track. See Table 7-7 for a list of possible return values.
pvlist	is an AUpvlist structure, to be filled with parameters and values related to the compression scheme <i>compression</i> . Currently, the only compression schemes that have any parameters are those supplied by Aware, Inc.
numitems	is the number of valid entries in the pvlist.

Table 7-7 lists the valid return values for AIFF-C files. AIFF files always return AF\_COMPRESSION\_NONE.

 Table 7-7
 Valid Return Values for Compression Algorithms and Parameters

Parameter Value	Compression Type
AF_COMPRESSION_UNKNOWN	Unrecognized compression scheme
AF_COMPRESSION_NONE	No compression
AF_COMPRESSION_G722	64 Kbps ADPCM for 16 kHz 16-bit
AF_COMPRESSION_G711_ULAW	64 Kbps encoding for 8 kHz 16-bit
AF_COMPRESSION_G711_ALAW	64 Kbps encoding for 8 kHz 16-bit
AF_COMPRESSION_AWARE_MPEG	Aware implementation of ISO/ MPEG I-audio Layers I and II
AF_COMPRESSION_AWARE_MULTIRATE	Aware MultiRate I lossless or near- lossless algorithm
AF_COMPRESSION_APPLE_ACE3	Not currently supported
AF_COMPRESSION_APPLE_ACE8	Not currently supported
AF_COMPRESSION_APPLE_MAC3	Not currently supported
AF_COMPRESSION_APPLE_MAC6	Not currently supported

The Audio File Library provides built-in codec support for five compression algorithms: CCITT G.722, CCITT G.711  $\mu$ -law and A-law, and the Aware, Inc. ISO/MPEG I-audio and MultiRate I algorithms. To get more specific information about the Aware algorithms, such as MPEG I layers, see Appendix B, "Aware Scalable Audio Compression Software."

**Note:** The four Apple compression algorithms listed in Table 7-7 are proprietary to Apple Computer Inc., and are not currently supported by the Audio File Library.

**AFgetcompressionname()** returns a null-terminated string containing the name of the compression algorithm used for an audio track in an opened audio file. Its function prototype is:

```
char *AFgetcompressionname(AFfilehandle file, long trackid)
```

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, and *trackid* is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK).

If compression is not used, as is the case with AIFF files, **AFgetcompressionname()** returns a null string.

### **Getting Audio Track Sample Frame Count**

**AFgetframecnt()** returns the total number of sample frames in the audio track of an opened audio file. Its function prototype is:

```
long AFgetframecnt(AFfilehandle file, long trackid)
```

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**. *trackid* is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK).

**AFgetframecnt()** returns a long integer value that is the current total of sample frames in the track.

### **Getting and Setting Audio Track Markers**

This section describes functions that get information about the markers in a given audio track and explains how to set the position of those markers. Markers point to positions between adjacent sample frames. For a track

containing n sample frames, position 0 is before the first sample frame, and position n is after the last sample frame in the track.

**AFgetmarkids()** retrieves an array of marker IDs from a given audio track in an opened audio file. It returns the number of marker structures in the specified audio track. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *trackid* is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK), and *markids* is an array of long integers that receives the marker IDs for the marker structures in the audio track.

**AFgetmarkids()** returns a non-negative integer value specifying the number of marker structures in the given audio track.

**Tip:** Check for unrecognized mark return values rather than recognized values. Write your application so that it expects any number of marks and any type of mark (not just the currently defined types) and rejects files containing marks it does not support.

Typically, you call **AFgetmarkids()** twice. The first time, you pass *markids* a null pointer and check the return value of the function. This value tells you how many locations to allocate in the *markids* array, which you pass back to **AFgetmarkids()** to obtain the list of marker IDs.

**AFgetmarkname()** returns the name string of a given marker within the audio track of an opened audio file. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *trackid* is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK), and *markid* is the ID of the marker whose name you want to retrieve.

**AFgetmarkname()** returns a null-terminated character string that is the name associated with the given *markid*.

**AFgetmarkpos()** returns the frame location of a given marker in the audio track of an opened audio file. Its function prototype is:

long AFgetmarkpos(AFfilehandle file, long trackid, long
markid)

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *trackid* is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK), and *markid* is the ID of the marker whose position you want to discover.

**AFgetmarkpos()** returns a non-negative long integer value indicating the position of the marker in the track.

**AFsetmarkpos()** sets the frame location of a given marker in the audio track of an audio file opened for write access. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *trackid* is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK), *markid* is the ID of the marker whose position you want to move, and *markpos* is a non-negative long integer that describes the position to which you want to move the marker in the track.

# Seeking, Reading, and Writing Audio Track Frames

This section describes functions that position the read pointer in a file's audio track and functions that read and write frames. You can read and seek only from a file opened for reading. Similarly, you can write frames only to a file opened for writing.

### Seeking to a Position in an Audio File Track

When a file is opened for read access by **AFopenfile()** or **AFopenfd()**, the logical track pointer for the audio track is initialized to point to the first frame in the track. This location can be changed by calling **AFseekframe()**. Before returning, **AFreadframes()** moves the logical track pointer so that it points to the frame following the one last copied into *frames*.

**Caution:** The logical track pointer is not the same thing as the IRIX file pointer which you position by calling the IRIX *lseek*(2) command.

**AFseekframe()** moves the logical track pointer in the audio track of an audio file opened for read-only access to a specified frame. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *trackid* is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK), and *offset* is the number of frames from the beginning of the track that the pointer will be moved to. This value is between 0 and the total number of frames in the track, minus 1. The total number of frames in the track can be determined by calling **AFgetframecnt()**.

When **AFseekframe()** succeeds, it returns the actual offset value; otherwise, it returns a negative value.

## Reading Audio Frames from an Audio Track

**AFreadframes()** copies sample frames from an audio file opened for reading to a buffer. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *trackid* is the ID for the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK), *frames* is a pointer to a buffer into which you want to transfer copies of sample frames from *file*, and *count* is the number of sample frames you want to read from *file*.

**AFreadframes()** returns a long value indicating the number of frames successfully read from the audio track.

The data copied into *frames* must be interpreted according to the sample format and sample width parameter returned by **AFgetsampfmt()** and channel count returned by **AFgetchannels()**, as described in "Getting Audio

Track Sample Format and Sample Width" on page 152. For AF\_SAMPFMT\_TWOSCOMP, **AFreadframes()** copies the frames to the buffer using the smallest data type (char, short, or long) that capable of holding the data. **AFreadframes()** automatically decompresses data encoded using any of the supported compression algorithms. (For Aware compression, an Aware license must be installed.)

**Tip:** Query for the sample format, sample width, and channels. Don't assume that a particular file format determines the sample format, sample width, or number of channels. Provide a mechanism for detecting and handling unsupported file configurations.

### Writing Audio Frames to an Audio Track

When a file is opened for write access by **AFopenfile()** or **AFopenfd()**, the logical track pointer for the file's audio track is initialized to point to the first frame in the track. Before returning, **AFwriteframes()** moves the logical track pointer so that it points to the frame following the one last copied into *samples*.

**Caution:** The logical track pointer is not the same thing as the IRIX file pointer which you position by calling the IRIX *lseek*(2) command.

**AFwriteframes()** copies frames from a buffer to an audio file opened for writing. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *track* is a long integer which identifies the audio track (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_TRACK), *samples* is a pointer to a buffer containing sample frames that you want to write to *file*, and *count* is the number of sample frames you want to write to *file*.

For AF\_SAMPFMT\_TWOSCOMP data, **AFwriteframes()** expects the frames to be buffered using the smallest data type (char, short, or long) capable of holding the data. **AFwriteframes()** automatically compresses data encoded using any of the supported compression algorithms.

**AFwriteframes()** returns a long value indicating the number of frames successfully written to the audio track. The return value is normally greater than or equal to 0; however, when a codec is being used and buffered data cannot be written to disk, that data is lost. In such a case, **AFwriteframes()** returns a negative value, indicating the number of sample frames lost.

# **Reading and Writing Instrument Configurations**

Use the functions in this section to retrieve and manipulate instrument configuration data and parameters.

#### **Getting and Setting Instrument Parameters**

Use the functions described in this section to retrieve and set the instrument configuration parameters of an audio file. The parameters can be read from any opened audio file and written to any audio file opened as write-only.

**AFgetinstids()** retrieves an array of instrument IDs corresponding to the instrument chunks in a given audio file. It returns the number of instrument chunks in the file. Its function prototype is:

```
long AFgetinstids(AFfilehandle file, long instids[])
```

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, and *instids* is an array of long integer instrument IDs that reference instrument chunks within the file.

Typically, you call **AFgetinstids()** twice. The first time, you pass *instids* a null pointer and check the return value of the function. This value tells you how many locations to allocate in the *instids* array, which you pass back to **AFgetinstids()** to obtain the list of instrument IDs.

**Note:** The AF Library currently supports only AIFF and AIFF-C file types, so the number of instrument chunks is always either 0 or 1. If the file does contain an instrument chunk, its ID will always be AF\_DEFAULT\_INST for AIFF and AIFF-C files. But other instrument configurations could be returned in future releases of the AF Library.

**Tip:** Write your application so that it checks for and rejects instrument configurations that you don't want to support.

**AFgetinstparamlong()** retrieves a long instrument configuration parameter value from an instrument configuration in an open audio file. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**. *instid* is the instrument ID for the instrument configuration chunk (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_INST). *param* is a symbolic constant that identifies an instrument parameter. See Table 7-3 and Table 7-8 for a list of valid parameter constants and values associated with them.

**AFgetinstparamlong()** returns the long integer value associated with the parameter specified in *param*. If *instid* or *param* is not valid, the value returned is 0.

Table 7-8 lists the instrument parameter constants and their valid values.

**Table 7-8** Instrument Parameter Constants and Valid Values

Instrument Parameter Constant	Valid Values
AF_INST_MIDI_BASENOTE	0–127
AF_INST_NUMCENTS_DETUNE	-50 to 50
AF_INST_MIDI LONOTE	0–127
AF_INST_MIDI_HINOTE	0–127
AF_INST_MIDI_LOVELOCITY	1–127
AF_INST_MIDI_HIVELOCITY	1–127
AF_INST_NUMDBS_GAIN	-32768 to 32767
AF_INST_SUSLOOPID	Any positive long integer value
AF_INST_RELLOOPID	Any positive long integer value

**Tip:** Check for unrecognized instrument configuration and parameters rather than recognized types. Write your application so that it expects any type of instrument configuration (not just the currently defined types) and rejects files containing instruments it does not recognize.

**AFsetinstparamlong()** writes a long instrument configuration parameter value to a given instrument configuration chunk in an audio file that has been opened for writing. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *instid* is the instrument ID for the instrument configuration chunk (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_INST), *param* is a symbolic constant that identifies an instrument parameter, and *value* is the long integer value you want to assign to parameter named by *param*. See Table 7-3 and Table 7-8 for a list of valid parameter constants and values associated with them.

### **Getting and Setting Loop Information**

This section describes functions that retrieve and set the positions of instrument loops within an opened audio file. The loop information may be read from any opened audio file and written to any audio file opened as write-only. To get and set instrument loop IDs, use **AFgetinstparamlong()** and **AFsetinstparamlong()**, as described in "Reading and Writing Instrument Configurations" on page 163.

**AFgetloopmode()** returns the loop mode of a given loop in the instrument configuration of an opened audio file. Its function prototype is:

```
long AFgetloopmode(AFfilehandle file, long instid, long
loopid)
```

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *instid* is the instrument ID for the instrument configuration chunk (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_INST), and *loopid* is the ID number associated with the loop whose mode you wish to read.

**AFgetloopmode()** returns a long integer value representing the loop mode. Current valid values for loop mode are:

- AF\_LOOP\_MODE\_NOLOOP (no loop)
- AF\_LOOP\_MODE\_FORW (forward loop)
- AF\_LOOP\_MODE\_FORWBAKW (alternating forward/backward)

**AFsetloopmode()** sets the loop mode of a given loop in the instrument configuration of an audio file opened as write-only. Its function prototype is:

where file is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *instid* is the instrument ID for the instrument configuration chunk (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_INST), loopid is the ID number associated with the loop whose mode you wish to write, and *mode* is the long integer value you wish to set for the loop mode. See **AFgetloopmode()** for the list of valid *mode* values.

**AFgetloopstart()** returns an audio track marker ID associated with the starting point of a given instrument loop. Its function prototype is:

```
long Afgetloopstart(Affilehandle file, long instid, long
loopid)
```

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *instid* is the instrument ID for the instrument configuration chunk (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_INST), and *loopid* is the ID number associated with the loop whose starting point you wish to read.

**AFgetloopstart()** returns a long integer value, which is a marker ID in the audio track. See "Getting and Setting Audio Track Markers" on page 158 in "Reading and Writing Audio Track Information" on page 150 for information on how to manipulate the position of the markers referred to by the marker IDs.

**AFsetloopstart()** causes an audio track marker ID to be associated with the starting point of a given instrument loop. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *instid* is the instrument ID for the instrument configuration chunk (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_INST), *loopid* is the ID number associated with the loop whose starting point you wish to write, and *markid* is the audio track marker that you wish to assign as the starting point of the given loop.

**AFgetloopend()** returns an audio track marker ID associated with the ending point of a given instrument loop. Its function prototype is:

```
long Afgetloopend(Affilehandle file, long instid, long
loopid)
```

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *instid* is the instrument ID for the instrument configuration chunk (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_INST), and *loopid* is the ID number associated with the loop whose ending point you wish to read.

**AFgetloopend()** returns a long integer value which is a marker ID in the audio track. See "Getting and Setting Audio Track Markers" on page 158 in "Reading and Writing Audio Track Information" on page 150 for information on how to manipulate the position of the markers referred to by the marker IDs.

**AFsetloopend()** causes an audio track marker ID to be associated with the ending point of a given instrument loop. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *instid* is the instrument ID for the instrument configuration chunk (for AIFF and AIFF-C files, this value should always be AF\_DEFAULT\_INST), *loopid* is the ID number associated with the loop whose ending point you wish to write, and *markid* is the audio track marker that you wish to assign as the ending point of the given loop.

**Tip:** Loop queries can return any configuration of loops within an instrument, not just the fixed value of 2 in AIFF/AIFF-C files. Have your application check for and reject loop configurations it does not support.

# Handling Miscellaneous Data Chunks

The following sections describe how to read to, write from, and get information about the miscellaneous data chunks in an audio file.

### **Getting Miscellaneous Data Parameters**

This section describes functions that get information about the number, size and type of miscellaneous data chunks in an opened audio file.

**AFgetmiscids()** returns the number of miscellaneous data chunks in a file and an array containing the IDs of each miscellaneous chunk. Its function prototype is:

```
long AFgetmiscids(AFfilehandle file, long miscids[])
```

*file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**. *miscids* is an array of positive long integers that contains the IDs for the miscellaneous data chunks in *file*.

**AFgetmiscids()** returns a long integer value equal to the number of miscellaneous data chunks in *file*.

To fill the *miscids* array with the corresponding IDs, you first call **AFgetmiscids()** with a null *miscids* pointer, and then allocate a *miscids* buffer according to the return value. You can then call **AFgetmiscids()** again, passing the properly dimensioned *miscids* buffer to obtain the list of IDs.

**AFgetmisctype()** returns the type of a given miscellaneous chunk. Its function prototype is:

```
long AFgetmisctype(AFfilehandle file, long chunkid)
```

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, and *chunkid* is a positive long integer miscellaneous chunk ID from the *miscids* array returned by **AFgetmiscids()**.

**AFgetmisctype()** returns a long integer constant that describes the chunk type. See Table 7-5 for the list of valid chunk types and constants. If the chunk is not of any of the types listed in Table 7-5, **AFgetmisctype()** will return the value AF\_MISC\_AIFF\_UNRECOGNIZED.

**Tip:** The set of chunk types may expand at any time. Check for unrecognized chunk types rather than recognized chunk types. Write your application so that it expects any type of MISC chunk (not just the currently defined types) and rejects miscellaneous chunks it does not recognize.

**AFgetmiscsize()** returns the size of a given miscellaneous data chunk in bytes. Its function prototype is:

long AFgetmiscsize(AFfilehandle file, long chunkid)

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, and *chunkid* is a positive long integer miscellaneous chunk ID from the *miscids* array returned by **AFgetmiscids()**.

**AFgetmiscsize()** returns a long integer value that describes the size of the data in the chunk in bytes. This number does not take into account null-terminators in strings, so you will need to add one to the value returned when actually reading string data (see **AFreadmisc()**).

### Reading, Writing, and Seeking Miscellaneous Data

This section describes functions that read and write miscellaneous data and to position the read/write location pointer within the data portion of a miscellaneous chunk. The AFfilehandle structure maintains a logical read/write pointer for each miscellaneous data chunk in the file. Each pointer is initialized to point at the first data byte with the chunk when the AFfilehandle structure is created.

**Tip:** To avoid file corruption, don't copy MISC chunks from one file to another unless the content of those chunks is known. A chunk can contain references to other parts of the file that have been modified by the application, in which case attempting to copy it without properly modifying its contents would cause an error.

**AFreadmisc()** reads data from a given miscellaneous chunk into a buffer, and returns the number of bytes read. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *chunkid* is a positive long integer miscellaneous chunk ID from the *miscids* array returned by **AFgetmiscids()**, *buf* is a pointer to a buffer that will receive the data from the miscellaneous chunk, and *nbytes* is the number of bytes you want to read from the audio file into *buf*, beginning at the current position of *file*'s logical read pointer for the data in *miscid*. **AFreadmisc()** will not read past the end of the chunk's data area. After reading the data, **AFreadmisc()** updates the position of the read/write pointer to point to the data byte following the last one read.

**AFwritemisc()** writes data from a buffer to a given miscellaneous chunk, and returns the number of bytes successfully written. Its function prototype is:

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *chunkid* is a positive long integer miscellaneous chunk ID from the *miscids* array returned by **AFgetmiscids()**, *buf* is a pointer to a buffer that contains the data you want to write to the miscellaneous chunk, and *nbytes* is the number of bytes you want to write to the audio file from *buf*, beginning at the current position of *file's* logical write pointer for the data in *miscid*. **AFwritemisc()** will not write past the end of the chunk's data area. After writing the data, **AFreadmisc()** updates the position of the read/write pointer to point to the data byte following the last one written.

It is up to the application to fill the data area of a chunk with consistent information (for example, if you don't use all the bytes you allocated in a MIDI data chunk, you need to fill the remaining bytes with no-ops).

**AFseekmisc()** moves the logical read/write pointer for a miscellaneous chunk to a specified offset from the beginning of the chunk's data area. Its function prototype is:

```
void AFseekmisc(AFfilehandle file, long chunkid, long offset)
```

where *file* is the AFfilehandle structure previously created by a call to **AFopenfile()** or **AFopenfd()**, *chunkid* is a positive long integer miscellaneous chunk ID from the *miscids* array returned by **AFgetmiscids()**, offset is a nonnegative long integer specifying the number of bytes past the start of the data area the read/write pointer should be moved, and *offset* should always be less than the size of the total data area (in bytes).

**AFseekmisc()** returns the new location of the logical read/write pointer, measured as the number of bytes from the beginning of the chunk data area.

# Audio File Library Programming Tips

This section describes important Audio File Library programming tips:

- "Minimizing Data and File Format Dependence" on page 171 describes how to maximize application compatibility by minimizing format dependence.
- "Preventing Concurrent Access from Multiple Threads" on page 172
  explains how to write a multithreaded AF application in order to
  prevent simultaneous access to an AFfilehandle from multiple threads.
- "Handling Errors in Multithreaded Applications" on page 176 explains how to prevent an error handler from reporting simultaneous errors from a multithreaded application.

## Minimizing Data and File Format Dependence

Currently, the Audio File Library supports the AIFF and AIFF-C file formats. As the AF Library evolves to support new file formats (beyond AIFF and AIFF-C) and new data formats (beyond 2's complement integer and compressed data formats), file-format dependent applications will require more modifications to maintain compatibility than file-format independent programs. Making your application file format independent decreases the likelihood of compatibility problems with future releases of the library and minimizes future modifications. Programming tips presented throughout this chapter call attention to methods you can use to make your application format independent.

# **Preventing Concurrent Access from Multiple Threads**

The AF is not multithread/multiprocessor safe. Making multiple, simultaneous, uncoordinated AF calls on *different* AFfilehandles from different threads is possible and correct. Each AFfilehandle completely encapsulates the state (except for error handling, which is global) needed to perform operations on that AFfilehandle. In contrast, making multiple, simultaneous, uncoordinated AF calls on *the same* AFfilehandle from different threads is currently possible, but it is *not* proper programming practice.

In the following code, two threads are using one AFfilehandle:

#### Thread 1

- Some amount of time
- No semaphore locking

•

AFseekframe(h,track,placel; AFreadframes(h,track,...);

Some amount of time

• No semaphore locking

•

#### Thread2

- Some amount of time
- No semaphore locking

•

AFseekframe(h,track,place2);

AFreadframes(h,track,...);

- Some amount of time
- No semaphore locking

•

It is possible that these calls would be executed in the following order, in which case both threads would read the wrong data.:

```
AFseekframe(h,track,placel); | | AFseekframe(h,track,place2); | | AFseekframe(h,track,place2); | | AFreadframes(h,track,...);
```

The only way to ensure that concurrent operations take place in the correct order is to use a process coordination facility such as semaphore locking.

Proper multithreading looks like this:

#### Thread 1

- •
- Some amount of time
- •

Lock Semaphore that guards h

AFseekframe(h,track,place1;

AFreadframes(h,track,...);

Unlock Semaphore that guards h

- •
- Some amount of time
- •

#### Thread 2

- •
- Some amount of time
- •

Lock Semaphore that guards h

AFseekframe(h,track,place2);

AFreadframes(h,track,...);

Unlock Semaphore that guards h

- •
- Some amount of time
- •

IRIX guarantees that only one of the Lock Semaphore calls will succeed immediately. The thread whose lock does not succeed waits in the Lock Semaphore call (and thus does not proceed to the **AFseekframe()** call) until the other thread has unlocked the semaphore (after it has finished seeking and reading). When the first thread unlocks the semaphore, the thread that is waiting can now proceed.

Follow these steps to add semaphore locking to a multithreaded application:

- 1. Use *usnewsema*(3P) to code to create a semaphore whose value is 1.
- 2. Use *uspsema*(3P) to lock the semaphore.
- 3. Use *usvsema*(3P) to unlock the semaphore.

Example 7-3 is a code fragment that demonstrates how to create a semaphore for protecting critical regions.

## **Example 7-3** Creating a Semaphore

```
#include <ulocks.h>
AFfilehandle h;
                    /* global file handle */
usema_t *HSema;
                    /* global semaphore to protect h */
/* Initialize semaphore support -- do this once. */
   usptr_t *usptr;
   char *arenafile;
   /* Use the fastest type (nondebugging) semaphores. */
   usconfig(CONF_LOCKTYPE, US_NODEBUG);
   /* Create a shared arena to hold the semaphore.
   arenafile = tmpnam(NULL);
   usptr = usinit(arenafile);
   Create the semaphore with count 1 in that arena.
   There is 1 resource (h) initially available.
   HSema = usnewsema(usptr,1);
   /* No need to refer to arena again, so unlink file */
   unlink(arenafile);
   }
```

To use the semaphore created in Example 7-3 do this:

```
Thread 1

• Some amount of time
• Some amount of time
• uspsema(HSema); /* lock */

AFseekframe(h,track,placel; AFseekframe(h,track,place2);

AFreadframes(h,track,...); AFreadframes(h,track,...);

usvsema(HSema); /* unlock */
• Some amount of time
• Some amount of time
```

Semaphore locking can prevent a worst-case scenario such as seeking from the second thread before the first thread has finished reading. Currently, an AF application without semaphores might not cause any problems when making simultaneous, uncoordinated AF calls on the same AFfilehandle from different threads. But this is because—by chance—the CPU scheduler timing has arranged the process timing so that both threads don't use the handle at the same time. Another time, the CPU scheduling might not be favorable, so it's best to protect the critical regions with semaphores.

In summary, you cannot make multiple, simultaneous, uncoordinated AF calls on the same AFfilehandle from different threads, even if the order of execution of those calls does not matter. Doing so is likely to cause a core dump, or at least corruption of the AFfilehandle. The application is responsible for implementing any semaphore protection that is needed; such protection is not built in to the AF calls themselves.

#### **Handling Errors in Multithreaded Applications**

You cannot make multiple, simultaneous, uncoordinated AF calls from different threads that affect the library's global state—namely, the error handler function. If two threads simultaneously try to set the error handler (even if it is the same error handler), the behavior is undefined.

If you write your own error handler and then make multiple, simultaneous, uncoordinated AF calls on different file handles from different threads (and both AF calls issue an error simultaneously), then two instances of your error handler are called in a simultaneous, uncoordinated manner in both threads. If this situation is possible in your program, you should use semaphores in your error handler (in addition to the semaphores in your main program) to prevent simultaneous error reporting or handling.

#### Sample Audio File Program

Example 7-4 contains a listing of *recordexample.c*, in /usr/people/4Dgifts/ examples/dmedia/audio/ program, which records stereo data from an audio port. If you incorporate this code in a program, use the method of rate querying shown in ratequery.c instead of the method used in *recordexample.c*.

**Example 7-4** Recording Stereo from an Audio Port: *recordexample.c* 

```
#include <stdio.h>
#include <signal.h>
#include <dmedia/audio.h>
#include <dmedia/audiofile.h>
 * small example program: "recordexample"
* record an AIFF-C file from an audio input port
 * stop recording when user sends an interrupt
 * file is configured for 16-bit stereo data at the current
      sampling rate of the audio hardware
 * usage: "recordexample <filename>"
* /
int
        caught_sigint;
* catch interrupt signal
* /
static void
catch_sigint()
   caught_sigint++;
main(int argc, char **argv)
                                                                         * /
   char
                *myname;
                                   /* name of this program
   char
               *portname;
                                   /* audio port name
               portconfig;
                                   /* audio port configuration
                                                                         * /
   ALconfig
                                                                         * /
                                   /* audio port
   ALport
                port;
                                   /* audio port channels
   long
                portchannels;
                                   /* audio port sampling rate
                                                                         * /
   long
                portrate;
```

```
/* audio port sample width
long
             portsampwidth;
                                                                        * /
                                  /* audio port sample format
                                                                         * /
long
             portsampfmt;
Affilesetup filesetup;
                                  /* audio file setup
                                                                         * /
AFfilehandle file;
                                  /* audio file handle
                                                                         * /
                                 /* audio file name
char *filename;
                                                                         * /
             filechannels;
                                 /* audio file channels
                                                                         * /
long
                                 /* audio file sampling rate
             filerate;
double
                                                                        * /
             filesampwidth;
                                 /* audio file sample width
                                                                        * /
long
                                 /* audio file sample format
long
            filesampfmt;
                                                                        * /
                                 /* parameter-value buffer
                                                                        * /
long
             pvbuf[2];
                                 /* sample transfer buffer
void
            *buf;
                                                                        * /
                                 /* number of frames written
                                                                        * /
int
             numframeswrit;
                              /* samples transfered per loop */
/* sample frames transfered per loop */
/* samples transfered per loop */
int
             done;
                                 /* flag
int.
             samplesperbuf;
int.
             framesperbuf;
                                 /* samples transfered per sec
int
              samplespersec;
myname = arqv[0];
portname = myname;
if (argc != 2)
    fprintf(stderr, "Usage: %s filename\n", myname);
    exit(1);
}
sigset(SIGINT, catch_sigint);
filename = argv[1];
 * get the global IRIS Audio Processor input rate
pvbuf[0] = AL_INPUT_RATE;
ALgetparams(AL_DEFAULT_DEVICE, pvbuf, 2);
portrate = pvbuf[1];
 * initialize the audio port and audio file configuration
 * /
               = AL_STEREO;
                                           /* port channels
portchannels
portsampwidth = AL_SAMPLE_16;
                                          /* port sample width */
portsampfmt = AL_SAMPFMT_TWOSCOMP;
filedhampels = 2:
                                          /* port sample format */
filechannels
               = 2;
                                           /* file channels
                                                                 * /
filesampwidth = 16;
                                          /* file sample width */
filesampfmt
               = AF_SAMPFMT_TWOSCOMP;
                                         /* file sample format */
```

```
* configure file sample rate to match IRIS audio processor input rate
switch (portrate)
  case AL_RATE_48000: filerate = 48000.0; break;
   case AL_RATE_44100: filerate = 44100.0; break;
   case AL_RATE_32000: filerate = 32000.0; break;
   case AL_RATE_22050: filerate = 22050.0; break;
   case AL_RATE_16000: filerate = 16000.0; break;
   case AL_RATE_11025: filerate = 11025.0; break;
  default:
  case AL_RATE_8000: filerate = 8000.0; break;
}
/*
* compute the number of input samples equal to half a
 * second and allocate a transfer buffer
 * /
               = ((long)filerate) * 2; /* stereo
samplespersec
               = samplespersec / 2; /* half second buffer */
samplesperbuf
                                       /* stereo
framesperbuf = samplesperbuf / 2;
               = (short *)malloc(samplesperbuf * sizeof(short));
buf
/*
 * open the audio port
 * /
portconfig
            = ALnewconfig();
ALsetchannels(portconfig, portchannels);
ALsetwidth(portconfig, portsampwidth);
ALsetqueuesize(portconfig, samplesperbuf);
port = ALopenport(portname, "r", portconfig);
* configure an audio file
* /
           = AFnewfilesetup();
filesetup
AFinitfilefmt(filesetup, AF_FILE_AIFFC);
AFinitchannels(filesetup, AF_DEFAULT_TRACK, filechannels);
AFinitrate(filesetup, AF_DEFAULT_TRACK, filerate);
AFinitsampfmt(filesetup, AF_DEFAULT_TRACK,
                  AF_SAMPFMT_TWOSCOMP, filesampwidth); /*in bits */
/*
* open the audio file
file = AFopenfile(filename, "w", filesetup);
```

```
* play the buffer
   done = 0;
    caught_sigint = 0;
   while (!done && !caught_sigint)
        ALreadsamps(port, buf, samplesperbuf);
        if ((numframeswrit
                 = AFwriteframes(file, AF_DEFAULT_TRACK,
                             buf, framesperbuf)) < framesperbuf)</pre>
            done++;
        }
    }
                         /* this is important: it updates the file header */
   AFclosefile(file);
   ALcloseport(port);
    exit(0);
}
```

# Chapter 8

# Programming with the CD Audio Library

This chapter describes the CD Audio Library, which lets you play and sample audio from CDs using your CD-ROM drive.

# Programming with the CD Audio Library

The IRIS Media Libraries have two libraries that help you retrieve and process digital audio and related information from two sources. This chapter describes the CD Audio Library, *libcdaudio.a*, which gives you access to the data on an audio compact disc (CD), including nonaudio information. Chapter 9, "Programming with the DAT Audio Library," describes the DAT Audio Library, *libdataudio*, which helps you process audio information stored on digital audio tape (DAT).

Because these libraries deal with digital audio information, they contain many analogous routines for processing audio data. But the libraries diverge when it comes to writing audio data and controlling their respective devices. *libcdaudio* includes calls that control the CD-ROM drive; the DAT drive uses the standard IRIX device drivers.

#### In this chapter:

- "CD Audio Library Basics" on page 184 explains basic concepts for using the CD Audio Library.
- "Navigating through a CD" on page 187 explains getting locations from and seeking to locations on a CD.
- "Using the CD-ROM Drive" on page 190 explains how to use the CD-ROM drive for playing audio CDs, reading and parsing CD information, and communicating CD status to the end user.
- "CD Sample Program" on page 196 presents a CD sample program.

### **CD Audio Library Basics**

The CD Audio Library lets you:

- control the CD-ROM drive (eject CDs, prohibit ejection of CDs)
- read or play information from that drive
- parse and process digital information

This section describes the basic concepts that underlie *libcdaudio*. Because both CDs and DATs digitally encode an audio signal as a series of samples, the concepts and terms used when dealing with these media are similar; however, there are differences between the two.

#### CD Frames, Samples, and Subcodes

Per second of playing time, a CD contains 75 CD frames, each containing 588 stereo audio frames (that is, pairs of left and right channel audio samples). A CD frame has both audio and nonaudio information. The sum of the nonaudio information in a frame composes a single complete chunk of *subcode*. When in audio mode and reading from a CD, you need complete subcodes. Thus, in audio mode, a CD frame is the smallest parcel of information you can read from a CD.

To give you controlled access to either the audio data or the subcode in a CD frame, *libcdaudio* hands you a CDFRAME structure:

```
typedef struct cdframe {
   char audio[CDDA_DATASIZE];
   struct subcodeQ ;subcode;
} ;CDFRAME;
```

An audio sample is linearly encoded in a 16-bit two's-complement format. Because a complete stereo audio sample contains two interleaved channels, it takes four bytes of *audio[]* to contain a complete stereo audio sample.

Figure 8-1 shows the structure of a CD audio sample.

least significant	most significant	least significant	most significant
byte, left channel	byte, left channel	byte, right channel	byte, right channel
audio[0]	audio[1]	audio[2]	

Figure 8-1 CD Audio Sample Structure

The byte ordering of the samples in <code>audio[]</code> is the raw data from the CD; its byte ordering is reversed from that on the Indigo workstation. The sampling rate at which CD audio data is originally recorded is 44.1 kHz; therefore, CDDA\_DATASIZE (the size of <code>audio[]</code>) is defined as 2352. This allows for 588 stereo audio samples per CD frame, which, at 75 frames per second, allows for a sample rate of 44.1 kHz.

The subcode member has three information modes:

mode1	for reporting on the track, index, and timing for the current CD track; or, if the current track is the nonaudio lead-in track (see "CD Tracks, Indices, and Time Codes" on page 186), <i>mode1</i> contains a table of contents for the CD
mode2	for reporting the catalog number for the CD as well as an absolute CD frame count
mode3	for reporting the International Standard Recording Code (ISRC) identification information: country, owner, year, and serial number

Thus, the subcodeQ structure in the CDFRAME structure contains a union of three structures: *mode1*, *mode2*, and *mode3*. Which mode is used depends on the information from the CD.

For more information on the CDFRAME structure and the subcodeQ structure, see the *CDFRAME*(4) man page.

#### CD Tracks, Indices, and Time Codes

As many as 99 audio program tracks are allowed on a CD. These tracks are numbered 01 through 99. Two nonaudio tracks of general interest are also available: the lead-in track (numbered 00) and the lead-out track (numbered AA). Track 00, the lead-in track, contains a table of contents in its subcodes.

A track can have up to 99 subdivisions containing audio information. These subdivisions use the index numbers 01 through 99. Index number 00 is used for the pause between the tracks. The time code gives the current minute, second, and CD frame for the current track. The *subcodeQ* structure with *mode1* uses a *cdtimecode* structure to contain time codes.

#### CD Seeking, Reading, and Playing

Accessing information from a CD-ROM drive is analogous to accessing information from a standard disk drive. To read a particular piece of information from the CD, you must move to that location. The process of moving to a location on the CD is known as seeking.

Reading from a CD-ROM drive is analogous to reading from a disk drive—you copy information from the device to a memory-resident buffer for further processing.

Playing the CD is a variation on reading it. But instead of transferring the information to a buffer for processing, the information is dumped out the audio jacks on the back of the CD-ROM drive, with a minimum of buffering and with no real chance to process it. For information on processing audio from a CD through the workstation's audio hardware, see "Reading Audio Data from the CD-ROM Drive" on page 191.

#### **CD Parser**

The parser lets your application change state in response to changes in the subcode data on a CD. This lets you deal with the audio data in a way that is based on its content. To use the parser, you must give it callback routines that can deal the subcode changes that interest you. Then you set up a loop that reads CD frames from the CD and calls the parser for each CD frame.

The parser checks the subcode in every submitted CD frame. If the parts of the subcode you care about have changed from the previous CD frame, the parser executes one of your callbacks and hands it the new subcode information. Within your callback, you can examine the subcode information and change the state of your application as needed.

#### Opening and Closing the CD-ROM Device

The CD-ROM device does not use a standard IRIX device driver. So, a session with the CD-ROM device starts by calling **CDclose()**. For detailed information on these routines, see the man pages *CDopen*(3) and *CDclose*(3).

#### **Controlling the CD-ROM Drive Caddy**

To give your application control over the caddy-eject feature on the CD-ROM drive, *libcdaudio* defines the following routines:

**CDeject()** to eject the caddy from the CD-ROM drive

**CDpreventremoval()** to lock the CD-ROM drive eject button to prevent

end users from ejecting the caddy at an

inopportune moment

**CDallowremoval()** to unlock the CD-ROM drive eject button

For more information on these routines, see the appropriate man pages.

## Navigating through a CD

To move through a CD, you use one of the *libcdaudio* calls: **CDseek()**, **CDseektrack()**, or **CDseekblock()**. But before you can call these routines, you need to know where you are going. For most applications, locations can come from either of two sources, the end user or calculations internal to your application.

Seek destinations can be in any one of three forms:

- integer CD frame counts
- <minute, second, CD frame> integer triples
- "minute:second:CD frame" ASCII strings

The ASCII format is the one you receive from an end user of your application; the other two formats are used for internal calculation.

#### Getting CD Locations from the End User

If your application wants to give end users the option of seeking to a CD location defined in terms of time, your application can prompt the user for the time and then call **CDatomsf()** to convert the ASCII string to a <minute, second, CD frame> triple that you can use for seeking. You can also let the user specify a track number, convert that track number to an integer and seek to that track.

# Getting CD Locations from Calculations Internal to Your Application

Generally, the pure CD frame count is the most convenient format to use when comparing two locations.

To convert to pure CD frame counts, call:

**CDmsftoframe()** to convert a <minute, second, CD frame> triple

into a pure CD frame count

**CDtctoframe()** to generate a pure CD frame count from a

cdtimecode structure

CDatomsf() followed by to convert an ASCII "minute:second:CD frame"

**CDmsftoframe()** string into a pure frame count

You can then make your calculations and determine the destination to which you want to seek. Despite the convenience of pure CD frame counts for calculation, they are not suitable for seeking. To seek, you must call **CDframetomsf()** to convert the pure CD frame count to a <minute, second, CD frame> triple.

It is also possible to make comparisons between locations expressed in terms of minutes, seconds, and CD frames. In that case, you can convert locations into <minute, second, CD frame> triples by calling:

**CDatomsf()** to convert an ASCII string to a

<minute, second, CD frame> triple

**CDframetomsf()** to convert a pure CD frame count to a

<minute, second, CD frame> triple

**CDtctoframe()** followed by to convert a time code to a

**CDframetomsf()** <minute, second, CD frame> triple

After making these calculations, the location is in terms suitable for seeking.

#### **Getting the Current CD Location**

To get your current location within a CD, call **CDgetstatus()**. This routine takes a *CDSTATUS* structure and fills it with information on current track, minute, second, CD frame, and additional data. To make it easier to compare your current location to another location, you should express the locations in terms of pure CD frame counts. But depending on how you got a location, it could be expressed as three separate integers giving the minute, second, and CD frame, or as an ASCII string, or as a *cdtimecode* structure. For more information on this routine, see the appropriate man pages.

#### Seeking to a CD Location

Seeking sets up the read pointer to retrieve data from a particular location on the CD. You can define the seek location in terms of:

track To seek to a track, call **CDseektrack()**.

absolute time To seek to a location defined in terms of minute, second, and

CD frame, call **CDseek()**.

logical block To seek to a location defined in terms of a logical block

number, call **CDseekblock()**. (On a CD-ROM, one logical block contains a single CD frame, which is 588 stereo audio

samples plus one complete subcode.)

To do a series of consecutive seeks, your first seek can be defined in any of the formats mentioned above. But, because all seek routines return the logical block number of the next logical block, it is often more convenient to define the subsequent seeks in terms of logical blocks.

If you want to do all seeks using **CDseekblock()**, but your first seek is defined in terms of time, call **CDmsftoblock()** to convert time to logical block number.

**Note:** Although logical blocks and CD frames are the same size, you cannot use CD frame counts as if they were logical block counts. The CD frame counts are relative to the start of the CD. A logical block count is offset from the start of the CD. In addition, the size of the offset varies from device to device.

#### **Using the CD-ROM Drive**

This section explains how to use the CD Audio Library routines for:

- playing an audio CD from the CD-ROM drive
- reading audio data from the CD-ROM drive
- parsing CD information
- communicating CD status to the end user

#### Playing an Audio CD from the CD-ROM Drive

This section explains how to use these *libcdaudio* routines to play audio from the CD-ROM as if it were a standard CD player:

CDplay() plays an audio CD through CD-ROM audio jacks
CDtogglepause() toggles a CD-ROM drive between pause and play

**CDstop()** stops play of an audio CD in CD-ROM

**CDplaytrack()** plays a single track of an audio CD through CD-

ROM audio jacks

CDplayabs() plays an audio CD through CD-ROM audio jacks

starting at a particular minute, second, and CD

frame

**CDplaytrackabs()** plays a single track of an audio CD starting at a

particular minute, second, and CD frame

When these routines play a CD, they direct the sound to the drive's headphones and to the audio jacks.

#### Reading Audio Data from the CD-ROM Drive

Once you have set the read pointer with a call to one of the seek routines, you are ready to read data from the CD. But how much data should you read at a time in order to create a continuous flow of data from the CD? To determine this, call **CDbestreadsize()**. The returned value of this function is the number of CD frames to request in your read call. To actually read data from the CD, call **CDreadda()**.

Because *libcdaudio* already includes routines for playing audio data from the CD, you might think that you would never need to read from the CD; however, the *libcdaudio* play routines allow for only a very simple CD-player application—one that cannot even display the current program time while the CD is playing.

Thus, if you are writing a real-world application, you probably want to read samples from the CD into the workstation's memory through the CD-ROM's SCSI interface, play the audio samples from the audio hardware using the Audio Library, parse the CD frames for the current program time, and display the program time in a continuously updated field of the control panel for your application.

To do this, you can write your own play routine that executes as a *cd\_audio* callback. You should also write a *cd\_ptime* callback to get the current program time and to update your "program time" display.

#### Controlling the CD Parser

After you have read data from the CD into a buffer, you can start to process it. Typically, how you process the audio data depends on what its associated subcodes tell you about the data. To make it possible for your application to avoid dealing with the complex *CDFRAME* structure directly, *libcdaudio* includes a parser.

If you write a loop that passes all read CD frames through the parser, the parser can examine all CD frames for changes in the subcode. When the parser finds a change (seeing a subcode for the first time counts as a change), it executes the appropriate callback routine—depending on what sort of subcode change occurred—and passes the new subcode data into your callback routine.

The CD parser distinguishes among eight categories of subcode information. Thus, if you are interested in subcode changes for only one category of subcode data, the parser does not bother your application with subcode changes that you consider irrelevant.

#### Allocating and Initializing the CD Parser

To allocate and initialize the parser data structures, call **CDcreateparser()**. To reset the parser after the user changes the CD in the CD-ROM drive, call **CDresetparser()**. This clears out any information the parser has about the last CD frame but leaves the callback routines in place.

#### **Defining Callbacks for the CD Parser**

When you define a callback for the parser, write a function of the form:

```
MyCDSomethingCallBack( void* arg, CDDATATYPES type,
    void* data) {
    /* your code here */
}
```

The parser uses the third parameter to pass in information it reads from the subcodes. The parser uses the second parameter to pass in the type of callback it thinks it is calling. You can use this to assign the same function to different types of callbacks. Internally, you can switch on the type. This

feature is useful if two callbacks are essentially the same, with the exception of a few lines.

The parser does not use the first parameter. You can use that to pass in information if your application needs to call the callback directly.

#### Adding Callbacks to the CD Parser

cd\_catalog

To add callback routines to the parser, call **CDaddcallback()**. If you do not specify a callback for a category, the parser assumes that you are not interested in changes of that type. You can add callbacks that respond to changes in any of the following categories of subcode data:

cd_audio	callbacks respond to changes in the audio data in a CD frame. You can use this class of callback to notify you when you are beyond the lead-in track and have started to see audio samples. When the parser calls this routine, it passes in the audio sample data. If this callback routine is a play routine for your application, it should write the audio sample to an audio port using the Audio Library. See the <i>ALwritesamps</i> (3) man page and Chapter 6, "Programming with the Audio Library."
cd_pnum	callbacks respond to changes in the program number. You can use this callback to notice when you have moved from one program (track) to the next.
cd_index	callbacks respond to changes in the index number. You can use this callback to notice when you have moved from one subsection of a track to the next.
cd_ptime	callbacks respond to changes in the program time. You can use this callback to continuously update a "program time display" in a CD-playing application.
cd_atime	callbacks respond to changes in the absolute time elapsed since the start of the CD. You can use this callback to continuously update your application's information about total elapsed time.

during the lead-in track for the CD.

callbacks respond to changes in the catalog number for the CD. Because this information should not change within the CD, this sort of callback executes only once—typically

*cd\_ident* callbacks respond to changes in the ISRC identification

number for the recording on the CD. Because this information should not change within the CD, this sort of callback executes only once—typically during the lead-in

track for the CD.

*cd\_control* callbacks respond to changes in the control bits. These bits

can tell you things such as whether the CD is copy protected and whether preemphasis is off or on. Because this information should not change within the CD, this sort of callback executes only once—typically during the lead-in

track.

For more information on each callback type, see the *CDaddcallback*(3) man page.

#### **Deleting and Changing a CD Parser Callback**

To delete a callback, call **CDremovecallback()**. To change a callback, call **CDremovecallback()** followed by **CDaddcallback()**.

#### **Parsing CD Frames**

To submit a group of CD frames to the parser, your loop should set up a loop that calls **CDparseframe()** for each frame that you have read into your buffer.

#### Freeing the Memory Allocated for the Parser

If you are done with the parser and want to free the memory it uses, call **CDdeleteparser()** to delete the parser.

#### Communicating CD Status to the End User

In addition to playing a CD or processing the information read from a CD, your application probably needs to tell the user something about the CD (even if it is only the number of the current track). Also, sometimes your application must take data from the end user and convert it to a form that the CD-ROM device can understand.

To get information for the end user, call:

CDgettrackinfo() to get information about a particular track
CDgetstatus() to get information about the CD as a whole

The CD frames, however, sometimes contain information that is not accessible to the routines mentioned above. For example, the subcodes of track 00 on a CD contain a table of contents. To access this information, you can inspect the subcodes in the *CDFRAME* structures, or, better still, you can submit that track to the parser. If you have added callbacks for the categories of subcode information that you want, the parser passes that information into your callbacks.

To help you present the information the parser hands to your callbacks (or that you read directly from a *CDFRAME* structure), *libcdaudio* contains the routines:

**CDsbtoa()** for converting the 6-bit ISRC country and owner code to

an ASCII string

**CDtimetoa()** for expressing the contents of a *cdtimecode* structure as an

**ASCII** string

For more information on the *CDFRAME* structure and the format of its data, set the *CDFRAME*(4) man page.

#### **CD Time Code Conversion Routines**

Other libcdaudio routines that you might find useful are:

**CDatotime()** for converting a CD frame number to a time code **CDatotime()** for converting an ASCII string to a time code

#### CD Sample Program

Example 8-1 contains a listing of *cdsample.c*, a program that lets you copy timed amounts of data from a CD to an audio file.

**Example 8-1** Copying CD Data to an Audio File: *cdsample.c* 

```
* cdsample--command line tool to read audio data off CD,
 * record it in an AIFF file. Hacked together from various
 * other sample programs.
 * Compile with
 * cc -o cdsample cdsample.c -lcdaudio -lds -laudiofile -lm
#include <sys/types.h>
#include <cdaudio.h>
#include <audio.h>
#include <audiofile.h>
#include <stdio.h>
#include <string.h>
Affilehandle audiofile;
openAudioFile(char *filename)
   Affilesetup filesetup;
   filesetup = AFnewfilesetup();
   AFinitfilefmt(filesetup, AF_FILE_AIFFC);
   AFinitchannels(filesetup, AF_DEFAULT_TRACK, 2);
   AFinitrate(filesetup, AF_DEFAULT_TRACK, 44100.0);
   AFinitsampfmt(filesetup, AF_DEFAULT_TRACK, AF_SAMPFMT_TWOSCOMP, 16);
   AFinitcompression(filesetup, AF_DEFAULT_TRACK, AF_COMPRESSION_G722);
   audiofile = AFopenfile(filename, "w", filesetup);
closeAudioFile()
{
   AFclosefile(audiofile);
```

```
writeAudioFile(void *arg, CDDATATYPES type, short *audio)
    AFwritesamps(audiofile, AF_DEFAULT_TRACK, audio, CDDA_NUMSAMPLES);
void parseTime(char *timestr, int *min, int *sec)
    char *tmp, buf[5];
    int n;
    tmp = strchr(timestr, ':');
    if (tmp == NULL) {
        *sec = atoi(timestr);
    } else {
        *tmp = ' \0';
        tmp++;
        *min = atoi(timestr);
        *sec = atoi(tmp);
    }
main(int argc, char **argv)
    CDPLAYER *cd;
    CDPARSER *cdp;
    CDSTATUS status;
    CDTRACKINFO trackinfo;
    CDFRAME buf[12];
    int i, n;
    int track, numframes, frame;
    char *filename;
    char *tmp, strbuf[12];
    int startmin, startsec, endmin, endsec, totalsec;
    extern int errno;
    if (argc != 5) {
        fprintf(stderr, "Usage: cdsample filename track start_time end_time\n");
        exit(1);
    }
    filename = argv[1];
    track = atoi(argv[2]);
```

```
/*
* Note that we do not check if the arguments are sane ...
parseTime(argv[3], &startmin, &startsec);
parseTime(argv[4], &endmin, &endsec);
if ((cd = CDopen(NULL, "r")) == NULL) {
   fprintf(stderr, "Can't open CD device\n");
   exit(1);
if ((cdp = CDcreateparser()) == NULL) {
    fprintf(stderr, "Can't create parser\n");
    exit(1);
}
* Set up a callback function to process the CD data.
* In this case, CDparseframe() will feed the data to the
* writeAudioFile() function (defined above).
CDsetcallback(cdp,cd_audio,(CDCALLBACKFUNC) writeAudioFile, 0);
openAudioFile(filename);
/*
* Determine the number of frames in the requested
* snippet (75 frames/sec)
numframes = ((endmin * 60 + endsec) - (startmin * 60 + startsec)) * 75;
if (CDgetstatus(cd, &status) == 0) {
    fprintf(stderr, "Couldn't get status\n");
    exit(1);
} else {
if (!status.scsi_audio) {
   fprintf(stderr, "This CD-ROM can't do SCSI audio\n");
   exit(1);
}
* Convert relative time (in track) to absolute time
* (on disk) so we can seek to the proper position.
* /
CDgettrackinfo(cd, track, &trackinfo);
totalsec = (trackinfo.start_min + startmin) * 60 +
trackinfo.start_sec + startsec;
startmin = totalsec / 60;
startsec = totalsec % 60;
```

```
CDseek(cd, startmin, startsec, 0);
for (frame=0;frame<numframes;frame += 12) {
    n = CDreadda(cd, buf, 12);
    if (n < 0) {
        fprintf(stderr, "Error reading CD data\n");
        exit(1);
    }
    if (n == 0) /* We're at the end of the disc */
        break;
    for (i = 0; i < 12; i++)
        CDparseframe(cdp, &buf[i]);
    }
CDclose(cd);
closeAudioFile();
exit(0);
}</pre>
```

# Chapter 9

# Programming with the DAT Audio Library

This chapter describes the DAT Audio Library, which lets you play, record, and sample audio from digital audio tapes (DATs) using your DAT drive.

# Programming with the DAT Audio Library

This chapter describes the DAT Audio Library, *libdataudio*, which you can use to process audio information stored on digital audio tape (DAT).

#### In this chapter:

- "DAT Audio Library Basics" on page 203 explains basic concepts for using libdataudio.
- "Navigating through a DAT" on page 206 explains getting locations from and seeking to locations on a DAT.
- "Using the DAT Drive" on page 209 explains how to use the DAT drive for playing and recording DATs, reading, writing and parsing DAT information, and communicating DAT status to the end user.
- "DAT Sample Program" on page 217 presents a DAT sample program.

### **DAT Audio Library Basics**

The DAT Audio Library (*libdataudio*) supports processing the data from a digital audio tape (DAT). Because the device driver for the DAT drive is a standard IRIX tape device driver, the *libdataudio* library does not need the special positioning and status calls. Instead, you can use the standard **open()**, **close()**, **read()**, **write()**, and **ioctl()** system calls.

This section describes the basic concepts that underlie *libdataudio*. Because both CDs and DATs digitally encode an audio signal as a series of samples, the concepts and terms used when dealing with these media are similar; however, there are some differences between them.

#### **DAT Frames, Samples, and Subcodes**

A DAT contains 33.33 DAT frames per second of playing time. A DAT frame has both audio and nonaudio information. The sum of the nonaudio information in a DAT frame composes a single complete DAT subcode. When in audio mode and reading from a DAT, you need complete subcodes. Thus, in audio mode, a DAT frame is the smallest parcel of information you should read from a DAT.

To give you controlled access to either the audio data or the subcode in a DAT frame, *libdataudio* hands you a *DTFRAME* structure:

```
typedef struct dtframe {
   char audio[DTDA_DATASIZE];
   struct dtsubcode sc;
} DTFRAME;
```

A DAT audio sample is linearly encoded in a 16-bit two's-complement format. Because a complete stereo audio sample contains two interleaved channels, it takes four bytes of *audio[]* to contain a complete stereo audio sample (see Figure 9-1).

least significant byte, left channel	most significant byte, left channel	least significant byte, right channel	most significant byte, right channel
audio[0]	audio[1]	audio[2]	audio[3]

Figure 9-1 DAT Audio Sample Structure

The byte ordering of audio sample frames in *audio[]* is based on the raw data from the DAT; its byte ordering is reversed from that on the IRIS workstation. DTDA\_DATASIZE (the size of *audio[]*) is defined as 5760. This allows for 1440 audio sample frames per DAT frame, which, at 33.33 DAT frames per second, is enough to deal with audio sampled at rates of up to 48 kHz.

The subcode member uses a *dtsubcode* structure to contain the subcode read from the DAT. The subcodes contain information on sampling frequency, the number of channels, table of contents, catalog number, and more. For more information on the *dtsubcode* structure, see the *DATFRAME*(3) man page.

#### **DAT Audio Program Numbers and Indices**

A DAT can have as many as 99 audio programs, each typically corresponding to a single song or musical piece. These programs are numbered 01 through 99. An audio program can have up to 99 subdivisions containing audio information. These subdivisions use the index numbers 01 through 99. Index number 00 is used for the pause between the audio programs.

#### **DAT Run Time, Absolute Time, and Program Time**

A time code gives the hour, minute, second, and DAT frame offset into a DAT. When dealing with program time, the time code is a measure of the time elapsed since the start of the audio program. When dealing with absolute time, the time code measures the time elapsed since the start of the DAT. When dealing with run time, the time code measures the time elapsed since the beginning of the recording and contains several audio programs.

#### **DAT Seeking and Reading**

Accessing information from a DAT drive is analogous to reading information from a standard tape drive. To read a particular piece of information from the DAT, you must move to that location. The process of moving to a location on the DAT is known as seeking. Reading from the DAT is analogous to reading from a tape drive. You copy information from the device to a memory-resident buffer for further processing.

#### **DAT Parser**

The parser lets your application change state in response to changes in the subcode data on a DAT. This lets you deal with the audio data in a way that is based on its content. To use the parser, you must give it callback routines that can deal with the subcode changes that interest you. Then you set up a loop that reads DAT frames from the DAT and calls the parser for each DAT frame.

The parser checks the subcode in every submitted DAT frame. If the parts of the subcode you care about have changed from the previous DAT frame, the parser executes one of your callbacks and hands it the new subcode information. Within your callback, you can examine the subcode information and change the state of your application as needed.

#### Opening and Closing the DAT Device for Audio

The DAT device driver is a standard IRIX device, so you can use the generic **open()**, **close()**, and **ioctl()** calls that you would use for any other tape device; however, unlike a standard tape drive, the DAT drive has an audio mode in addition to a straight data mode.

To put the DAT drive in audio mode, use **ioctl()** with MTIOCTOP and an *mtop* type structure, but set the *mt\_count* member of the *mtop* structure to 1 before submitting that *mtop* structure to **ioctl()**. For example:

```
struct mtop mt_com;
mt_com.mt_op = MTAUD;
mt_com.mt_count = 1; /* 1 == audio mode, 0 == data mode */
ioctl(fd, MTIOCTOP, &mt_com);
```

## Navigating through a DAT

To move through a DAT tape, you use **ioctl()**, a standard IRIX system call. But before you can call **ioctl()**, you need to know where you are going. For most applications, destinations can come from either of two sources, the end user or calculations internal to your application.

Destinations from the end user come to your application in the form of ASCII strings. Destinations from internal calculations typically come in the form of a DAT frame count or, sometimes, as four values that specify the location in terms of hours, minutes, seconds, and DAT frames. Unfortunately, these forms are not suitable for seeking, so you must convert them before you can use them.

#### **Getting DAT Locations from the End User**

If your application wants to give end users the option of seeking to a DAT location defined in terms of time, your application can prompt the user for the time and then call **DTatotime()** to convert the string to a time code that you can submit to **ioctl()** for seeking.

# **Getting DAT Locations from Calculations Internal to Your Application**

Generally, the pure DAT frame count is the most convenient format to use when comparing two locations.

To convert to pure DAT frame counts, call:

DTtctoframe() to extract a pure DAT frame count from a

dttimecode structure

DThmsftoframe() to convert hours, minutes, seconds, and DAT

frames to a pure DAT frame count

**DTatohmsf()** followed to convert an ASCII string to a pure DAT frame

by **DThmsftoframe()** count

You can then make your calculations and call **DTframetotc()** to convert the DAT frame count to a time code suitable for seeking.

It is also possible to make comparisons between locations expressed in terms of hours, minutes, seconds, and DAT frames. In that case, you can convert all locations into hours, minutes, seconds, DAT frames format by calling:

DTatohmsf() to convert an ASCII string to hours, minutes,

seconds, and DAT frames

DTframetohmsf( to convert a pure frame count to hours,

minutes, seconds, and frames

**DTtctoframe()** followed by to convert a time code to hours, minutes,

**DTframetohmsf()** seconds, and frames

After making your calculations, convert the destination to a time code suitable for seeking by calling **DThmsftoframe()** followed by **DTframetotc()**.

#### Seeking to a DAT Location

To seek to a location on a DAT, call **ioctl()** with MTSETAUDIO and an *mtaudio* type structure.

To specify the type of seek, set the *seektype* member of the *mtaudio* type structure to the appropriate MTAUDPOSN\_\* constant:

MTAUDPOSN\_PROG to seek to a program number
MTAUDPOSN\_ABS to seek to an absolute time
MTAUDPOSN\_RUN to seek to a running time

MTAUDPOSN\_PTIME to seek to a program time (within program)

To seek to a particular audio program on the DAT, set *seektype* to MTAUDPOSN\_PROG, and use *pno1*, *pno2*, and *pno3* members to pass in the three BCD numbers that identify the audio program you want. Program numbers range from 001 to 799. The *pno1* member contains the most significant digit and *pn3* contains the least significant digit. Thus, to seek to program 578, set the *pn\** members as follows:

```
struct mtaudio AudioProgNum;
AudioProgNum.pn1 = 5;
AudioProgNum.pn2 = 7;
AudioProgNum.pn3 = 8;
```

To seek to a location on the tape defined in terms of time, set the *mtaudio seektype* member to MTAUDPOSN\_ABS, MTAUDPOSN\_RUN, or MTAUDPOSN\_PTIME and then specify the time location in the *mtaudio* members:

atime for MTAUDPOSN\_ABSrtime for MTAUDPOSN\_RUNptime for MTAUDPOSN\_PTIME

These *atime*, *rtime*, and *ptime* members contain structures of type *mtaudtimecode*:

```
struct mtaudtimecode {
    unchar hhi:4, hlo:4; /* hours */
    unchar mhi:4, mlo:4; /* minutes */
    unchar shi:4, slo:4; /* seconds */
    unchar fhi:4, flo:4; /* DAT frame # */
};
```

The *hhi* and *hlo* members expect two digits that specify the hour to which you want to seek. The valid range for these two digits is from 00 to 99. The *mhi* and *mhl* expect the two digits that specify the minute to which you want to seek. The valid range for these two digits if from 00 to 59. The *shi* and *slo* expect the two digits that specify the second to which you want to seek. The valid range for these two digits is from 00 to 59. The *fhi* and *fhl* expect the two digits that specify the DAT frame to which you want to seek. The valid range for these two digits is from 00 to 33.

#### **Using the DAT Drive**

This section explains how to use the DAT Audio Library routines for:

- playing a DAT
- recording a DAT
- reading and writing audio data from a DAT
- parsing DAT information
- communicating DAT status to the end user

#### Playing a Tape in the DAT Drive

Playing audio from a DAT is a little more complicated than playing a CD. For example, the sample rate for all CDs is 44.1 kHz; however, DAT audio may have been recorded at a sampling rate of 48 kHz, 44.1 kHz, or 32 kHz. Fortunately, a DAT records its sampling rate in the subcodes at the start of the tape, so you can read this sampling rate from the DAT before you must write DAT audio samples to the audio port.

In outline, a simple DAT-playing application must:

- 1. Define a callback routine for *dt\_sampfreq*. When the parser calls this routine, it passes in the frequency just read from the tape. Your callback should set a global variable to the frequency it gets from the parser. (See the *DTaddcallback*(3) man page.)
- 2. Define a callback routine for *dt\_audio*. When the parser calls this routine, it passes in the audio data from the DAT frame just parsed. The callback routine should write this data to the audio port using the sampling rate set by the *dt\_sampfreq* callback.
- 3. Open the audio port.
- 4. Open the DAT drive.
- 5. Create a parser.
- 6. Add your *dt\_sampfreq* and *dt\_audio* callbacks to the parser.
- 7. Read samples from the DAT.
- 8. Parse the samples.
- 9. Write the samples to an audio port using the Audio Library.

When the application first starts reading the tape, it sees the frequency, calls your *dt\_sampfreq* callback, and hands it the sampling frequency. As the parser continues to parse DAT frames, it also sees the audio data and executes your *dt\_audio* callback for each new DAT frame containing audio.

For an example of a simple program that plays a tape in the DAT drive, see "DAT Sample Program" on page 217. For more information on using the audio port, see Chapter 6, "Programming with the Audio Library."

#### Making DAT Recordings for Playback on the DAT Drive

When making recordings on DAT recorders that you want to play on a Silicon Graphics DAT drive, you must make sure you record at least one of the time codes. Most recorders will let you record audio without any time codes, so be certain you record the time codes. Record in standard mode; the DAT drive does not support long play (LP) mode or 4-channel (4CH) mode tapes.

### Reading Audio Data from the DAT Drive

To read audio data from the DAT drive, you need to open the DAT drive and put it in audio mode. Then you can call the standard IRIX **read()** system call as you would for any other tape device. The only complicating factor is that you need to ensure that you read complete DAT frames. This is not particularly difficult if you declare your receiving buffer to be an array of *DTFRAME* structures.

#### For example:

```
DTFRAME MyDATbuffer[4];
```

declares a buffer of four *DTFRAME* structures. If you then do a read such as:

```
n = read( MyDATtapeDevice, MyDATbuffer, sizeof(MyDATbuffer)
);
```

you read in complete DAT frames and can easily access those complete DAT frames when you want to parse them.

### Writing Audio Data to the DAT Drive

To write audio data to the DAT drive, you need to open the DAT drive and put it in audio mode. Then you can call the standard IRIX **write()** system call as you would for any other tape device. Writing the tape is just a matter of writing DAT frames to the tape.

But setting the contents of the DAT frames is not just a matter of gathering together your audio samples. You must write subcode information that specifies things such as the sampling rate at which the audio was recorded. You must also update the DAT time code for each DAT frame that you write to the tape.

To help you set the subcode information for the DAT frames you want to write, *libdataudio* contains these routines:

**DTsetdate()** to set a date pack to the current time (useful for timestamps)

**DTinctime()** to increment a DAT time code

**DTtcvalid()** to check that a time code is valid (use it after calling

DTinctime())

For more information on the time code routines, see the appropriate man pages. For information on what you can write into the DAT frame subcodes, see the *DTFRAME*(4) man page. For additional information about properly writing DAT subcodes, see the DAT specification.

#### Ensuring that your DAT Recording Is Recognized as Audio

The DAT drive determines whether a tape is audio by looking for valid audio DAT frames. These frames must contain at least one valid time code field (absolute time, run time, or program time). When making recordings on DAT recorders that you want to later play on the Silicon Graphics DAT drive, you must make sure you record one of these time codes and that you record in standard mode.

#### Recording the DAT Lead-in Area

The DAT specification requires that a tape begin with a special lead-in area of 100 DAT frames. Recording 100 frames ensures that the real recording will not begin over the plastic leader on the tape.

The following procedure provides the proper lead-in area:

- create an empty DTFRAME
- 2. set the program number contained in the DAT frame to 0x0BB (beginning-of-tape, or BOT, code)
- 3. set the START bit in the control ID
- 4. set the subcode packs to 0x0AA (readable, not valid)
- 5. fill the audio data block with zeros
- 6. rewind the tape and repeatedly write the DAT frame at least 100 times

#### Recording Digital Audio over Digital Data Storage (DDS) Tapes

This section explains special precautions that must be taken when recording audio onto a tape that has previously been used as a data (DDS) tape.

When you insert a DDS tape into the DAT drive, it is rewound to the logical beginning-of-tape (BOT). On data tapes, the logical BOT differs from the physical BOT by approximately 10 centimeters (30 seconds). If you attempt

to write data to the drive in audio mode, writing begins at the logical BOT. When you then rewind and play this tape, there is an initial 30-second gap before playback starts. If the tape is removed and then reinserted into a DAT drive, it is recognized as a data tape because DDS format data exists between the physical BOT and the DDS logical BOT.

**Note:** With the current DAT drives (firmware revision 2.63), the following procedure is necessary to work around the problem: Check to see if the tape in the drive is DDS media and at BOT. If so, switch the drive to audio mode and write a frame of data to move the tape off logical BOT, and then issue a rewind. This rewinds the tape all the way back to the physical BOT.

#### **Example Programs Demonstrating DAT Recording**

Two sample programs are available to help you with DAT recording:

- cdtodat.c, in /usr/people/4Dgifts/examples/dmedia/cd+dat
   This program copies audio from a CD to a DAT. It contains example code for recording to DAT, including handling of the lead-in area and recording over data tapes.
- verifydat.c, in /usr/people/4Dgifts/examples/dmedia/cd+dat
   This program verifies that a DAT has been recorded correctly and has continuously running absolute time code.

#### Controlling the DAT Parser

After you have read in data from a DAT, you can start to process it. Typically, how you process the audio data depends on what its associated subcodes tell you about the data (for example, the sample rate at which the audio was recorded). If you want, you can directly examine the subcode associated with each DAT frame and respond appropriately.

The *DTFRAME* structure, however, is large and complicated and subject to change. *libdataudio* includes a parser so that your application can avoid dealing with the *DTFRAME* structure directly.

If you write a loop that passes all the read DAT frames through the parser, the parser can examine all the DAT frames for changes in the subcode. When the parser finds a change (seeing a subcode for the first time counts as a

change), it executes the appropriate callback routine—depending on what sort of subcode change occurred—and passes the new subcode data into your callback routine.

The DAT parser distinguishes among 14 categories of subcode information. Thus, if you are interested in subcode changes for only one category of subcode data, the parser does not bother your application with subcode changes that you consider irrelevant.

#### Allocating and Initializing the DAT Parser

To allocate and initialize the data structures for the DAT parser, you must call **DTcreateparser()**.

To reset the parser after the user changes the tape in the DAT drive, call **DTresetparser().** This clears out any information the parser has about the last DAT frame but leaves the callback routines in place.

#### **Defining Callbacks for the DAT Parser**

When you define a callback for the parser, you must write a function of the form:

```
My_dat_SomethingCallBack( void* arg, DTDATATYPES type,
   void* data)
   {
    /* your code here */
}
```

The parser uses the third parameter to pass in information it read from the subcodes. The parser uses the second parameter to pass in the type of callback it thinks it is calling. You can use this to assign the same function to different types of callbacks. Internally, you can switch on the type. This feature is useful if two callbacks are the same except for a few lines.

The parser does not use the first parameter. You can use that to pass in information if your application needs to call the callback directly.

#### Adding and Removing DAT Parser Callbacks

To add callback routines to the parser, call **DTaddcallback()**. If you do not specify a callback for a category, the parser assumes you are not interested in changes of that type.

You can add callbacks that respond to changes in any of the following categories of subcode data:

callbacks respond to changes in the audio data in a DAT
frame. You can use this class of callback to notify you when
you have gotten past the lead-in track and have started to
see audio samples. When the parser calls this routine, it
passes in the audio sample data. If this callback routine is a
play routine for your application, it should write the audio
sample to an audio port using the Audio Library. See the

with the Audio Library."

 $dt\_pnum$  callbacks respond to changes in the program number. You

can use this callback to notice when you have moved from

ALwritesamps(3) man page and Chapter 6, "Programming

one program (track) to the next.

*dt\_index* callbacks respond to changes in the index number. You can

use this callback to notice when you have moved from one

subsection of a track to the next.

dt\_ptime callbacks respond to changes in the program time. You can

use this callback to continuously update a "program time

display" in a DAT-playing application.

*dt\_atime* callbacks respond to changes in the absolute time elapsed

since the start of the DAT. You can use this callback to continuously update your application's information about

total elapsed time.

*dt\_rtime* callbacks respond to changes in the run time elapsed since

the start of a recording on the DAT. You can use this callback to continuously update your application's information about total elapsed time since the start of a recording.

dt_prortime	callbacks are like $dt\_rtime$ callbacks in that they respond to changes in the elapsed run time—however, the parser hands the callback more information than it gives to a $dt\_rtime$ callback—this type of callback is intended for professional uses
dt_mainid	callbacks respond to changes in the contents of the ID field
dt_sampfreq	callbacks respond to changes in the subcodes that describe the sampling frequency for the recording on the DAT
dt_toc	callbacks respond to changes in the subcode data that describe the table of contents for the tape
	<b>Note:</b> When parsing the DAT, you should note a separate subcode for each entry in the table of contents.
dt_date	callbacks respond to changes in the timestamp for a recording
dt_catalog	callbacks respond to changes in the DAT catalog number
dt_ident	callbacks respond to changes in the ISRC identification number for the recording on the DAT
dt_probinary	callbacks respond to changes in the IEC (SMPTE) or Pro DIO time codes

For more information on each subcode category, see the *DTaddcallback*(3) man page and the DAT specification.

# **Deleting or Changing a DAT Parser Callback**

To delete a callback, call **DTremovecallback()**. To change a callback, call **DTremovecallback()** followed by **DTaddcallback()**.

# **Parsing DAT Frames**

To submit a group of DAT frames to the parser, set up a loop that calls **DTparseframe()** for each DAT frame that you have read into your buffer.

#### Freeing the Memory Reserved for the DAT Parser

If you are done with the parser and want to free the memory it uses, call **DTdeleteparser()** to delete the parser.

# Communicating DAT Status to the End User

Whether you get status information for the DAT directly from the *DTFRAME* structure or from one of your parser-callback routines, you need to convert that information to an ASCII string.

libdataudio includes these conversion routines:

**DTsbtoa()** to convert a 6-bit country and owner code to an ASCII string

**DTtimetoa()** to convert a time code to an ASCII string

**DTpnotodec()** to convert a BDC program number to a decimal, which you

can then check and convert to ASCII if appropriate

For more information on these routines, see the relevant man pages.

### **DAT Sample Program**

This section contains *datplay.c*, a simple program for reading and processing DAT data.

### Playing a DAT

Example 9-1 reads samples from the DAT and uses the parser and two callbacks to process the data read. One callback, **frequency()**, extracts the sampling rate from the subcodes on the DAT. The other callback, **playaudio()**, extracts audio samples from the frames and writes them to the audio port.

#### **Example 9-1** Reading DAT Samples

```
/* DAT example from digital audio and MIDI programming guide */
#include <stdio.h>
#include <sys/fcntl.h>
#include <sigfpe.h>
                        /* Floating point exception error handling package */
                        /* with this you'll need to compile w/ -lfpe */
                        /* DAT audio library. */
#include <dataudio.h>
                        /* with this you'll need to compile w/ -ldataudio */
#include <audio.h>
                        /* audio library */
                        /* with this you'll need to compile w/ -laudio */
                                                 /* Number of samples you'll */
static int sampsperframe = DTDA_NUMSAMPS48K;
                                                 /* get in 1 DAT frame when */
                                                 /* the sampling rate is at */
                                                 /* 48kHz.
Alport audioport;
                                                 /* Audio port to output the */
                                                 /* DAT sample data */
/* Our dt_audio callback */
/* It gets called when there's a change in the audio data in a frame. */
playaudio(void *arg, DTDATATYPES type, short *audio)
   ALwritesamps(audioport, audio, sampsperframe);
        /* Send the audio samples read out to the audio port */
/* Our dt_sampfreq callback. */
^{\prime \star} It gets called when there's a change in the subcodes that describe the ^{\star \prime}
/* sampling frequency for the recording on the DAT. */
frequency(void *arg, DTDATATYPES type, int *freq)
{
   switch (*freq)
     case DT_FREQ48000:
       sampsperframe = DTDA_NUMSAMPS48K;
                                                 /* Number of samples you'll */
                                                 /* get in 1 DAT frame when */
                                                 /* the sampling rate is at */
                                                 /* 48kHz.
       break;
```

```
case DT FREO44100:
                                               /* Number of samples you'll */
      sampsperframe = DTDA_NUMSAMPS44K;
                                               /* get in 1 DAT frame when */
                                               /* the sampling rate is at */
                                                /* 44kHz.
      break;
     case DT_FREQ32000:
      sampsperframe = DTDA_NUMSAMPS32K;
                                               /* Number of samples you'll */
                                               /* get in 1 DAT frame when */
                                               /* the sampling rate is at */
                                                /* 32kHz.
      break;
main()
int tape = open("/dev/nrtape", O_RDONLY);
                /* Open the file descriptor for reading data off of */
                /* DAT Tape assumed to be /dev/nrtape.
                                       /* Initialize DAT parser. */
DTPARSER *dtp = DTcreateparser();
DTFRAME buf[4];
                                        /* Will describe content of current */
                                        /* DAT frame. This is what gets */
                                        /* sent to the DAT parser.
                                                                            * /
                                        /* See man pagfe for DATFRAME for
                                        /* detailed info.
struct mtop mt_com;
                                        /* Message structure for magntic
                                        /* tape device interface for use in */
                                        /* passing to the ioctl command. De-*/
                                        /* fined in /usr/include/sys/mtio.h */
                                        /* See mtio and dataudio man pages. */
  audioport = ALopenport("DAT Test", "w", 0); /* Open audio port. */
   if (dtp)
                                                /* Check for DAT parser. */
      DTsetcallback(dtp, dt_audio, (DTCALLBACKFUNC)playaudio, 0);
                                /* Set up function to be called when there's */
                                /* a change in audio information.
     DTsetcallback(dtp, dt_sampfreq, (DTCALLBACKFUNC)frequency, 0);
                                /* Set up function to be called when there's */
                                /* a change in sampling frequency information.*/
```

```
/* Make sure we get sane underflow exception handling */
    sigfpe_[_UNDERFL].repls = _ZERO;
    handle_sigfpes(_ON, _EN_UNDERFL, NULL, _ABORT_ON_ERROR, NULL);
                          /* See man page for sigfpe */
}
else
                         /* Can't do much without a DAT parser. */
  exit(1);
                          /* Check for tape reading OK. */
if (tape >= 0)
{
   mt_com.mt_op = MTAUD;
                        /* Set up MT(io) AUD(io) message to tell */
                          /* DAT drive to turn audio mode on/off. */
                         /* 1 == audio mode, 0 == data mode
   mt_com.mt_count = 1;
   ioctl(tape, MTIOCTOP, &mt_com);    /* Perform M(agnetic) T(ape) I/O */
                                  /* C(ontrol) T(ape) OP(eration). */
                                  /* This is needed for both
                                  /* reading and writing audio to */
                                  /* or from the DAT drive. */
   for (;;)
       n = read(tape, buf, sizeof(buf));
                                         /* Read frame of DAT audio */
                                         /* data from the tape's */
                                                                 * /
                                         /* file descriptor.
       if (n < 0)
           exit(2);
       if (n == 0)
                                    /* We're at the end of the tape */
          break;
       for (i = 0; i < 4; i++)
          DTparseframe(dtp, &buf[i]);
                                         /* Sort out what info was in */
                                         /* the frame we just read and */
                                         /* invoke the callbacks we */
                                         /* specified previously.
                                                                     */
   exit(0);
exit(3);
                         /* Can't do much without a tape to read from */
```

}

# Chapter 10

**Programming with the MIDI Library** 

This chapter describes the MIDI Library, which provides MIDI access.

# Programming with the MIDI Library

The MIDI Library, *libmd.so*, provides an API for sending, receiving, and processing musical instrument digital interface (MIDI) messages through the serial interface of Silicon Graphics IRIS Indigo, Indigo<sup>2</sup>, and Indy workstations.

### The MIDI Library features

- timed input and output of MIDI data
- buffered I/O with user-adjustable buffering time
- active sensing and system-exclusive data handling
- simultaneous access to MIDI devices from multiple programs
- the ability to have multiple input and output streams open concurrently
- a correlation to other media streams with unadjusted system time (UST)
- sample applications online in /usr/people/4Dgifts/examples/dmedia/midi

Hands-on experiences are presented throughout this chapter:

- "Hands-On MIDI Output Experience" on page 234 demonstrates sending MIDI messages.
- "Hands-On Multiplexed MIDI I/O Experience" on page 240 demonstrates a MIDI thru box.
- "Hands-On MIDI File Player Experience" on page 244 demonstrates a multithreaded MIDI file player with a graphical user interface.
- "Hands-On MIDI and Audio Synchronization Experience" on page 246 demonstrates synchronized MIDI playback and recording.

#### In this chapter:

- "MIDI System Architecture" on page 224 describes the system configurations and I/O interfaces for MIDI.
- "MIDI Library Basics" on page 231 discusses basic MIDI concepts and the main MIDI Library data structures.
- "Opening and Closing MIDI Ports" on page 232 explains how to open and close MIDI ports and how to get a file descriptor for a MIDI port.
- "Programming MIDI I/O" on page 234 explains how to implement basic MIDI I/O functions.
- "Multiplexing MIDI I/O with File Descriptors" on page 240 explains how to use file descriptors to multiplex synchronous I/O.
- "Controlling MIDI Timing" on page 241 describes timestamping modes and explains how to specify and scale MIDI tempo.
- "Synchronizing MIDI I/O with Other Media" on page 246 explains how to use Unadjusted System Time (UST) for synchronizing MIDI timing with other media streams. It contains synchronized MIDI recording and playback applications that use the MIDI and Audio libraries in conjunction.

# **MIDI System Architecture**

This section describes system configurations for MIDI development and the MIDI input and output interfaces.

### **Configuring Your System for MIDI Development**

The most essential peripheral for MIDI development is a serial-to-MIDI converter. You can use any Apple Macintosh® compatible serial-to-MIDI converter. Many MIDI converters include additional features, including SMPTE-to-MIDI conversion (which is useful for synchronizing MIDI to tape and film), and built-in MIDI patchbays for switching between multiple MIDI inputs and outputs. You should choose your serial-to-MIDI converter based on the functionality you expect your users to require.

Once you have selected a MIDI converter, you will need some MIDI devices to attach to it. The kind and number of MIDI devices you choose to create your MIDI network depends largely on the scope of the application you are writing and your budget. A single keyboard synthesizer may be sufficient for your needs if you are writing a very simple sequencer, but for more complex programs, you should have a keyboard, several rack-mount synthesizer modules, an alternate MIDI controller (such as a wind controller), and a mixer with enough channels for all the instruments you have (so you can hear the results).

If you are not satisfied with listening through headphones or through your workstation's internal speaker, you should probably invest in an amplifier and/or a pair of powered speakers. A multi-track tape recorder may also be useful for testing your application if it will ultimately be used for recording music. You should plan on testing your application using the kind of equipment that you anticipate the end users of your application to have.

Figure 10-1 shows one possible MIDI setup.

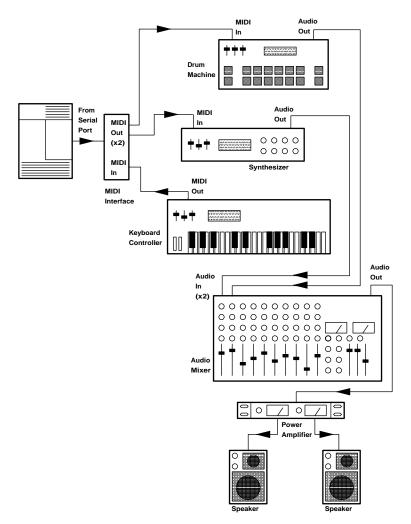


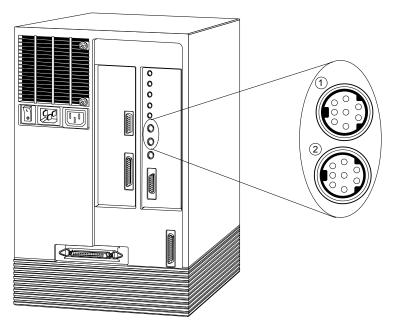
Figure 10-1 MIDI Setup

# **Connecting Devices to MIDI I/O Interfaces**

The MIDI Library is currently supported on the Indigo, Indigo<sup>2</sup>, and Indy workstations. The two serial ports on your workstation are configured for MIDI from the Port Setup tool, as described in "Configuring Serial Ports for MIDI WIth the Port Setup Tool" on page 229.

Any Apple Macintosh-compatible serial-to-MIDI interface operates when connected to either or both of the serial ports. Many of these interfaces offer additional useful features, such as SMPTE time-code conversion and integrated software-configurable MIDI patching.

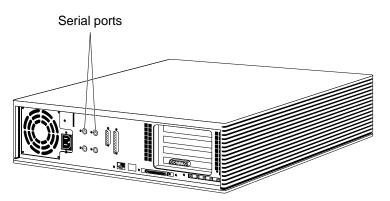
Figure 10-2 shows the serial ports on the back panel of the Indigo workstation.



**Figure 10-2** Serial Ports on the Back Panel of the Indigo Workstation

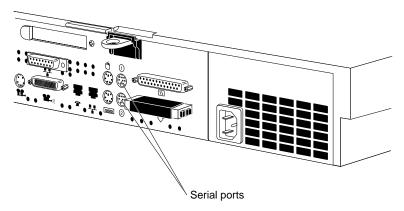
**Note:** Do not use the keyboard port for MIDI.

Figure 10-3 shows the serial ports on the back panel of the  $\rm Indigo^2$  workstation.



**Figure 10-3** Serial Ports on the Back Panel of the Indigo<sup>2</sup> Workstation

Figure 10-4 shows the serial ports on the back panel of the Indy workstation.



**Figure 10-4** Serial Ports on the Back Panel of the Indy Workstation

### **Configuring Serial Ports for MIDI With the Port Setup Tool**

Before you can run a MIDI application, you must first configure your workstation's serial ports for MIDI by using the Port Setup tool. The Port Setup tool provides a GUI for configuring connections to your system's serial ports. Once set up, a serial port remains configured for MIDI, even if the system reboots, until you reset it from the Port Setup tool.

To configure a serial port for MIDI:

- 1. Open the System Manager and select the System Administration tools from the Tools menu.
- 2. Click the Port Setup icon, shown in Figure 10-5.



Figure 10-5 Port Setup Icon

Figure 10-6 shows the Port Setup tool.



Figure 10-6 Port Setup Tool

- 3. Select the available serial port by clicking its icon.
- 4. Click Connect.

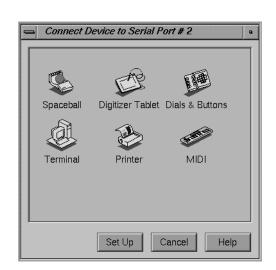


Figure 10-7 shows the device connections available from the Port Setup tool.

Figure 10-7 Serial Port Connections

- 5. Select the MIDI device by clicking its icon.
- 6. Click Set Up.

The system displays the MIDI Port Configuration menu shown in Figure 10-8, asking you to confirm whether you want to start MIDI.

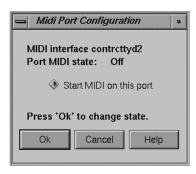


Figure 10-8 MIDI Port Configuration

7. Click OK to start MIDI on the selected port.

### **MIDI Library Basics**

This section discusses fundamental MIDI concepts and describes the primary data structures used by the MIDI Library.

MIDI is a control protocol, as opposed to a data protocol, meaning that a MIDI network does not carry audio signals, but rather instructions that tell MIDI instruments how to behave. MIDI information is transmitted in the form of an event: a control instruction (or message), combined with time information (called a timestamp). Typical messages are *note on* and *note off* (describing the beginning and ending time of a certain musical note) and values for continuous controllers such as sustain pedals, modulation wheels, and pitch bend controllers.

# **Initializing MIDI Library Programs**

Before calling any MD Library routines, you must initialize the MIDI Library by calling **mdInit()**, which returns the number of available MIDI ports.

If the MIDI daemon is not running, **mdInit()** returns –1. See "Configuring Serial Ports for MIDI WIth the Port Setup Tool" on page 229 for instructions on configuring the serial ports and starting and stopping the MIDI daemon.

# **Compiling and Linking MIDI Library Programs**

To compile a MIDI Library program, enter:

```
cc -g MLsample.c -o MLsample -lmd
```

You should also link with any other libraries, such as the Audio Library, that your application uses.

### MIDI Library Error Handling

All *libmd* functions return –1 on error, and set *oserror*(3C) appropriately.

### **MIDI Library Programming Model**

The MIDI Library has two basic data structures:

MDport An opaque structure containing information about the state

of MIDI data and timing, as well as the state of all options

for the port.

MDevent A public structure containing fields for regular and

system-exclusive MIDI messages, timestamps, and message

lengths.

# **Opening and Closing MIDI Ports**

The MDport, or MIDI port, is the basic MIDI I/O structure in the MIDI Library. An MDport provides a one-way (input-only or output-only) connection to a MIDI device. Each port can transmit or receive on as many as 32 independent MIDI channels, 16 per serial port. The use of separate MIDI channels allows complex orchestration of MIDI instruments, when each MIDI device is "tuned" to a different channel.

# Getting the Name of an Available MIDI Port

Once the MIDI Library has been initialized by calling **mdInit()**, you can get the name of an available port by calling **mdGetName()**:

```
char *mdGetName(int portno)
```

Then you can build a menu of available MIDI ports for your application. **mdGetName()** returns the name string associated with *portno* or NULL, if *portno* does not refer to an existing port.

For example, if MIDI has been initialized with the following command:

```
startmidi -n ttyd2 -d /dev/ttyd2
```

mdInit() returns 1, and mdGetName(0) returns the "ttyd2" string.

### **Opening and Closing MIDI Input and Output Ports**

The MIDI Library has task-specific calls for opening MDports: **mdOpenInPort()** opens an input port and **mdOpenOutPort()** opens an output port. Their function prototypes are:

```
MDport mdOpenInPort(char *name)
MDport mdOpenOutPort(char *name)
```

Each returns a handle to the appropriate type of port.

Use the name returned by **mdGetName()** to indicate a particular MIDI device to which a port is to be connected, as demonstrated in Example 10-1.

#### **Example 10-1** Opening MIDI Input and Output Ports

The initial state of a newly-opened port is undefined, except for the timestamping mode, which is MD\_DELTASTAMP.

You can open up to 64 MDports, less the number of devices. For example, if you have both serial ports configured for MIDI, you can open 62 MDports. When a port is no longer needed, call **mdClosePort()** to close the port and free its associated resources.

# **Programming MIDI I/O**

This section explains how to implement the most basic tasks for MIDI applications: sending and receiving MIDI messages.

### Hands-On MIDI Output Experience

To begin, try playing a sound through the MIDI equipment connected to your system. The sample application plays a sound by sending a MIDI message. To test whether you can send MIDI output, try this:

- 1. If you have not already done so, connect your MIDI equipment to your workstation's serial port, and configure the port for MIDI, as described in "Connecting Devices to MIDI I/O Interfaces" on page 227.
- 2. to launch the *scale* program, which sends a musical scale through the MIDI output.

**Note:** The application will not launch if you don't have your MIDI equipment connected and set up.

See *scale.c*, in /usr/people/4Dgifts/examples/dmedia/midi/simple to view the code that plays the musical scale.

A tone is played through the MIDI output by specifying the note to play and its duration, and then sending a *note on* event to sound the tone, followed by another event to end the sound (often, a *note on* with zero velocity is used to silence a note).

Example 10-2 shows the **playnote()** routine from *scale.c*, which creates a MIDI event structure (mdEvent) named *mdv*. The *mdv* structure contains a 3-byte message, a timestamp, and the message length. The 3-part message consists of the MD\_NOTEON event, or'ed to a channel, followed by the note and its velocity. See "About MIDI Events" on page 235 for a description of the mdEvent structure.

The message is given an initial timestamp of 0. After the specified time interval has elapsed, the note's velocity is set to zero and then a zero velocity *note on* message is sent to silence the output.

#### **Example 10-2** Sending a MIDI Message

```
#include "dmedia/midi.h"
playnote ( MDport port, char note, unsigned long long time,
           char channel, char velocity )
    mdEvent mdv;
    mdv.msg[0] = MD_NOTEON | (channel & 0xf);
    mdv.msg[1] = note;
    mdv.msg[2] = velocity;
    mdv.stamp = 0;
    mdv.msglen = 3;
    if (mdSend(port, &mdv, 1) < 0) {
        exit(-1);
    mdv.stamp = time;
    mdv.msg[2] = 0;
    if (mdSend(port, &mdv, 1) < 0) {
        exit(-1);
    }
}
```

MIDI event structures are described in "About MIDI Events" on page 235.

#### **About MIDI Events**

MIDI events are contained in the MIDI Library *mdEvent* data structure:

msg

is an array of characters representing the data of a non-system-exclusive message, which can include status, note, and controller information and is from 1 to 3 bytes in length.

sysexmsg	is a pointer to a string of characters representing a block of system-exclusive (SYSEX) data, which can include bulk data such as instrument patch configuration parameters and can be of arbitrary length. When SYSEX data is received, $msg[0]$ is set to MD_SysEx (0xf0), and then the actual data storage is allocated with $mdMalloc()$ ; similarly, it must be released with $mdFree()$ .
stamp	is the timestamp of the event, in nanoseconds or in ticks, if you are using one of the tick modes.
msglen	is used by system-exclusive messages to indicate the length of the SYSEX packet, or when sending multiple messages in a single event. For single events, <i>msglen</i> should be set to 0.

The timestamp for the MIDI event, which is the time at which the event did or should occur, is reckoned from either a fixed time or the previous event's time. The MIDI Library supports two types of timestamping:

- *relative* stamping, in which time is reckoned for all events as an interval from an initial specified time. This is useful for sequencers.
- *delta* stamping, in which time for each individual event is reckoned as the interval since the last event occurred. This is useful for insertions of events in lists.

See "Controlling MIDI Timing" on page 241 for information about setting the timestamping mode and other parameters.

### **Sending and Receiving MIDI Events**

This section explains how to send and receive MIDI events.

#### **Sending MIDI Events**

To send a MIDI event from a MIDI output port, call **mdSend()**. Its function prototype is:

```
int mdSend(MDport port, MDevent *buf, int count)
```

Depending on the port's timestamping mode, MIDI events have either relative or delta timestamps, or no timestamps at all. If temporal buffering is

used, **mdSend()** blocks, waiting for the output to catch up, until the amount of time represented by the sum of the timestamps in the event buffer exceeds the *timeout* value set by **mdSetTemporalBuffering()**.

If no errors occur, **mdSend()** returns the number of messages successfully sent; otherwise, it returns either 0, indicating that an error occurred and no messages were sent, or –1 times the number of messages not sent. **mdSend()** sleeps if the output queue size limit is exceeded.

### **Receiving MIDI Events**

To receive a MIDI event into a MIDI input port, call **mdReceive()**. Its function prototype is:

```
int mdReceive(MDport port, MDevent *buf, int count)
```

**mdReceive()** allocates storage for messages coming into the designated port, except for system-exclusive messages; these require the application to allocate and free the necessary storage. **mdReceive()** copies the message(s) and timestamp(s) into a buffer, and returns either the number of messages read from the given port or -1, if an error occurred.

#### **Handling System-Exclusive MIDI Events**

SYSEX messages are received in chunks of up to 1 kilobyte. To check for SYSEX messages, scan for the EOX marker in *buf.sysexmsg[buf.msglen-1]*. When receiving sysex messages, *buf.msg[0]* is set to 0xf0 for each chunk received, while the actual data is stored in *buf.sysexmsg*.

#### **Printing MIDI Events**

To print the messages in the MIDI event buffer, you must first convert them to a human-readable format by calling **mdPrintEvent()**. Its function prototype is:

```
int mdPrintEvent(char *buf, mdEvent *evbuf, int count)
```

buf

is a pointer to buffer allocated by the application. It should be large enough to contain the formatted representation of all the events in *evbuf*. Eighty bytes per message is sufficient.

evbuf is a pointer to the event buffercount is the number of events to format

### The message format is:

```
timestamp : channel : status type (string) : byte 1 : byte 2
```

If the message is a *note on* or *note off,* then the note name (for example, A3) is printed in the field occupied by byte 1; otherwise, the numeric value is printed.

### **Processing MIDI Event Messages**

A MIDI message is an array of 2 or 3 bytes. The first byte contains the status in the high nibble, and the channel in the low nibble. The remaining 1 or 2 bytes contain the values.

### **Setting and Getting MIDI Message Status**

The status byte determines the type of message and its length. Table 10-1 lists the name, length, and purpose of each status byte.

**Table 10-1** MIDI Message Status Bytes

Status	Length	Byte 1	Byte 2
MD_CHANNELMODESELECT	2		
MD_CHANNELPRESSURE	2		
MD_CONTROLCHANGE	2	Controller number	Controller value
MD_NOTEOFF	3		
MD_NOTEON	3	Note number	Velocity
MD_PITCHBENDCHANGE	3	MSB	LSB
MD_POLYKEYPRESSURE	3		
MD_PROGRAMCHANGE	2	Program number	Unused

To set the status, call **mdSetStatus()**; to get the status, call **mdGetStatus()**.

#### **Setting and Getting MIDI Message Channel**

The channel is the low nibble of the high byte. It takes a value of 0 through 15, which corresponds to a MIDI channel range of 1 through 16.

The functions for setting and getting the channel are:

```
int mdGetChannel(char *msg)
void mdSetChannel(char *msg, int x)
```

There are sixteen channels for each serial port (device 0 and device 1). Channels 0–15 are sent on device 0, and channels 16–31 are sent on device 1.

#### **Setting and Getting MIDI Message Value**

Byte1 and Byte2 are the values associated with the message. Message-specific information (for example, note names for *note on* and *note off* messages and channel numbers for channel messages), is contained in these bytes. The functions that set and get MIDI message values are:

```
void mdSetByte1(char *msg, int x)
void mdSetByte2(char *msg, int x)
int mdGetByte1(char *msg)
int mdGetByte2(char *msg)
```

See the following references for MIDI message codes:

- *MIDI 1.0 Detailed Specification* and *Standard MIDI Files 1.0*, International MIDI Association, 5316 W. 57th St., Los Angeles, CA 90056.
- MIDI Sequencing in C, by Jim Conger, ISBN 1-55851-045-1, M & T Books, 1989. Available from:

M&T Books A Division of M&T Publishing, Inc. 501 Galveston Drive Redwood City, CA 94063

# Multiplexing MIDI I/O with File Descriptors

You can multiplex MIDI input and output by using the IRIX *select*(2) system call to wait on MD file descriptors. Using this technique allows your workstation to function as a MIDI thru box.

### Hands-On Multiplexed MIDI I/O Experience

To send and receive MIDI messages through your workstation:

- 1. If you have not already done so, connect your MIDI equipment to your workstation's serial port, and configure the port for MIDI, as described in "Connecting Devices to MIDI I/O Interfaces" on page 227.
- 2. to launch the *thru* sample program, which receives MIDI events and in turn sends them out through an mdOutport.

**Note:** The application will not launch if you don't have your MIDI equipment connected and set up.

See *thru.c*, in /*usr/people/4Dgifts/examples/dmedia/midi/simple* to view the code that implements MIDI-thru capability.

#### **Getting a File Descriptor for a MIDI Port**

File descriptors can be used with the IRIX <code>select(2)</code> or <code>poll(2)</code> system calls to multiplex input and output of MIDI messages with other I/O devices. To get a file descriptor for an MDport, call <code>mdGetFd()</code>, which returns a file descriptor associated with the port:

```
int mdGetFd(MDport port)
```

Example 10-3 is an excerpt from *thru.c* that demonstrates putting the file descriptor returned by **mdGetFd()** into a file descriptor set and using *select* to wait on the file descriptor set. Using *select* requires including the *sys/select.h* header file.

#### **Example 10-3** Using MIDI File Descriptors

# **Controlling MIDI Timing**

The MIDI Library provides for timed input and output of MIDI data. Messages are timestamped on input and are scheduled for output on the basis of their timestamp. Output scheduling can be disabled.

You can synchronize I/O for MIDI streams with other media streams by correlating timestamps in terms of unadjusted system time (UST).

# **Controlling MIDI Timing Mode**

You can choose to time MIDI events using either actual time or musical beats, also called ticks. The default mode of an MDport is delta timestamping, which measures the time elapsed from the previous event, but you can reset it to use any mode. To get a port's timestamping mode, call mdGetStampMode(); to set a port's timestamping mode, call mdSetStampMode().

The MIDI Library has three modes in which timing is controlled by actual time:

MD\_NOSTAMP Causes the output timestamps to be ignored.

Input timestamps are undefined.

MD\_DELTASTAMP Causes each input event to be timestamped with

the number of milliseconds from the previous event, and interprets output timestamps in the

same way.

MD\_RELATIVESTAMP Causes input timestamps to be marked relative to

a time specified by **mdSetOrigin()**. Output timestamps are also reckoned against the same

origin time.

The MIDI Library has two modes in which timing is controlled by beats:

MD\_RELATIVETICKS Allows the ticks to be reckoned from an origin

time that is set with **mdSetOrigin()**.

MD\_DELTATICKS Uses simple delta timestamps. Ticks are defined in

terms compatible with Standard MIDI files; for example, pulses per quarter note (PPQ). Ticks are

controlled by **mdSetTempo()** and

mdSetDivision().

Time is measured from the time the MDport was opened. You can reset the reference time by calling **mdSetOrigin()**, which sets the start time to the 64-bit UST that you specify. The result depends on the value used:

O Sets the start time to the system's current UST.

< 0 Sets the start time to the number of nanoseconds before the

current UST. This allows streams of files to be restarted in

the middle of the data

> UST Sets the start time to some time in the future.

Upon successful completion, **mdSetOrigin** returns 0; otherwise it returns –1 and sets an error code that you can retrieve with *oserror*(3C). To get the start time, call **mdGetOrigin()**.

Setting the reference time to match a UST value is useful in MIDI recording, for setting the recording start time to correspond to the arrival of an audio

signal at the input jacks. See "Hands-On MIDI and Audio Synchronization Experience" on page 246 for a demonstration of this technique.

### **Controlling MIDI Tempo**

You can vary the tempo for ports whose timestamping mode is in ticks. Tempo is expressed in microseconds per beat, as in Standard MIDI Files (SMF). Tempo and division values specify the conversion from MIDI clock ticks to real time values for the MIDI driver. Divisions represent the subsamples per beat, or pulses per quarter note (PPQ). This is useful for MIDI sequencing.

To set the tempo, call **mdSetTempo()**; to get the tempo, call **mdGetTempo()**.

To set the divisions per beat (pulses per quarter note), call **mdSetDivision()**; to get the divisions per beat, call **mdGetDivision()**.

To convert a timestamp from ticks to nanoseconds, taking into account the port's tempo, call **mdTicksToNanos()**. Similarly, to convert from nanoseconds to ticks, call **mdNanosToTicks()**.

Sometimes you need to adjust the tempo when audio is not synchronized or to compensate for a slow tape deck when recording MIDI. You can specify a tempo scale factor by calling **mdSettemposcale()**. When messages whose timestamps are expressed in ticks are written to a port that has a tempo scale factor, the timestamps are multiplied by the scale factor before being queued for output. This allows for nondestructive tempo matching.

### **Controlling MIDI Output Buffering**

The MIDI driver buffers data according to time. An application that responds to user interaction must compensate for the fact that events can be sent to the MIDI port faster than they can actually be transmitted. By default, the MIDI library does not allow a process to get more than 2 seconds ahead of the actual output.

You can control how much playback can get ahead of data transmission by specifying the amount of temporal buffering. To set the number of

milliseconds an application can get ahead of its output, that is, the time to drain an MDport, call **mdSetTemporalBuffering()**.

When the event timestamps exceed the specified *timeout* value, **mdSend()** sleeps until the output catches up.

To determine the amount of time an application can get ahead of its output, call mdGetTemporalBuffering().

You can pause output momentarily or even completely silence output when necessary.

To stop pending output on a port and return the UST value or the tick of the last message sent to or from a port, call **mdPause()**. **mdPause()** immediately halts output.

Sometimes you may need to do more than simply pause the output. You can send a panic event for a given port by calling **mdPanic()**, which sends an *all notes off* message and a *reset controllers* message on each channel.

#### Hands-On MIDI File Player Experience

The MIDI file player sample application illustrates the use of the timing concepts presented in this section—it lets the user change the tempo of a MIDI file, pause playback, or drag a slider to start playback at a random location.

To play MIDI files using the MIDI file player:

- 1. If you have not already done so, connect your MIDI equipment to your workstation's serial port, and configure the port for MIDI, as described in "Connecting Devices to MIDI I/O Interfaces" on page 227.
- 2. Enter Mfp to launch the MIDI file player.

**Note:** The application will not launch if you don't have your MIDI equipment connected and set up.

See *player.c++*, in */usr/people/4Dgifts/examples/dmedia/midi/mfp* to view the playback source code.

The MIDI file player application uses three threads: one to manage the user interface (UI), one to manage the playback, and one to update the song position. These threads use shared memory; semaphores are used to protect critical regions so that both processes don't try to access the same data simultaneously.

The playback thread waits on the semaphore. Processes that are waiting on a semaphore are queued on a first-come, first-served basis. When the UI process acquires the semaphore, playback stops; when it releases the semaphore, playback starts. An important point to note is that the playback loop uses **uscpsema()**, which tests the semaphore and returns immediately if it can't be acquired. This provides an opportunity where it is known to be safe to pause the output. Without doing this, it is possible to send the pause command without having the playback thread acknowledge it, because it is busy sending data.

Another interesting point to note is that error checking is performed to determine whether the application is sending more data than can be handled; if so, the playback thread releases the semaphore and polls the MIDI port until enough data has drained to allow more data to be sent.

The song position thread loops until either the stop button is hit or the song finishes. If a sequence is paused, the position resets to 0, so you need to save the starting position, then add it back in when resuming playback.

The MIDI file player uses non-blocking I/O so that stopping a sequence is possible without flushing currently queued data. If **mdSend()** is waiting for either room or time to send data, that data is sent as soon as the currently waiting data is flushed.

When setting the division and tempo, the division must always be set first, because the constants set in the driver by the tempo change, depending on the division.

When setting the origin time, putting the position of the file in a signed quantity avoids a compiler warning. Multiplying the start time by –1 allows the file to start playing in the middle without waiting for the first timestamp to expire.

When pausing playback, keep track of the timestamp of the MIDI message that was most recently sent and add it to the previous pause time, so that playback will resume from the proper location in the file.

# Synchronizing MIDI I/O with Other Media

You can synchronize I/O for MIDI streams with other media streams by correlating timestamps in terms of unadjusted system time (UST).

One technique is to set the MIDI port to use relative timestamping, and then use **mdSetOrigin()** to set the port's origin time to match the UST of the media stream to which you want to synchronize.

Alternatively, you can obtain the UST for a MIDI event and compare it to the UST of another media stream counter. To return the UST or tick of the last event sent out, call mdTell().

### Hands-On MIDI and Audio Synchronization Experience

To try a synchronized audio and MIDI application:

- 1. If you have not already done so, connect your MIDI equipment to your workstation's serial port, and configure the port for MIDI, as described in "Connecting Devices to MIDI I/O Interfaces" on page 227.
- 2. Click on *syncrecord* to launch a sample application that demonstrates synchronized audio and MIDI recording.

**Note:** The application will not launch if you don't have your MIDI equipment connected and set up.

See *recordmidi.c*++ and *playmidi.c*++, in /usr/people/4Dgifts/examples/dmedia/midi/syncrecord to view the code that implements the synchronized MIDI record and play application.

#### **PART THREE**

# **Video Programming**

Chapter 11, "Video Basics," explains basic video concepts that apply to both the Video Library and the IndigoVideo Library.

Chapter 12, "Getting Started with the Video Library," describes the Video Library and explains how to use it to perform video input and output for workstations equipped with standard and optional Silicon Graphics video hardware.

Chapter 13, "Using VL Controls," describes how to use VL controls to set video parameters for data transfer and video effects.

Chapter 14, "VL Event Handling," describes how to handle video events using the Video Library.

# Chapter 11

# **Video Basics**

This chapter explains how to use the VL controls to set and adjust video parameters.

# Video Basics

Computer graphics and video differ in a number of ways; understanding the differences can help you produce better results with the VL and your Silicon Graphics video option. This chapter introduces some of the important terms and concepts used in conjunction with video. For more detail about a particular term, see the Glossary included in this guide.

Video differs from computer graphics in these ways:

- interlacing
- broadcast standards
- color encoding
- video signals
- tape formats

# Interlacing

Unlike the way the screen is typically drawn for computer graphics, most video signals are *interlaced*: each time the video screen is refreshed, only half of the horizontal lines are drawn. That is, each *frame* is composed of two *fields*.

During one screen refresh, the video monitor draws the first field, which contains all the odd-numbered lines; during the next refresh, it draws the second field, which contains all the even-numbered lines. Therefore, two refresh cycles are required to draw one frame.

The display rate of interlaced video signals can be measured either in terms of field rate, or refresh rate, or in terms of frame rate, which equals half of the field rate, because each frame contains two fields.

Figure 11-1 shows a frame and its two fields for NTSC, the broadcast standard used in North America and some other parts of the world.

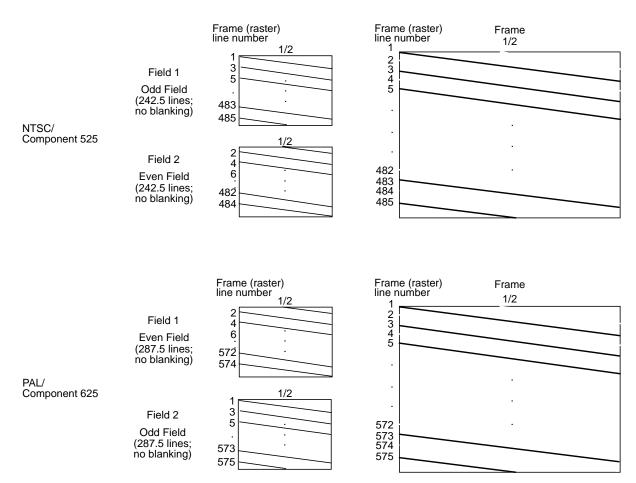


Figure 11-1 Fields and Frame

In contrast, the Silicon Graphics workstation monitor is typically noninterlaced: it draws every line each time it refreshes the screen. Refresh rates vary, depending on the type of monitor your Silicon Graphics workstation has. The video output capability of the graphics subsystem for some Silicon Graphics workstation models supports interlaced monitor formats, including component RGB at 525 and 625 lines per frame.

## **Broadcast Standards**

Broadcast standards, or video timing formats, are ways of encoding video information for broadcast to television receivers. These standards are also used to describe the display capabilities of video monitors and are thus also called video timing formats or video output formats (VOFs). The three broadcast standards are:

NTSC Named after the National Television Systems Committee,

which developed it, this standard is used in all of North and South America, except Brazil, and in much of East Asia.

PAL (Phase Alternated by Line) This standard is used in western

Europe, including the United Kingdom but excluding

France, and in East Asia, including Australia.

SECAM (Sequentiel Couleur avec Memoire) This standard is used in

France, eastern Europe, the Near East and Mideast, and

parts of Africa and the Caribbean.

**Note:** NTSC implementations can vary slightly by country; PAL and SECAM implementations can vary considerably.

NTSC employs a total of 525 horizontal lines per frame, with two fields per frame of 262.5 lines each. Each field refreshes at 60Hz (actually 59.94Hz). NTSC encodes brightness, color, and synchronizing information in one signal.

PAL employs a total of 625 horizontal lines per frame, with two fields per frame of 312.5 lines per frame. Each field refreshes at 50Hz. PAL encodes brightness, color, and synchronizing information in one signal also, but in a different way from NTSC.

SECAM transmits the same number of lines at the same rate as PAL, but transmits each color difference signal on alternate lines, using the frequency modulation of the subcarrier.

These numbers of horizontal lines—525 and 625, respectively—are a shorthand description of what actually happens. For NTSC, the first (odd) field starts with a whole line and ends with a half line; the second (even) field starts with a half line and ends with a whole line. Each NTSC field contains 242.5 active lines and 20 lines of vertical blanking.

Similarly, for PAL, the first (even) field starts with a half line and ends with a whole line; the second (odd) field starts with a whole line and ends with a half line. Each PAL field contains 287.5 active lines and 25 lines of vertical blanking.

In each case, the numbers 525 and 625 refer to transmitted lines; the active video lines are fewer—typically, 485 for NTSC and 575 for PAL. The remaining lines are used for delimiting frame boundaries and for synchronization and other information.

To minimize frame flickering and reduce the bandwidth of the video signal, the active video lines are *interlaced*, as explained earlier in this chapter.

NTSC and PAL can be recorded digitally; these recording techniques are referred to as D2 525 (digital NTSC) and D2 625 (digital PAL).

# **Color Encoding**

Color-encoding methods are:

- RGB (component)
- YUV (component)
- YIQ (component)
- YC (separate luminance (Y) and chrominance (C)), YC-358, YC-443, S-Video
- composite video

#### **RGB**

RGB is the color-encoding method used by most graphics computers, as well as some professional-quality video cameras. The three colors red, green, and blue are generated separately; each is carried on a separate wire.

#### YUV

YUV, a form of which is used by the PAL video standard and by Betacam and D1 cameras and VCRs, is also a component color-encoding method, but in a different way from RGB. In this case, brightness, or *luminance*, is carried on a signal known as Y, and color, or *chrominance*, is carried on the U and V signals. The two chrominance signals U and V are two-phase amplitude-modulated: the U component modulates the subcarrier at an angle of 0 degrees, but the V component modulates it at 90 degrees or 180 degrees on alternate lines. The color burst is also line-alternated at +135 and -135 degrees relative to the U signal.

The YUV matrix multiplier derives colors from RGB via the following formula:

$$Y = .299R + .587 G + .114 B$$
  
 $C_R = R-Y$   
 $C_B = B-Y$ 

in which Y represents luminance and R-Y and B-Y represent the color difference signals used by this format. In this system, which is sometimes referred to as Y/R-Y/B-Y, R-Y corresponds to  $C_R$  and V, and B-Y corresponds to  $C_B$  and U. R-Y and B-Y are obtained by subtracting luminance (Y) from the red (R) and blue (B) camera signals, respectively.  $C_R$ ,  $C_B$ , V, and U are derived through different normalization methods, depending on the video format used. The U and V signals are carried on the same signal.

YUV component color encoding can be recorded digitally, according to the CCIR 601 standard; this recording technique is referred to as D1.

#### YIQ

YIQ color encoding, which is typically used by the NTSC video format, encodes color onto two signals called I and Q (for intermodulation and quadrature, respectively). These two signals have different phase modulation in NTSC transmission. Unlike the U and V components of YUV, I and Q are carried on different bandwidths.

The YIQ formula is as follows:

```
Y = .299 R + .587 G + .114 B (the same as for YUV) 
 I = .596 R - .275 G - .321 B 
 Q = .212 R - .523 G + .311 B
```

# YC, YC-358, YC-443, or S-Video

YC, a two-wire signal, results when I and Q are combined into one signal, called chrominance. YC-358 is the most common NTSC version of this luminance/chrominance format; YC-443 is the most common PAL version. These formats are also known as S-Video; S-Video is one of the formats used for S-VHS<sup>TM</sup> videotape recorders.

# **Composite Video**

The composite color-encoding schemes combine the brightness and color signals into one signal for broadcast. NTSC and PAL both combine brightness and color but use different methods.

Figure 11-2 shows the relationships between color-encoding methods and video formats.

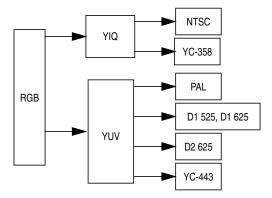


Figure 11-2 Relationships Between Color-encoding Methods and Video Formats

# **Video Signals**

The video signal, whatever the broadcast standard being used, carries other information besides video (luminance and chrominance) and audio. For example, horizontal and vertical synchronization information is required, as well as a color phase reference, which is called color sync burst. Figure 11-3 shows a composite video signal waveform.

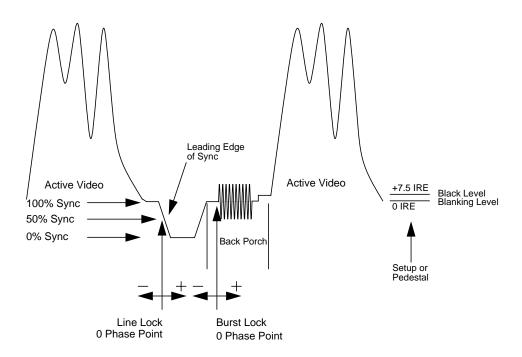


Figure 11-3 Composite Video Waveform

# **Videotape Formats**

Videotape recorders are available for analog and digital recording in various formats. They are further classified by performance level, or market: consumer, professional, and broadcast. In addition, during postproduction (editing, including addition of graphics), the original footage can be transferred to digital media; digital videotape formats are available for

composite and component video formats. There are no official standards for videotape classifications.

Table 11-1 summarizes the formats.

**Table 11-1** Tape Formats and Video Formats

Electronics	Consumer	Professional	Broadcast	Postproduction
Analog	VHS cassette (composite)	U-Matic (SP) cassette, 3/4- inch (composite)	Type C reel-to- reel, 1-inch (composite)	
	S-VHS (YC, composite)		Type B (Europe) (composite)	
	S-Video (YC-358)	S-Video (YC-358)		
	Beta (composite)		Betacam (component)	
	8mm (composite)		Betacam SP (YUV, YIQ, composite)	
	Hi-8mm <sup>TM</sup> (YC, composite)	Hi-8mm (YC)	MII <sup>TM</sup> (YUV, YIQ, composite)	
Digital				D1 525 (YUV)
				D1 625 (YUV)
				D2 525 (NTSC)
				D2 625 (PAL)

Although the VL and other software for Silicon Graphics video options do not distinguish between videotape formats, you need to know what kind of connector your video equipment uses. For example, the Galileo board has composite and S-Video connectors.

Most home VCRs use composite connectors. S-Video, on the other hand, carries the color and brightness components of the picture on separate wires; hence, S-Video connectors are also called Y/C connectors. Most S-VHS and Hi-8mm VCRs feature S-Video connectors.

# Chapter 12

# **Getting Started with the Video Library**

This chapter explains how to begin developing a VL application, including setting up paths, nodes, and controls for transferring video data.

# Getting Started with the Video Library

The Video Library (VL) is a collection of device-independent C language calls for Silicon Graphics<sup>®</sup> workstations equipped with video options, such as Sirius Video <sup>TM</sup>, Indigo<sup>2</sup> Video <sup>TM</sup>, Indy Video <sup>TM</sup>, or Galileo Video <sup>TM</sup>, or workstations equipped with on-board video (VINO <sup>TM</sup>: video in, no out), such as Indy <sup>TM</sup>. The VL includes generic video tools, including simple tools for importing and exporting digital data to and from current and future Silicon Graphics products, as well as to and from third-party video devices that adhere to the Silicon Graphics architectural model for video devices.

VL calls enable you to perform video teleconferencing on platforms that support it, to blend computer-generated graphics with frames from videotape or any video source, and to present video in a window on the workstation screen and to digitize video data.

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

Topics in this chapter include:

- VL Features
- VL System Software Architecture
- VL Architectural Model of Video Devices
- VL Programming Model
- Opening a Connection to the Video Daemon and Setting up a Data Path
- Setting Parameters for Data Transfer to or from Memory
- Displaying Video Data Onscreen
- Transferring Video Data to and from Devices
- Ending Data Transfer

The chapter concludes with example code illustrating a simple screen application and frame grabs (video to memory, memory to video, and continuous frame capture).

## **VL Features**

## How the VL Works with Hardware

The VL includes calls for querying features of all supported Silicon Graphics video options. The VL supports conversion from one video format to another (for example, YUV to RGB or RGB to YUV).

In some cases, the VL can support multiple devices of the same or of different types. For example, the CHALLENGE $^{\text{TM}}$  architecture supports multiple Sirius Video boards; Indy supports its built-in video (VINO) and the Indy Video option.

#### **How the VL Works with Other Software**

The Video Library works with other Silicon Graphics libraries, such as the OpenGL, the IRIS GL and the IRIS ImageVision Library (IL). Software supplied with optional video hardware provides additional video capabilities through extensions to the VL. For example, Sirius Video software includes controls specific to that hardware.

The VL allows programs to get events 60 times per second on a quiescent system; it also enables programs to share resources or to gain exclusive use of resources. It supports input and output of video data to or from locked-down memory at the nominal frame rate. Frame rate depends on the capabilities of the hardware you are using.

**Note:** The VL does not depend on the X Window System, but you can use X Window System libraries or toolkits to create a windowing interface.

# **VL System Software Architecture**

This section describes features of these VL system components and tools:

- · video daemon
- generic video tools
- library and header files

Figure 12-1 diagrams the interaction between the VL, the video daemon, the kernel, the hardware, and the X Window System server.

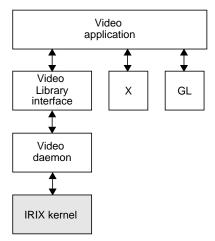


Figure 12-1 VL System Components

The VL communicates with the IRIX kernel for device initialization, vertical retrace, setup, and maintenance of any device-supported direct memory access (DMA).

Besides these components, the VL includes a collection of applications that support device configuration and control setting and retrieval, generic tools that display video on a workstation, and video control panels.

#### Video Daemon

The video daemon, /usr/etc/videod, which has device-dependent and device-independent portions, handles video device management and status information.

#### **Device Management**

Management that the video daemon performs includes:

multiple device management

Some hardware devices support multiple video products in one system. The video daemon is responsible for establishing and coordinating the availability of all video devices installed.

multiple client access to multiple devices

The library supports connections from multiple client applications and manages their access to a limited number of video devices.

dispatching events

As events are handled and noted by devices, the daemon notifies applications that have expressed interest in those events.

handling events

As events are generated by the various devices, the daemon initiates any action required by an event before it hands the event off to interested applications.

maintaining exclusive use

Types of data or control usage for video clients in a Video Library application are *Done Using*, *Read-only*, *Lock*, and *Shared*. These usage levels apply only to write access on controls, not read access. Any application can open and read the control's values at any time.

client cleanup on exit

When a client exits or is terminated abnormally, its connection to the daemon is broken; the daemon performs any cleanup required of the system. Any exclusive-use modes that have been set are cleared; interested clients are notified that the device is no longer in exclusive use. Controls set by the client might persist, but are not guaranteed to remain after the client closes the connection.

#### Status Information

Status information for which the video daemon is responsible includes:

system status of video devices

The video devices installed in a system can be queried as to availability and control status.

- video positioning (offset) information
- control setting and retrieval

Device-independent and device-dependent controls are set and retrieved through the video daemon.

#### **Generic Video Tools**

The generic video tools include:

videopanel (vcp) Use this graphical user interface to set controls, such as hue

or contrast, on devices. The panel resizes itself dynamically

to reflect available video devices.

videoin Use the video input window tool to view video in a

window.

videoout Use the video output tool to output video from a

rectangular or full-screen area of the screen on hardware

that supports the screen-to-video path.

vlinfo Use the video info tool to display information about video

devices available through the VL, such as the name of the server, number of devices on the server, and the types and

ID numbers of nodes, sources, and drains on each device.

vidtomem Use this tool to capture a single frame (the current video

input) or a specified number of frames, depending on the hardware limits for burst capture, and write the data to disk. Capture size can also be specified. The data can be translated or left as raw data, which can be used by the

memtovid tool.

memtovid Use this tool to output single frames (images) to video out

on hardware that supports the memory-to-video path.

The *vlinfo*, *vidtomem*, and *memtovid* tools are command-line tools. In addition to their man pages, these tools have explanations in the *Media Control Panels User's Guide*, which you can view using the IRIS InSight viewer; similar applications are supplied in source-code form as examples in the *4Dgifts* directory (/usr/people/4Dgifts/examples/dmedia/video/vl).

**Note:** Additional video tools may be available for specific on-board video or video options; see the documentation for those products.

## **Library and Header Files**

The client library is /usr/lib/libvl.a. The header files for the VL are in /usr/include/dmedia/vl; the main file is vl.h. This file contains the main definition of the VL API and controls that are common across all hardware.

Device-dependent files use the form  $vl_XXX.h$ , where XXX is replaced with the device-dependent name. Table 12-1 lists header files for hardware options that use the VL. These files contain additional controls specific to the devices.

**Table 12-1**Header Files for Video Options

Hardware	Header File
Galileo Video, Indigo <sup>2</sup> Video, Indy Video	vl_ev1.h
Sirius Video	vl_sirius.h
Video capability built into Indy workstation (VINO: video in, no out)	vl_vino.h

## **VL Architectural Model of Video Devices**

The two central concepts for VL are:

- path: an abstraction for a way of moving data around
- *node*: an endpoint of the path, such as a video *source* (such as a VTR), video *drain* (such as the screen), a *device* (such as Indy Video), or the *blender* in which video sources are combined for output to a drain

VL routines explained in this chapter enable you to build a fully connected topology of sources and drains.

A path defines the useful connections between video sources and video drains. Figure 12-2 shows a simple path in which a frame from a videotape is displayed in a workstation window.

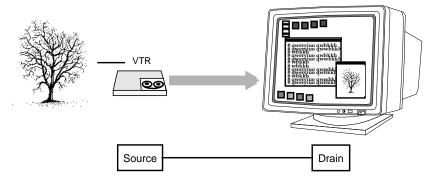


Figure 12-2 Simple VL Path

Some Silicon Graphics platforms are capable of supporting more than one video device; for example, Indy supports VINO and Indy Video. Each video device has its own data paths with sources and drains. The application is responsible for looking at the capabilities of the platform and choosing the video device it will run on.

Figure 12-3 shows a more complex path with two video sources: a frame from a videotape and a computer-generated image are blended and output to a workstation window. This path is set up in stages.

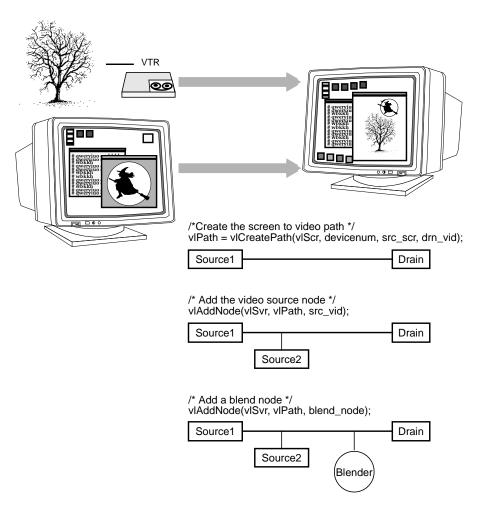


Figure 12-3 Simple VL Blending

# **VL Programming Model**

The VL recognizes five classes of objects:

- devices, each including sets of nodes
- nodes: sources, drains, and internal nodes
- paths, connecting sources and drains
- *controls*, or parameters that modify how data flows through nodes; for example:
  - video device parameters, such as blanking width, gamma value, horizontal phase, sync source
  - video data capture parameters
  - blending parameters
- buffers, for sending and receiving frame data to and from host memory; the VL buffers are implemented as ring buffers containing a number of blocks; each maintains a pointer, a size, and pointers to the head (oldest) and tail (newest) valid data

Data transfers fall into two categories:

- transfers involving memory (video to memory, memory to video), which require setting up a ring buffer
- transfers not involving memory (such as video to screen and graphics to video), which do not require a ring buffer (such transfers are not supported on VINO)

Syntax elements are as follows:

- VL types and constants begin with uppercase VL; for example, VLServer
- VL functions begin with lowercase vl; for example, vlOpenVideo()

For the two categories of data transfer, based on the VL programming model, the process of creating a VL application consists of these steps:

- opening a connection to the video daemon (vlOpenVideo()); if necessary, determining which device the application will use (vlGetDevice(), vlGetDeviceList())
- 2. specifying nodes on the data path (vlGetNode())
- 3. creating the path (vlCreatePath())
- 4. optional step: adding more connections to a path (vlAddNode())
- 5. setting up the hardware for the path (vlSetupPaths())
- 6. specifying path-related events to be captured (vlSelectEvents())
- 7. setting input and output parameters (controls) for the nodes on the path (vlSetControl())
- 8. transfers involving memory: creating a ring buffer to hold data for memory transfers (vlGetTransferSize(), vlCreateBuffer())
- 9. transfers involving memory: registering the buffer (vlRegisterBuffer())
- 10. starting the data transfer (vlBeginTransfer())
- 11. transfers involving memory: getting the data (vlGetNextValid() or vlGetLatestValid(), vlGetActiveRegion(), vlPutFree()) to manipulate frame data
- 12. cleanup (vlEndTransfer(), vlDeregisterBuffer(), vlDestroyPath(), vlDestroyBuffer(), vlCloseVideo())

Table 12-2 lists calls explained in this chapter.

**Table 12-2** Video Library Calls for Data Transfer

All Transfers	Transfers Involving Memory	Setting Controls	
vlOpenVideo()	vlGetTransferSize()	vlSetControl()	
vlGetDevice()	vlCreateBuffer()	vlGetControl()	
vlGetDeviceList()	vlRegisterBuffer()	vlControlList()	
vlGetNode()	vlGetNextValid()	vlGetControlInfo()	
vlCreatePath()	vlGetLatestValid()	V	
vlAddNode()	vlPutValid()		
vlRemoveNode()	vlGetNextFree()		
vlSetupPaths()	vlGetActiveRegion()		
vlSelectEvents()	vlPutFree()		
vlBeginTransfer()	vlGetDMediaInfo()		
vlEndTransfer()	vlGetImageInfo()		
vlDestroyPath()	vlDeregisterBuffer()		
vlCloseVideo()	vlDestroyBuffer()		

# Opening a Connection to the Video Daemon and Setting up a Data Path

Preliminary procedures required to create the data path are:

- opening the device
- specifying nodes on the data path
- creating and setting up the data path

Each procedure is explained separately.

## Opening a Connection to the Video Daemon

The first thing a VL application must do is open the device with **vlOpenVideo()**. Its function prototype is:

VLServer vlOpenVideo(const char \*sName)

where *sName* is the name of the server to which to connect; set it to a NULL string for the local server. For example:

```
svr = vlOpenVideo("")
```

## Specifying Nodes on the Data Path

Use **vlGetNode()** to specify nodes; this call returns the node's handle. Its function prototype is:

VLNode vlGetNode(VLServer vlServer, int type, int kind, int number)

where:

**VLNode** 

is a handle for the node, used when setting controls or

setting up paths

vlServer

names the server (as returned by vlOpenVideo())

type

specifies the type of node:

- VL\_SRC: source
- VL\_DRN: drain
- VL\_DEVICE: device for device-global controls

**Note:** If you are using VL\_DEVICE, the kind should be set to 0.

VL\_INTERNAL: internal node, such as the blend node

kind

specifies the kind of node:

- VL\_VIDEO: connection to a video device; for example, a video tape deck or camera
- VL\_GFX: graphics system (Sirius Video only)
- VL\_MEM: region of workstation memory
- VL\_SCREEN: workstation screen (Galileo Video, Indigo<sup>2</sup> Video, and Indy Video only)
- VL\_TEXTURE: texture RAM (Sirius Video only)
- VL\_BLENDER: a blender node

**Note:** The use of VL\_BLENDER is explained in Chapter 15, "VL Blending," later in this guide.

• VL\_ANY: use any available node

number

is the number of the node in cases of two or more identical nodes, such as two video source nodes

To use the default node kind, use VL\_ANY.

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

To discover which node the default is, use the control VL\_DEFAULT\_SOURCE after getting the node handle the normal way. The default video source is maintained by the VL. For example:

```
vlGetControl(svr, path, VL_ANY, VL_DEFAULT_SOURCE, &ctrlval);
nodehandle = vlGetNode(svr, VL_SRC, VL_VIDEO,
ctrlval.intVal);
```

In the second line above, the last argument is a struct that gets the value.

**Note:** If either VINO analog channel is active when the first video application starts, the default is analog; otherwise, the default is digital.

## Creating and Setting Up the Data Path

Once nodes are specified, use VL calls to:

- determine the device ID (optional step)
- create the path
- get the device ID
- add nodes (optional step)
- set up the data path
- specify the path-related events to be captured

#### **Determining the Device ID**

In this optional step, use one of the following calls to determine the device on which the data path will be created, depending on the situation.

 If you do not know which device of several available is appropriate for the data path, get the device list with vlGetDeviceList(). Its function prototype is:

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

• If you know the device you want, parse *devlist* to get its handle. Otherwise, the VL selects the first device that the path you have specified can run on.

Use this step for systems with multiple devices of different capabilities; for example, on an Indy workstation with VINO and Indy Video, for full frame rate capture, specify VINO, though both devices support the video input path.

The struct for *nodeinfo* in *vl.h* is:

```
typedef struct __vlNodeInfo {
char name[VL_NAME_SIZE]; /* name of node */
int type; /* see list above */
int number; /* number of this node */
int kind; /* see list above */
} VLNodeInfo;
The struct for dev in vl.h is:
typedef struct __vlDevice {
VLDev dev;
char name[VL_NAME_SIZE]; /* name of device */
uint numNodes; /* number of nodes on this device */
VLNodeInfo *nodes; /* list of nodes */
} VLDevice;
The struct for devlist in vl.h is:
typedef struct __vlDevList {
                              /* number of devices */
   uint numDevices;
                              /* list of devices */
   VLDevice *devices;
} VLDevList;
```

## Creating the Path

Use vlCreatePath() to create the data path. Its function prototype is:

```
\begin{tabular}{ll} VLPath vlCreatePath(VLServer vlServer, VLDev vlDev \\ VLNode src, VLNode drn) \end{tabular}
```

This code fragment creates a path if the device is unknown:

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

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

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

**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 vlServer, VLPath path)
```

## For example:

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

#### Adding a Node

For this optional step, use **vlAddNode()**. Its function prototype is:

```
int vlAddNode(VLServer vlServer, VLPath path, VLNodeId node)
```

#### where:

vlServer names the server to which the path is connected

vlPath is the path as defined with vlCreatePath()

node is the node ID

This example fragment adds a source node and a blend node:

```
vlAddNode(vlSvr, vlPath, src_vid);
vlAddNode(vlSvr, vlPath, blend_node);
```

#### **Setting Up the Data Path**

Use vlSetupPaths() to set up the data path. Its function prototype is:

#### where:

vlServernames the server to which the path is connectedpathsspecifies a list of paths you are setting upcountspecifies the number of paths in the path listctrlusagespecifies usage for path controls:

 VL\_SHARE: 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: controls cannot be set, but can only be read; for example, this control can be used to monitor controls
- VL\_LOCK: prevents other paths from setting controls on this path; controls cannot be used by another path
- VL\_DONE\_USING: the resources are no longer required; the application releases this set of paths for other applications to acquire

#### streamusage

specifies usage for the data:

• VL\_SHARE: 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 device, a VLStreamPreempted event occurs.

- VL\_READ\_ONLY: the path cannot perform transfers, but other resources are not locked; set this value to use the path for controls
- VL\_LOCK: prevents other paths that share data transfer resources with this path from transferring; existing paths that share resources with this path will be preempted
- VL\_DONE\_USING: the resources are no longer required; the application releases this set of paths for other applications to acquire

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

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

#### Specifying the Path-related Events to Be Captured

Use **vlSelectEvents()** to specify the events you want to receive. Its function prototype is:

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

#### where:

vlServer names the server to which the path is connected.

path specifies the data path.

eventmask specifies the event mask; Table 12-3 lists the possibilities.

Table 12-3 lists and describes the VL event masks.

Table 12-3VL Event Masks

Symbol	Meaning	
VLStreamBusyMask	Stream is locked	
VLStreamPreemptedMask	Stream was grabbed by another application	
VLAdvanceMissedMask	Time was already reached	
VLSyncLostMask	Irregular or interrupted signal	
VLSequenceLostMask	Field or frame dropped	
VLControlChangedMask	A control has changed	
VL Control Range Changed Mask	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 on a Galileo Video device	
VLDefaultSourceMask	Default source changed	

## For example:

vlSelectEvents(svr, path, VLTransferCompleteMask);

## Event masks can be ORed together. For example:

vlSelectEvents(svr, path, VLTransferCompleteMask |
VLTransferFailedMask);

# **Setting Parameters for Data Transfer to or from Memory**

Transferring data to or from memory requires creating a ring buffer; its size is determined by the size of the frame data you are transferring.

To set frame data size and to convert from one video format to another, apply controls to the nodes. The use of source node and drain node controls is explained separately in this section.

**Note:** All controls are available for all platforms unless otherwise noted. The reference "Galileo Video" includes Indigo<sup>2</sup> Video and Indy Video, unless otherwise noted.

# **Setting Source Node Controls for Data Transfer**

Important data transfer controls for source nodes are summarized in Table 12-4. They should be set in the order in which they appear in the table.

 Table 12-4
 Data Transfer Controls for Source Nodes

Control	Values	Basic Usage
VL_MUXSWITCH	See Table 12-5	Determines physical input for path
VL_TIMING	Default: timing produce by active signal VL_TIMING_525_SQ_PIX VL_TIMING_625_SQ_PIX	Set or get video timing For Betacam, MII, composite tape formats: Analog: 12.27 MHz, 646 x 486 Analog: 14.75 MHz, 768 x 576
	VL_TIMING_525_CCIR601 VL_TIMING_625_CCIR601 VL_TIMING_525_4FSC VL_TIMING_625_4FSC	For D1 tape formats: Digital component: 13.50 MHz, 720 x 486 Digital component: 13.50 MHz, 720 x 576
	VE_THVIHVG_020_H oc	For D2 tape formats: 4X NTSC subcarrier: 14.32 MHz, 768 x 486 4X PAL subcarrier: 17.72 MHz, 948 x 576
VL_SIZE	Coordinates	Set or get active unmodified video area
VL_SYNC_SOURCE	Galileo Video: Composite 1: set 0 Composite 2: set 1 Composite 3: set 2	Not applicable to VINO
	Indigo <sup>2</sup> Video and Indy Video: Composite 1: set 0 Composite 2: set 2	

The use of VL\_MUXSWITCH and VL\_TIMING is explained in further detail in the following sections.

## Using VL\_MUXSWITCH

Use VL\_MUXSWITCH to switch between physical inputs on a single path. Table 12-5 summarizes values for VL\_MUXSWITCH, which vary, depending on the platform.

Table 12-5VL\_MUXSWITCH Values

Platform	Values
Galileo Video	S-Video input 1: set 0; input 2: set 1; input 3: set 2 Composite input 1: set 3; input 2: set 4; input 3: set 5 Y/R-Y/B-Y input 1: set 6; input 2: set 7
Indigo <sup>2</sup> Video and Indy Video	Y/C (RCA jacks): set 0 Y/C (S-Video connector): set 1 Composite input 1: set 3; input 2: set 5
VINO	Node VL_VINO_SRC_DV_IN: VL_VINO_INDYCAM VL_VINO_CCIR601
	Node VL_VINO_SRC_AV_IN: VL_VINO_COMPOSITE VL_VINO_SVIDEO

For Indy Video, the default source depends on which input is active; that is, which input has equipment that is both plugged in and powered on. In other words, the VL assumes that you want to use the piece of equipment that is plugged in and powered on, without you having to tell it so. If the S-Video input is active, it is the default. If the composite input is active, it is the default. If both S-Video and composite equipment are inactive and the IndyCam is active (plugged in), the IndyCam is the default. Composite becomes the default video in two cases: if it is active or if all other inputs are inactive.

You can control the default by unplugging or plugging in equipment and/or turning equipment power on or off. For example:

- set S-Video active by switching on the equipment plugged into the S-Video input
- set composite active by switching off the equipment plugged into the S-Video input

Of course, you can change the settings—this just gives you an idea of what default to expect. If equipment is plugged in to all the inputs, the default VINO input is established by the precedence listed in top-down order in Table 12-6.

**Table 12-6** Default Sources for VINO Inputs

Input	S-Video	Composite	IndyCam
S-Video	Active	N/A	N/A
Composite	Inactive	Active	N/A
IndyCam	Inactive	Inactive	Active
Composite	Inactive	Inactive	Inactive

# Using VL\_TIMING

Table 12-7 summarizes VL\_TIMING choices for combinations of nodes and mux switches for VINO.

Table 12-7VINO Timing Choices

Node Value	VL_MUXSWITCH Value	Timing Choices
VL_VINO_SRC_DV_IN	VL_VINO_INDYCAM	VL_TIMING_525_SQ_PIX (NTSC)
	VL_VINO_CCIR601	VL_TIMING_525_SQ_PIX (NTSC) VL_TIMING_625_SQ_PIX (PAL) VL_TIMING_525_CCIR601(digital NTSC) VL_TIMING_625_CCIR601 (digital PAL)
VL_VINO_SRC_AV_IN	VL_VINO_COMPOSITE	VL_TIMING_525_SQ_PIX (NTSC) VL_TIMING_625_SQ_PIX (PAL)
	VL_VINO_SVIDEO	VL_TIMING_525_SQ_PIX (NTSC) VL_TIMING_625_SQ_PIX (PAL)

Timing type expresses the timing of video presented to a source or drain. Table 12-8 summarizes dimensions for VL\_TIMING.

 Table 12-8
 Dimensions for Timing Choices

Timing	Maximum Width	Maximum Height	First Active Line (Offset=0)
VL_TIMING_525_SQ_PIX (12.27 MHz)	640	480	22
VL_TIMING_625_SQ_PIX (14.75 MHz)	768	576	32
VL_TIMING_525_CCIR601 (13.50 MHz)	720	480	1
VL_TIMING_625_CCIR601(13.50 MHz)	720	576	1
VL_TIMING_525_SQ_PIX (12.27 MHz) (with input VL_VINO_INDYCAM)	640	480	2

# **Setting Drain Node Controls for Data Transfer**

Important data transfer controls for drain nodes are summarized in Table 12-9. They should be set in the order in which they appear in the table.

 Table 12-9
 Data Transfer Controls for Drain Nodes

Control	Basic Usage	Video Nodes	Memory Nodes	Screen Nodes
VL_FORMAT	Video format on the physical connector	See "Using VL_FORMAT"		
VL_TIMING	Video timing	See Table 12-4 for values	Not applicable	Not applicable
VL_CAP_TYPE	Setting type of field(s) or frame(s) to capture; see "Interlacing" in Chapter 11	Not applicable	VL_CAPTURE_NONINTERLEAVED VL_CAPTURE_INTERLEAVED VL_CAPTURE_EVEN_FIELDS VL_CAPTURE_ODD_FIELDS	Not applicable
VL_PACKING	Pixel packing (conversion) format	Not applicable	Changes pixel format of captured data; see Table 12-10 for values	Not applicable
VL_ZOOM	Decimation or zoom factor (fraction)  Galileo Video: 1/1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8, 2/1, 4/1  VINO: 1/1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8	Not applicable	Decimation or zoom: resizes data to remain within limits	Decimation or zoom: resizes data to remain within limits
VL_SIZE	Clipping size	Full size of video; read only	Clipped size	Clipped size
VL_OFFSET	Position within larger area	Position of active region	Offset relative to video offset	Pan within the video
VL_ORIGIN	Position within video	Not applicable	Not applicable	Screen position of first pixel displayed; not applicable to VINO

Table 12-9 (continued)

Data Transfer Controls for Drain Nodes

Control	Basic Usage	Video Nodes	Memory Nodes	Screen Nodes
VL_WINDOW	Setting window ID for video in a window	Not applicable	Not applicable	Window ID; not applicable to VINO
VL_RATE	Field or frame transfer speed	Depends on capture type as specified by VL_CAP_TYPE	Not applicable	Not applicable

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. For all devices, VL\_PACKING must be set first. For VINO, set offset before size. Note that changes in one parameter may change the values of other parameters set earlier; for example, clipped size may change if VL\_PACKING is set after VL\_SIZE.

To determine default values, use **vlGetControl()** to query the values on the video source or drain node before setting controls. The initial offset of the video node is the first active line of video.

Similarly, the initial size value on the video source or drain node is the full size of active video being captured by the hardware, beginning at the default offset. Because some hardware can capture more than the size given by the video node, this value should be treated as a default size.

For all these controls, it pays to track return codes. If the value returned is VLValueOutOfRange, the value set will not be what you requested.

To specify the controls, use **vlSetControl()**, for which the function prototype is:

The use of VL\_FORMAT, VL\_PACKING, VL\_ZOOM, VL\_SIZE, VL\_OFFSET, VL\_RATE, and VL\_CAP\_TYPE is explained in more detail in the following sections.

#### Using VL\_FORMAT

To specify video input and output formats of the video signal on the physical connector, use VL\_FORMAT. Each video platform has a video format native to it; for example, YUV 4:2:2 is native to Galileo Video and RGB is native to Sirius Video. The native format is always the fastest for that platform. To discover the native format for your video platform, consult the release notes or other documentation for the product.

**Note:** To convert formats, use VL\_PACKING, which is explained in the next section.

When VL\_FORMAT is applied to a source or drain that is a VL\_MEM (memory) node, it selects the format of the video stored in memory. This may imply a software conversion of the video data after the transfer is completed.

Values for VL\_FORMAT for Galileo Video are:

- VL\_FORMAT\_RGB (output only)
- VL\_FORMAT\_BETACAM (input and output)
- VL\_FORMAT\_SMPTE\_YUV (input and output)

#### Using VL\_PACKING

To convert a video output format to another in memory, use the control VL\_PACKING. Packing type expresses the packing in memory of the video data at the source or drain.

Packing types are summarized in Table 12-10, which shows the most significant byte on the left. An x means don't care; this bit is not used.

**Table 12-10** Packing Types and Their Sizes and Formats

Туре	Size	Format MSBLSB
VL_PACKING_RGB_332_P	8-bit word	BBGGGRRR (four pixels packed into a 32-bit word)
VL_PACKING_RGBA_8	32-bit word	AAAAAAA BBBBBBBB GGGGGGG RRRRRRR
VL_PACKING_RGB_8	24-bit word	XXXXXXXX BBBBBBBB GGGGGGG RRRRRRR
VL_PACKING_Y_8_P	8-bit word	YYYYYYYY (four pixels packed into a 32-bit word)
VL_PACKING_YVYU_422_8	32-bit word	UUUUUUUU YYYYYYY VVVVVVV YYYYYYYY

**Note:** The packing names follow the naming conventions used by the IRIS GL; other libraries such as the OpenGL may use different names.

#### For example:

```
VLControlValue val;
val.intVal = VL_PACKING_RGB;
vlSetControl(svr, path, memdrn, VL_PACKING, &val);
```

## Using VL\_ZOOM

VL\_ZOOM controls the expansion or decimation of the video image. Values greater than one expand the video; values less than one perform decimation. Figure 12-4 illustrates zooming and decimation.

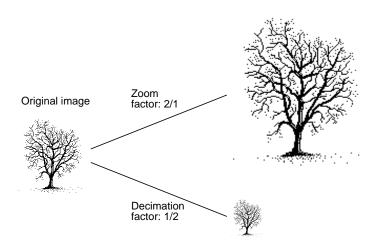


Figure 12-4 Zoom and Decimation

 $\label{eq:Note:optimization} \textbf{Note:} \ \ Zooming, that is, VL\_ZOOM \ values \ greater \ than \ one, is \ not \ supported \ on \ VINO.$ 

VL\_ZOOM takes a nonzero fraction as its argument; do not use negative values. For example, this fragment captures half-size decimation video to memory:

```
val.fractVal.numerator = 1;
val.fractVal.denominator = 2;
if (vlSetControl(server, memory_path, memory_drain_node,
VL_ZOOM, &val)){
  vlPerror("Unable to set zoom");
  exit(1);
}
```

**Caution:** Not all video devices support all aspects of zooming. If you use a control on a video device that does not support it, a VLValueOutOfRange error is returned. Use **vlGetControl()** to show what your results were.

**Note:** For a source, zooming takes place before blending; for a drain, blending takes place before zooming.

This fragment captures half-size decimation video to memory, with clipping to  $320 \times 240$  (NTSC size minus overscan).

This fragment captures  $xsize \times ysize$  video with as much decimation as possible, assuming the size is smaller than the video stream.

```
if (vlGetControl(server, memory_path, video_source, VL_SIZE,
&val))
 vlPerror("Unable to get size");
 exit(1);
if (val.xyVal.x/xsize < val.xyVal.y/ysize)</pre>
 zoom_denom = (val.xyVal.x + xsize - 1)/xsize;
else
 zoom_denom = (val.xyVal.y + ysize - 1)/ysize;
val.fractVal.numerator = 1;
val.fractVal.denominator = zoom_denom;
if (vlSetControl(server, memory_path, memory_drain_node,
VL_ZOOM,
   &val))
 /* allow this error to fall through */
 vlPerror("Unable to set zoom");
}
```

```
val.xyVal.x = xsize;
val.xyVal.y = ysize;
if (vlSetControl(server, memory_path, memory_drain_node,
VL_SIZE,
    &val))
{
  vlPerror("Unable to set size");
  exit(1);
}
```

#### Using VL\_SIZE

VL\_SIZE controls how much of the image sent to the drain is used, that is, how much clipping takes place. This control operates on the zoomed image; for example, when the image is zoomed to half size, the limits on the size control change by a factor of 2. Figure 12-5 illustrates clipping.

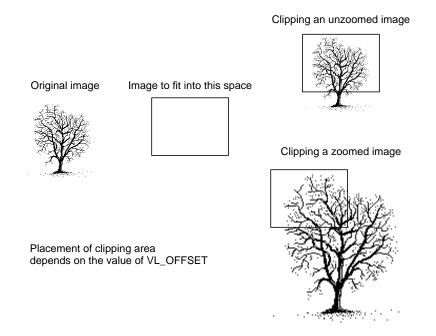


Figure 12-5 Clipping an Image

For example, to display PAL video in a 320 × 240 space, clip the image to that size, as shown in the following fragment:

```
VLControlValue value;
value.xyval.x=320;
value.xyval.y=240;
vlSetControl(svr, path, drn, VL_SIZE, &value);
```

**Note:** Because this control is device-dependent and interacts with other controls, always check the error returns. For example, if offset is set before size and an error is returned, set size before offset.

#### Using VL\_OFFSET

VL\_OFFSET puts the upper left corner of the video data at a specific position; it sets the beginning position for the clipping performed by VL\_SIZE. The values you enter are relative to the origin.

VL\_OFFSET operates on the unzoomed image; it does not change if the zoom factor is changed.

This example places the data ten pixels down and ten pixels in from the left:

```
VLControlValue value;
value.xyval.x=10;
value.xyval.y=10;
vlSetControl(svr, path, drn, VL_OFFSET, &value);
```

To capture the blanking region, set offset to a negative value.

# Using VL\_RATE and VL\_CAP\_TYPE

VL\_RATE determines the data transfer rate by field or frame, depending on the capture type as specified by VL\_CAP\_TYPE, as shown in Table 12-11.

**Table 12-11** VL\_RATE Values (Items per Second)

VL_CAP_TYPE Value	VL_RATE Value
VL_CAPTURE_NONINTERLEAVED only	NTSC: 10, 12, 20, 24, 30, 36, 40, 48, 50, 60 PAL: 5, 10, 15, 20, 25
VL_CAPTURE_INTERLEAVED, VL_CAPTURE_EVEN_FIELDS, and VL_CAPTURE_ODD_FIELDS	NTSC: 5, 6, 10, 12, 15, 18, 20, 24, 25, 30 PAL: 10, 20, 30, 40, 50

Figure 12-6 shows the relationships between the source and drain zoom, size, offset, and origin.

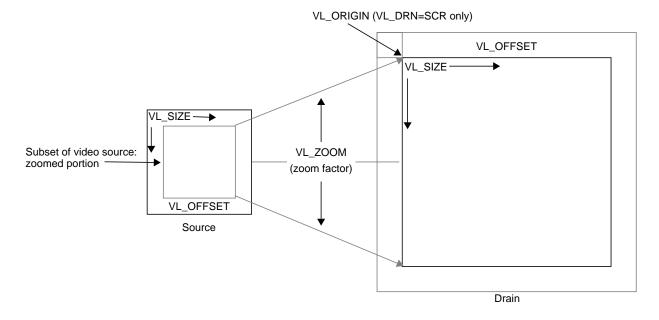


Figure 12-6 Zoom, Size, Offset, and Origin

## **Displaying Video Data Onscreen**

To set up a window for live video on Galileo Video, Indigo<sup>2</sup> Video, or Indy Video, follow these steps, as outlined in the example program *simplev2s.c.* 

**Note:** This information does not apply to VINO. Because the video resident in the Indy workstation has no screen node, use the memory node. Capture the video and use the **lrectwrite()** function or the analogous X or OpenGL function.

1. Open an X display window; for example:

```
if (!(dpy = XOpenDisplay("")))
    exit(1);
```

2. Connect to the video daemon; for example:

```
if (!(svr = vlOpenVideo("")))
    exit(1);
```

3. Create a window to show the video; for example:

4. Create a source node on a video device and a drain node on the screen; for example:

```
src = vlGetNode(svr, VL_SRC, VL_VIDEO, VL_ANY);
drn = vlGetNode(svr, VL_DRN, VL_SCREEN, VL_ANY);
```

5. Create a path on the first device that supports it; for example:

```
if((path = vlCreatePath(svr, VL_ANY, src, drn)) < 0)
    exit(1);</pre>
```

6. Set up the hardware for the path and define the path usage; for example:

7. Set the X window to be the drain; for example:

```
val.intVal = vwin;
vlSetControl(svr, path, drn, VL_WINDOW, &val);
```

8. Get X and VL into the same coordinate system; for example:

```
XTranslateCoordinates(dpy, vwin, RootWindow(dpy,
DefaultScreen(dpy)), 0, 0,&x, &y, &dummyWin);
```

9. Set the live video to the same location and size as the window; for example:

```
val.xyVal.x = x;
val.xyVal.y = y;
vlSetControl(svr, path, drn, VL_ORIGIN, &val);
XGetGeometry(dpy, vwin, &dummyWin, &x, &y, &w, &h, &bw, &d);
val.xyVal.x = w;
val.xyVal.y = h;
vlSetControl(svr, path, drn, VL_SIZE, &val);
```

10. Begin the data transfer:

```
vlBeginTransfer(svr, path, 0, NULL);
```

11. Wait until the user finishes; for example:

```
printf("Press return to exit.\n");
c = getc(stdin);
```

12. End the data transfer, clean up, and exit:

```
vlEndTransfer(svr, path);
vlDestroyPath(svr, path);
vlCloseVideo(svr);
```

# Transferring Video Data to and from Devices

The processes for data transfer are:

- creating a buffer for the frames (transfers involving memory)
- registering the ring buffer with the path (transfers involving memory)
- starting data transfer
- reading data from the buffer (transfers involving memory)

Each process is explained separately.

## Creating a Buffer for the Frames

Once you have specified frame parameters in a transfer involving memory (or have determined to use the defaults), create a buffer for the frames.

Like other libraries in the IRIS digital media development environment, the VL uses ring buffers. Ring buffers provide a way to read and write varying sizes of frames of data. A frame of data consists of the actual frame data and an information structure describing the underlying data, including device-specific information.

When a ring buffer is created, constraints are specified that control the total size of the data segment and the number of information buffers to allocate.

A head and a tail flag are automatically set in a ring buffer so that the latest frame can be accessed. A sector is locked down if it is not called; that is, it remains locked until it is read. When the ring buffer is written to and all sectors are occupied, data transfer stops. The sector last written to remains locked down until it is released.

The ring buffer can accommodate data of varying size. You can specify a ring buffer at a fixed size or can determine the size of the data in the buffer.

To determine frame data size, use **vlGetTransferSize()**. Its function prototype is:

```
long vlGetTransferSize(VLServer svr, VLPath path)
```

#### For example:

```
transfersize = vlGetTransferSize(svr, path);
```

where *transfersize* is the size of the data in bytes.

To create a ring buffer for the frame data, use **vlCreateBuffer()**. Its function prototype is:

```
VLBuffer vlCreateBuffer(VLServer vlServer, VLPath path, VLNode node, int numFrames)
```

where:

VLBuffer is the handle of the buffer to be created

vlServer names the server to which the path is connected

path specifies the data path

node specifies the memory node containing data to transfer to or

from the ring buffer

numFrames specifies the number of frames in the buffer

#### For example:

```
buf = vlCreateBuffer(svr, path, src, 1);
```

# Registering the Ring Buffer

Use **vlRegisterBuffer()** to register the ring buffer with the data path. Its function prototype is:

```
int vlRegisterBuffer(VLServer vlServer, VLPath path,  {\tt VLNode\ memnodeid},\ {\tt VLBuffer\ buffer}) \\
```

#### where:

vlServer names the server to which the path is connected

path specifies the data path

memnodeid specifies the memory node IDbuffer specifies the ring buffer handle

#### For example:

```
vlRegisterBuffer(svr, path, drn, Buffer);
```

#### Starting Data Transfer

To begin data transfer, use vlBeginTransfer(). Its function prototype is:

#### where:

vlServer names the server to which the path is connected

path specifies the data path

count specifies the number of transfer descriptors

Tailor the data transfer by means of *transfer descriptors*. This example fragment transfers the entire contents of the buffer immediately.

```
xferDesc.mode = VL_TRANSFER_MODE_DISCRETE;
xferDesc.count = imageCount;
xferDesc.delay = 0;
xferDesc.trigger = VLTriggerImmediate;
```

#### The transfer descriptors are:

*xferDesc.mode* Transfer method:

- VL\_TRANSFER\_MODE\_DISCRETE: a specified number of frames are transferred (burst mode)
- VL\_TRANSFER\_MODE\_CONTINUOUS (default): frames are transferred continuously, beginning immediately or after a trigger event occurs (such as a frame coincidence pulse), and continues until transfer is terminated with vlEndTransfer()
- VL\_TRANSFER\_MODE\_AUTOTRIGGER: frame transfer takes place each time a trigger event occurs; this mode is a repeating version of VL\_TRANSFER\_MODE\_DISCRETE

xferDesc.count Number of frames to transfer; if mode is

VL\_TRANSFER\_MODE\_CONTINUOUS, this value is

ignored

*xferDesc.delay* Number of frames from the trigger at which data transfer

begins

*xferDesc.trigger* Set of events to trigger on; an event mask. This transfer

descriptor is always required. VLTriggerImmediate specifies that transfer begins immediately, with no pause for

a trigger event

This fragment shows the default descriptor, which is the same as passing in a null for the descriptor pointer. Transfer begins immediately; *count* is ignored.

```
xferDesc.mode = VL_TRANSFER_MODE_CONTINUOUS;
xferDesc.count = 0;
xferDesc.delay = 0;
xferDesc.trigger = VLTriggerImmediate;
```

# Reading Data from the Buffer

If your application uses a buffer, use various VL calls for reading frames, getting pointers to active buffers, freeing buffers, and other operations. Table 12-12 lists the buffer-related calls.

 Table 12-12
 Buffer-Related Calls

Call	Purpose
vlGetNextValid()	Returns a handle on the next valid frame of data
vlGetLatestValid()	Reads only the most current frame in the buffer, discarding the rest $% \left( 1\right) =\left( 1\right) \left( 1\right$
vlPutValid()	Puts a frame into the valid list (memory to video)
vlPutFree()	Puts a valid frame back into the free list (video to memory)
vlGetNextFree()	Gets a free buffer into which to write data (memory to video)
vlBufferDone()	Informs you if the buffer has been vacated
vlBufferReset()	Resets the buffer so that it can be used again

Figure 12-7 illustrates the difference between **vlGetNextValid()** and **vlGetLatestValid()**, and their interaction with **vlPutFree()**.

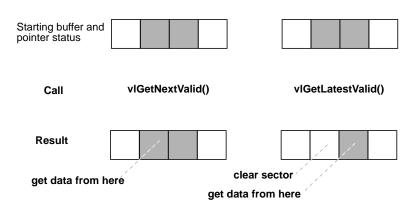


Figure 12-7 vlGetNextValid(), vlGetLatestValid(), and vlPutFree()

Table 12-13 lists the calls that extract information from a buffer.

 Table 12-13
 Calls for Extracting Data from a Buffer

Call	Purpose
vlGetActiveRegion()	Gets a pointer to the data region of the buffer (video to memory); called after vlGetNextValid() and vlGetLatestValid()
vlGetDMediaInfo()	Gets a pointer to the DMediaInfo structure associated with a frame; this structure contains timestamp and field count information
vlGetImageInfo()	Gets a pointer to the DMImageInfo structure associated with a frame; this structure contains image size information

**Caution:** None of these calls has count or block arguments; appropriate calls in the application must deal with a NULL return in cases of no data being returned.

#### In summary, for video-to-memory transfer use:

```
buffer = vlCreateBuffer(svr, path, memnodel);
vlRegisterBuffer(svr, path, memnodel, buffer);
vlBeginTransfer(svr, path, 0, NULL);
info = vlGetNextValid(svr, buffer);
/* OR vlGetLatestValid(svr, buffer); */
dataptr = vlGetActiveRegion(svr, buffer, info);
/* use data for application */
...
vlPutFree(svr, buffer);

For memory-to-video transfer, use:
buffer = vlCreateBuffer(svr, path, memnodel);
vlRegisterBuffer(svr, path, memnodel, buffer);
vlBeginTransfer(svr, path, 0, NULL);
buffer = vlGetNextFree(svr, buffer, bufsize);
/* fill buffer with data */
...
vlPutValid(svr, buffer);
```

These calls are explained in separate sections.

#### Reading the Frames to Memory from the Buffer

Use **vlGetNextValid()** to read all the frames in the buffer or get a valid frame of data. Its function prototype is:

```
VLInfoPtr vlGetNextValid(VLServer vlServer, VLBuffer
vlBuffer)
```

Use **vlGetLatestValid()** to read only the most current frame in the buffer, discarding the rest. Its function prototype is:

```
VLInfoPtr vlGetLatestValid(VLServer vlServer, VLBuffer vlBuffer)
```

After removing interesting data, return the buffer for use with **vlPutFree()** (video to memory). Its function prototype is:

```
int vlPutFree(VLServer vlServer, VLBuffer vlBuffer)
```

#### Sending Frames from Memory to Video

Use **vlGetNextFree()** to get a free buffer to which to write data. Its function prototype is:

```
VLInfoPtr vlGetNextFree(VLServer vlServer, VLBuffer vlBuffer, int size)
```

After filling the buffer with the data you want to send to video output, use **vlPutValid()** to put a frame into the valid list for output to video (memory to video). Its function prototype is:

```
int vlPutValid(VLServer vlServer, VLBuffer vlBuffer)
```

**Caution:** These calls do not have count or block arguments; appropriate calls in the application must deal with a NULL return in cases of no data being returned.

#### **Getting DMediaInfo and Image Data from the Buffer**

Use **vlGetActiveRegion()** to get a pointer to the active buffer. Its function prototype is:

Use **vlGetDMediaInfo()** to get a pointer to the DMediaInfo structure associated with a frame. This structure contains timestamp and field count information. The function prototype for this call is:

Use **vlGetImageInfo()** to get a pointer to the DMImageInfo structure associated with a frame. This structure contains image size information. The function prototype for this call is:

## **Ending Data Transfer**

To end data transfer, use vlEndTransfer(). Its function prototype is:

```
int vlEndTransfer(VLServer vlServer, VLPath path)
```

To accomplish the necessary cleanup to exit gracefully, use:

- for transfer involving memory: vlDeregisterBuffer(), vlDestroyPath(), vlDestroyBuffer()
- for all transfers: vlCloseVideo()

#### The function prototype for **vlDeregisterBuffer()** is:

#### where:

vlServeris the server handlepathis the path handlememnodeidis the memory node IDringbufhandleis the ring buffer handle

# The function prototypes for vlDestroyPath(), vlDestroyBuffer() and vlCloseVideo() are, respectively:

```
int vlDestroyPath(VLServer vlServer, VLPath path)
int vlDestroyBuffer(VLServer vlServer, VLBuffer vlBuffer)
int vlCloseVideo(VLServer vlServer)
```

#### This example ends a data transfer that used a buffer:

```
vlEndTransfer(svr, path);
vlDeregisterBuffer(svr, path, memnodeid, buffer);
vlDestroyPath(svr, path);
vlDestroyBuffer(svr, buffer);
vlCloseVideo(svr);
```

# **VL Examples**

The example code in this section illustrates:

- a simple screen application
- a video-to-memory frame grab
- a memory-to-video frame output
- a continuous frame capture

Source code for these programs is in /usr/people/4Dgifts/examples/dmedia/video/vl.

**Note:** To simplify the code, these examples do not check returns. The programmer should, however, always check returns.

# **Simple Screen Application**

Example 12-1 shows how to send live video to the screen (for systems that have a video output port).

**Example 12-1** Sending Live Video to the Screen: *simplev2s.c* 

```
* File:
                  simplev2s.c
 * Usage:
                  simplev2s
  Description:
                  Simplev2s demonstrates live video to screen.
                  This application only runs on video hardware
                  that has a video output port. It will not run
                  on a VINO video board.
 * Functions:
                  SGI Video Library functions used
                  vlOpenVideo()
                  vlGetNode()
                  vlCreatePath()
                  vlSetupPaths()
                  vlSetControl()
                  vlBeginTransfer()
                  vlEndTransfer()
                  vlDestroyPath()
                  vlCloseVideo()
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <X11/X.h>
#include <X11/Xlib.h>
#include <Xm/MwmUtil.h>
#include <X11/Xutil.h>
#include <vl/vl.h>
```

```
main(int argc, char **argv)
    VLServer svr;
    VLPath path;
    Display *dpy;
    Window vwin;
    VLNode src, drn;
    VLControlValue val;
    char *progname, *ptr;
    int x, y, c;
    uint w, h, bw, d;
    Window dummyWin;
    XSizeHints size_hints;
    XClassHint class_hints;
    /* get basename of argv */
    if ((ptr = strrchr(*argv, '/')) != NULL) progname = ++ptr;
    else progname = *argv;
    /* Open an X display */
    if (!(dpy = XOpenDisplay("")))
        exit(1);
    /* Connect to the video daemon */
    if (!(svr = vlOpenVideo("")))
        exit(1);
    /* Create a window to show the video */
    vwin = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy),
                               10, 10, 640, 480, 0,
                               BlackPixel(dpy, DefaultScreen(dpy)),
                               BlackPixel(dpy, DefaultScreen(dpy)));
    /* Ignore window manager placement set the window to 10, 10 */
    size_hints.flags = USPosition;
    size hints.x = 10;
    size_hints.y = 10;
    /* set class properties for 4Dwm desktop */
    class_hints.res_name = progname;
    class_hints.res_class = progname;
    XSetClassHint(dpy, vwin, &class_hints);
    XSetWMNormalHints(dpy, vwin, &size_hints);
    XMapWindow(dpy, vwin);
    XFlush(dpy);
```

```
/* Create a source node on a video device */
src = vlGetNode(svr, VL_SRC, VL_VIDEO, VL_ANY);
/* Create a drain node on the screen */
drn = vlGetNode(svr, VL_DRN, VL_SCREEN, VL_ANY);
/* Create a path on the first device that supports it */
if((path = vlCreatePath(svr, VL_ANY, src, drn)) < 0)</pre>
    exit(1);
 /* Set up the hardware for and define the usage of the path */
vlSetupPaths(svr, (VLPathList)&path, 1, VL_SHARE, VL_SHARE);
/* Set the X window to be the drain */
val.intVal = vwin;
vlSetControl(svr, path, drn, VL_WINDOW, &val);
/* Get X and VL into the same coordinate system */
XTranslateCoordinates(dpy, vwin, DefaultRootWindow(dpy),
                      0, 0,&x, &y, &dummyWin);
/* Set the live video to the same location and size as the X window */
val.xyVal.x = x;
val.xyVal.y = y;
vlSetControl(svr, path, drn, VL_ORIGIN, &val);
XGetGeometry(dpy, vwin, &dummyWin, &x, &y, &w, &h, &bw, &d);
val.xyVal.x = w;
val.xyVal.y = h;
vlSetControl(svr, path, drn, VL_SIZE, &val);
/* Begin the data transfer */
vlBeginTransfer(svr, path, 0, NULL);
/* Wait until the user presses a key */
printf("Press return to exit.\n");
c = getc(stdin);
/* End the data transfer */
vlEndTransfer(svr, path);
/* Clean up and exit */
vlDestroyPath(svr, path);
vlCloseVideo(svr);
```

}

## **Video-to-memory Frame Grab**

Example 12-2 demonstrates video frame grabbing.

**Example 12-2** Video Frame Grabbing: *simplegrab.c* 

```
* File:
                  simplegrab.c
 * Usage:
                  simplegrab
 * Description:
                  simplegrab grabs a video frame to memory and screen
 * Functions:
                  IRIS Video Library functions used
                  vlOpenVideo()
                  vlGetNode()
                  vlCreatePath()
                  vlSetupPaths()
                  vlSetControl()
                  vlCreateBuffer()
                  vlRegisterBuffer()
                  vlGetActiveRegion()
                  vlGetNextValid()
                  vlPutFree()
                  vlBeginTransfer()
                  vlEndTransfer()
                  vlDeregisterBuffer()
                  vlDestroyPath()
                  vlDestroyBuffer()
                  vlCloseVideo()
                  vlPerror()
 * /
#include <stdlib.h>
#include <stdio.h>
#include <gl/gl.h>
#include <dmedia/vl.h>
char *_progName;
/* Report errors */
void
error_exit(void)
    vlPerror(_progName);
    exit(1);
```

```
void
main(int argc, char **argv)
    VLServer svr;
    VLPath path;
    VLNode src, drn;
    VLControlValue val;
    VLBuffer buffer;
    VLInfoPtr info;
    char *dataPtr;
    int c;
    int xsize;
    int ysize;
    long win;
    _progName = argv[0];
    foreground();
    /* Connect to the daemon */
    if (!(svr = vlOpenVideo("")))
        error_exit();
    /* Set up a drain node in memory */
    drn = vlGetNode(svr, VL_DRN, VL_MEM, VL_ANY);
    /* Set up a source node on any video source */
    src = vlGetNode(svr, VL_SRC, VL_VIDEO, VL_ANY);
    /* Create a path using the first device that will support it */
    path = vlCreatePath(svr, VL_ANY, src, drn);
    /* Set up the hardware for and define the usage of the path */
    if ((vlSetupPaths(svr, (VLPathList)&path, 1,
        VL_SHARE, VL_SHARE)) < 0)</pre>
        error_exit();
    /* Set the packing to RGB */
    val.intVal = VL_PACKING_RGB_8;
    vlSetControl(svr, path, drn, VL_PACKING, &val);
    /* Get the video size */
    vlGetControl(svr, path, drn, VL_SIZE, &val);
    xsize = val.xyVal.x;
    ysize = val.xyVal.y;
```

```
/* Set up and open a GL window to display the data */
prefsize(xsize,ysize);
win = winopen("Simplegrab Window");
RGBmode();
pixmode(PM_TTOB, 1);
gconfig();
/* Create and register a buffer for 1 frame */
buffer = vlCreateBuffer(svr, path, drn, 1);
if (buffer == NULL)
   error_exit();
vlRegisterBuffer(svr, path, drn, buffer);
/* Begin the data transfer */
if (vlBeginTransfer(svr, path, 0, NULL))
    error_exit();
/* Wait for a frame */
do {
    info = vlGetNextValid(svr, buffer);
} while (!info);
/* Get a pointer to the frame */
dataPtr = vlGetActiveRegion(svr, buffer, info);
/* Write the data to the screen */
lrectwrite(0,0, xsize-1, ysize-1, (ulong *)dataPtr);
/* Finished with frame, unlock the buffer */
vlPutFree(svr, buffer);
/* End the data transfer */
vlEndTransfer(svr, path);
/* Wait until the user presses a key */
printf("Press <Enter> to exit: ");
c = getc(stdin);
/* Cleanup before exiting */
vlDeregisterBuffer(svr, path, drn, buffer);
vlDestroyBuffer(svr, buffer);
vlDestroyPath(svr, path);
vlCloseVideo(svr);
```

### **Memory-to-video Frame Output**

Example 12-3 sends a frame to the video output (for systems that have a video output port).

**Example 12-3** Frame Output: *simplem2v.c* 

```
* Files:
                 simplem2v.c
* Usage:
                 simplem2v <filename>
* Description:
                Simplem2v sends a frame of image data from memory
                 to the video output. Image data must be in YUV422
                 format. Images in this format may be generated by
                 running the vidtomem application with the -r option
                 on an Indy Video board.
                 Simplem2v only runs on video hardware that has a
                 video output port. It will not run on a VINO video
                 board.
* Functions:
                 SGI Video Library functions used
                 vlOpenVideo()
                 vlGetNode()
                 vlCreatePath()
                 vlSetupPaths()
                 vlRegisterBuffer()
                 vlCreateBuffer()
                 vlGetTransferSize()
                 vlGetNextFree()
                 vlGetActiveRegion()
                 vlBufferDone()
                 vlBeginTransfer()
                 vlEndTransfer()
                 vlDeregisterBuffer()
                 vlDestroyBuffer()
                 vlDestroyPath()
                 vlCloseVideo()
                 vlGetErrno()
                 vlPerror()
                 vlStrError()
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/errno.h>
#include <dmedia/vl.h>
\#define MIN(x,y) ((x>y)?y:x)
extern int errno;
main(int argc, char **argv)
    VLServer svr;
    VLPath MEMtoVIDPath;
    VLNode src, drn;
    VLBuffer buf;
    VLInfoPtr info;
    struct stat status_buffer;
    char *dataPtr;
    ulong transferSize;
    int ret, fd;
    int c;
    char *_progName;
    char *fileName;
    _progName = argv[0];
    if (argc != 2)
        fprintf(stderr,"%s <filename>\n", _progName);
        exit(1);
    }
    fileName = argv[1];
    /* Connect to the daemon */
    if (!(svr = vlOpenVideo("")))
        fprintf(stderr,"%s: can't open video: %s\n", _progName,
                vlStrError(vlGetErrno()));
        exit(1);
    }
```

```
/* Set up a source node in memory */
src = vlGetNode(svr, VL_SRC, VL_MEM, VL_ANY);
/* Set up a video drain node on the first device that has one */
drn = vlGetNode(svr, VL_DRN, VL_VIDEO, VL_ANY);
/* Create a path using the selected devices */
MEMtoVIDPath = vlCreatePath(svr, VL_ANY, src, drn);
/* Set up the hardware for and define the usage of the path */
if (vlSetupPaths(svr, (VLPathList)&MEMtoVIDPath, 1, VL_SHARE, VL_SHARE)<0)
    fprintf(stderr, "%s: can't setup path: %s\n", _progName,
            vlStrError(vlGetErrno());
    exit(1);
}
/* Find out what size this path supports */
transferSize = vlGetTransferSize(svr,MEMtoVIDPath);
/* Create a ring buffer for the data transfers */
buf = vlCreateBuffer(svr, MEMtoVIDPath, src, 1);
/* Associate the ring buffer with the path */
vlRegisterBuffer(svr, MEMtoVIDPath, src, buf);
/* Get the next free frame in the buffer, reserve it for data */
do
    info = vlGetNextFree(svr, buf, transferSize);
} while (!info && !vlBufferDone(buf));
/* Get a pointer to where the data will go */
dataPtr = vlGetActiveRegion(svr, buf, info);
/* Open raw YUV data file */
fd = open(fileName, O_RDONLY);
if (!fd)
    fprintf(stderr,"%s: cannot open file %s.\n", _progName, fileName);
    exit(1);
}
```

```
/* Get the file's size (image size of this data)*/
if (fstat(fd,&status_buffer) == -1)
   perror(fileName);
   exit(1);
/* Make sure the hardware supports this image size */
if (status_buffer.st_size != transferSize)
    fprintf(stderr, "%s: The image is not the right size for this device\n",
    exit(1);
}
/* Read in the data */
ret = read(fd, dataPtr, transferSize);
close(fd);
/* Check the size of the data read in */
if (ret != transferSize)
    fprintf(stderr, "%s: Unable to read the image data\n", _progName);
   exit(1);
}
/* Put the data into the ring buffer */
vlPutValid(svr, buf);
/* Begin the data transfer */
vlBeginTransfer(svr, MEMtoVIDPath, 0, NULL);
/* Wait until user presses a key */
printf("Hit return to exit.\n");
c = getc(stdin);
/* End the data transfer */
vlEndTransfer(svr, MEMtoVIDPath);
/* Clean up and exit */
vlDeregisterBuffer(svr, MEMtoVIDPath, src, buf);
vlDestroyPath(svr, MEMtoVIDPath);
vlDestroyBuffer(svr, buf);
vlCloseVideo(svr);
```

# **Continuous Frame Capture**

Example 12-4 demonstrates continuous frame capture.

**Example 12-4** Continuous Frame Capture: *simplecapt.c* 

```
/*========A Simple Continuous Capture Application========
 * File:
                 simpleccapt.c
 * Usage:
                simpleccapt
 * Description: simpleccapt captures a stream of video to memory
 * Functions:
                IRIS Video Library functions used
                 vlOpenVideo()
                 vlGetNode()
                 vlCreatePath()
                 vlSetupPaths()
                 vlSetControl()
                 vlCreateBuffer()
                 vlRegisterBuffer()
                 vlGetActiveRegion()
                 vlGetNextValid()
                 vlPutFree()
                 vlBeginTransfer()
                 vlEndTransfer()
                 vlDeregisterBuffer()
                 vlDestroyPath()
                 vlDestroyBuffer()
                 vlCloseVideo()
                 vlPerror()
 * /
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <gl/gl.h>
#include <dmedia/vl.h>
char *_progName;
```

```
/* Report errors */
void
error_exit(void)
    vlPerror(_progName);
    exit(1);
void
main(int argc, char **argv)
    VLServer svr;
    VLPath path;
    VLNode src, drn;
    VLControlValue val;
    VLBuffer buffer;
    VLInfoPtr info;
    char *dataPtr;
    int c;
    int xsize;
    int ysize;
    long win;
    _progName = argv[0];
    foreground();
    /* Connect to the daemon */
    if (!(svr = vlOpenVideo("")))
        error_exit();
    /* Set up a drain node in memory */
    drn = vlGetNode(svr, VL_DRN, VL_MEM, VL_ANY);
    /* Set up a source node on any video source */
    src = vlGetNode(svr, VL_SRC, VL_VIDEO, VL_ANY);
    /* Create a path using the first device that will support it */
    path = vlCreatePath(svr, VL_ANY, src, drn);
    /* Set up the hardware for and define the usage of the path */
    if ((vlSetupPaths(svr, (VLPathList)&path, 1, VL_SHARE, VL_SHARE)) < 0)</pre>
        error_exit();
```

```
/* Set the packing to RGB */
val.intVal = VL_PACKING_RGB_8;
vlSetControl(svr, path, drn, VL_PACKING, &val);
/* Get the video size */
vlGetControl(svr, path, drn, VL_SIZE, &val);
xsize = val.xyVal.x;
ysize = val.xyVal.y;
/* Set up and open a GL window to display the data */
prefsize(xsize,ysize);
win = winopen("Simpleccapt Window");
RGBmode();
pixmode(PM_TTOB, 1);
gconfig();
/* Create and register a buffer for 1 frame */
buffer = vlCreateBuffer(svr, path, drn, 1);
if (buffer == NULL)
    error_exit();
vlRegisterBuffer(svr, path, drn, buffer);
/* Begin the data transfer */
if (vlBeginTransfer(svr, path, 0, NULL))
    error_exit();
printf("Type <control-c> to exit.\n");
for(;;) {
   do {
        sginap(1);
                            /* wait a tick */
        info = vlGetNextValid(svr, buffer);
    } while (!info);
    /* Get a pointer to the frame */
    dataPtr = vlGetActiveRegion(svr, buffer, info);
    /* Write the data to the screen */
    lrectwrite(0,0, xsize-1, ysize-1, (ulong *)dataPtr);
    /* Finished with frame, unlock the buffer */
    vlPutFree(svr, buffer);
}
```

```
/* End the data transfer */
vlEndTransfer(svr, path);

/* Cleanup before exiting */
vlDeregisterBuffer(svr, path, drn, buffer);
vlDestroyBuffer(svr, buffer);
vlDestroyPath(svr, path);
vlCloseVideo(svr);
}
```

# Chapter 13

# **Using VL Controls**

This chapter explains how to use the VL controls to set and adjust video parameters.

# Using VL Controls

### VL controls enable you to:

- specify data transfer parameters, such as the frame rate or count
- specify the capture region and decimation, or output window
- specify video format and timing
- adjust signal parameters, such as hue, brightness, vertical sync, horizontal sync
- specify sync source

#### Topics in this chapter include:

- VL Control Type and Values
- VL Control Fraction Ranges
- VL Control Classes
- VL Control Groupings
- Galileo Video Controls
- VINO Controls

Device-independent controls are documented in /usr/include/dmedia/vl.h. Device-dependent controls are documented in the respective header files for the devices: dmedia/vl\_vino.h (VINO), dmedia/vl\_ev1.h (Galileo Video), and dmedia/vl\_sv1.h (Sirius Video).

Table 13-1 lists device-independent VL controls alphabetically, along with their values or ranges. See the *Sirius Video Owner's and Programming Guide* for Sirius Video controls.

**Note:** All controls are available for all platforms unless otherwise noted. The reference "Galileo Video" includes Indigo<sup>2</sup> Video and Indy Video, unless otherwise noted.

 Table 13-1
 Device-Independent VL Controls

Control	Sets	VINO	Indigo <sup>2</sup> Video and Indy Video	Galileo Video
VL_BLEND_A	Input source for foreground (channel A) image	N/A		erived from vlGetNode(); the source nodes
VL_BLEND_B	Input source for background (channel B) image	N/A		erived from vlGetNode(); the source nodes
VL_BLEND_A_ALPHA	Input source for foreground (channel A) alpha	N/A	N/A	N/A
VL_BLEND_B_ALPHA	Input source for background (channel B) alpha	VLNode type derived fr nodes	rom vlGetNode();	must be one of the source
VL_BLEND_A_FCN	Blend function that controls mixing of foreground (channel A) signals	N/A	VL_BLDFCN_ZERO VL_BLDFCN_ONE VL_BLDFCN_A_ALPHA: (foreground alpha)/255 VL_BLDFCN_MINUS_A_ALPHA: 1 - ((foreground alpha) / 255)	
VL_BLEND_B_FCN	Blend function that controls mixing of background (channel B) signals	N/A	VL_BLDFCN_ZERO VL_BLDFCN_ONE VL_BLDFCN_B_ALPHA: (background alpha)/255 VL_BLDFCN_MINUS_B_ALPHA: 1 - ((background alpha) / 255)	
VL_BLEND_A_NORMALIZE	Follows Porter-Duff model (background (channel A) pixels premultiplied by their corresponding alphas before blending)	N/A	N/A	N/A

 Table 13-1 (continued)
 Device-Independent VL Controls

Control	Sets	VINO	Indigo <sup>2</sup> Video and Indy Video	Galileo Video
VL_BLEND_B_NORMALIZE	Premultiplies foreground (channel B) by alpha	N/A	0 = off 1 = on	0 = off 1 = on
VL_BLEND_OUT_NORMALIZE	Scaled output from blender	N/A	N/A	N/A
VL_BRIGHTNESS	Brightness	N/A	N/A	N/A
VL_CAP_TYPE	Type of frame(s) or field(s) to capture; see "Interlacing" in Chapter 11	VL_CAPTURE_NONINT VL_CAPTURE_INTERLI VL_CAPTURE_EVEN_F VL_CAPTURE_ODD_FII	EAVED IELDS	
VL_CONTRAST		N/A	N/A	N/A
VL_DEFAULT_SOURCE  VL_FORMAT	Default source for the video path  Video format	VL_VINO_SRC_DV_IN VL_VINO_SRC_AV_IN If either VINO analog channel is active when the first video application starts, the default is analog; otherwise, the default is digital (corresponding to the IndyCam).		Analog: set 0 Digital 1: set 1 Digital 2: set 2 Indigo <sup>2</sup> Video, Indy Video: Analog only: set 0  VL_FORMAT_RGB (output only) VL_FORMAT_BETACAM
VL_FREEZE	Data transfer freeze; suspends transfer at the memory drain node, with	0 = off 1 = on	0 = off 1 = on	(input and output) VL_FORMAT_SMPTE_YUV (input and output) $0 = off$ $1 = on$
VL_H_PHASE	no picture regeneration Horizontal phase	N/A		Integer VL_EV1_H_PHASE
VL_HUE	Hue; the control panel <i>vcp</i> does numerator and denominator calculations	(-180,178 19/32) in steps	of 1 13/32 degree	es

 Table 13-1 (continued)
 Device-Independent VL Controls

Control	Sets	VINO	Indigo <sup>2</sup> Video and Indy Video	Galileo Video
VL_MUXSWITCH	Switch between inputs on a single path, corresponding to the physical connector to the option	Analog node VL_VINO_SRC_AV_I VL_VINO_COMPOSITE VL_VINO_SVIDEO Digital node VL_VINO_SRC_DV_IN: VL_VINO_INDYCAM VL_VINO_CCIR601	Y/C (S-Video connector):	S-Video input 1: set 0 input 2: set 1 input 3: set 2 Composite input 1: set 3 input 2: set 4; input 3: set 5 Y/R-Y/B-Y input 1: set 6 input 2: set 7
VL_OFFSET	On VL_VIDEO nodes, the offset to the active region of the video; on all other nodes, the offset within the video  Because the default is 0,0,			
	use negative values to get blanking data			
VL_ORIGIN	Upper left corner of image in drain (usually a window); the offset within the node;	N/A	Coordinates; de	fault is 0,0

 Table 13-1 (continued)
 Device-Independent VL Controls

Control	Sets	VINO	Indigo <sup>2</sup> Video and Indy Video	Galileo Video
VL_PACKING	Packing of video data at source or drain	VL_PACKING_ABGR_8 VL_PACKING_AUYV_8 VL_PACKING_AYU_AY VL_PACKING_A_2_BG VL_PACKING_BGR_33: VL_PACKING_BGR_33: VL_PACKING_RBG_32: VL_PACKING_RBG_32: VL_PACKING_RBG_32: VL_PACKING_RGB_10 VL_PACKING_RGB_33: VL_PACKING_RGB_33: VL_PACKING_RGB_33: VL_PACKING_RGB_33: VL_PACKING_RGB_36: VL_PACKING_RGB_56: VL_PACKING_RGB_56: VL_PACKING_RGB_56: VL_PACKING_RGB_56: VL_PACKING_RGB_8 VL_PACKING_RGB_8 VL_PACKING_RGB_8 VL_PACKING_YUY_41: VL_PACKING_YUV_44: VL_PACKING_YUV_45: PACKING_YUV_46: PACKING_YUV_46: PACKING_YUV_46: PACKING_YUV_46: PACKING_YUV_46: PACKING_YUV_46: PACKING_YUV_46: PACKING_YUV_46: PACKING_Y_8_IP VL_PACKING_Y_8_IP	S (V_10   R_10   V_10   R_10   V_10   V_10   V_10   V_2   IP   V_2   IP   V_3   V_444_10   V_444_8   V_4   V_48   V_88   V_88   V_88   V_88	<del>)</del>
VL_RATE	Transfer rate in fields or frames	With a VL_CAP_TYPE or NTSC: 5, 6, 10, 12, 15, 18 PAL: 5, 10, 15, 20, 25 With a VL_CAP_TYPE or VL_CAPTURE_EVEN_FUL_CAPTURE_ODD_FINTSC: 10, 12, 20, 24, 30, PAL: 10, 20, 30, 40, 50	, 20, 24, 25, 30 of VL_CAPTURE_ PIELDS, or ELDS:	NONINTERLEAVED only: INTERLEAVED,
VL_SIGNAL		N/A	N/A	N/A
VL_SIZE	On VL_VIDEO nodes, the size of the video; on all other nodes, the clipped size of the video	Coordinates; default dep	ends on signal	

 Table 13-1 (continued)
 Device-Independent VL Controls

Control	Sets	VINO	Indigo <sup>2</sup> Video and Indy Video	Galileo Video
VL_SYNC	Sync mode	N/A	VL_SYNC_INT VL_SYNC_GEN	
VL_SYNC	Slave sync mode	N/A	N/A	VL_EV1_SYNC_SLAVE
VL_SYNC_SOURCE	Sets sync source for analog breakout box	N/A only one sync input	Composite 1: set 0 Composite 2: set 2	Composite 1: set 0 Composite 2: set 1 Composite 3: set 2
VL_TIMING	Video timing	Default: timing produced by active signal For Betacam, MII, composite tape formats: Analog: 12.27 MHz, 646 x 486 (NTSC): VL_TIMING_525_SQ_PIX Analog: 14.75 MHz, 768 x 576 (PAL): VL_TIMING_625_SQ_PIX (VINO default) For D1 tape formats: Digital component: 13.50 MHz, 720 x 486: VL_TIMING_525_CCIR601 Digital component: 13.50 MHz, 720 x 576: VL_TIMING_625_CCIR601 For D2 tape formats:		_TIMING_525_SQ_PIX TIMING_625_SQ_PIX (VINO VL_TIMING_525_CCIR601 VL_TIMING_625_CCIR601
		4X PAL subcarrier, 17.72	•	: VL_TIMING_525_4FSC VL_TIMING_625_4FSC
VL_V_PHASE	Vertical phase	N/A	Integer	Integer
VL_WINDOW	Window ID for video in a window (screen node only)	N/A	Integer	Integer
VL_ZOOM	Zoom factor for video stream; fractions greater than 1 expand the picture, fractions less than one reduce the picture	1/1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8	4/1, 2/1, 1/1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8	4/1, 2/1, 1/1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8

**Note:** For information on controls for keying, blending, or wipes, see Chapter 15, "VL Blending."

For detailed information on using VL\_CAP\_TYPE, VL\_FORMAT, VL\_MUXSWITCH, VL\_OFFSET, VL\_PACKING, VL\_RATE, VL\_SIZE, VL\_TIMING, and VL\_ZOOM, see "Setting Parameters for Data Transfer to or from Memory" in Chapter 12.

### **VL Control Type and Values**

```
The type of VL controls is:
```

```
typedef long VLControlType;
```

Common types used by the VL to express the values returned by the controls are:

```
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 */
  uint numFractRanges; /* number of ranges */
  VLFractionRange *ranges; /* range of values of control */
  uint numItems; /* number of enumerated items */
  VLControlItem *itemList; /* the actual enumerations */
} VLControlInfo;
```

To store the value of different controls, *libvl.a* uses the struct:

```
typedef union {
  VLFraction fractVal;
  VLBoolean boolVal;
  int intVal;
  VLXY xyVal;
  uint pad[24];
} VLControlValue;

typedef struct {
  int x, y;
} VLXY;

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

The control info structure is returned by a **vlGetControlInfo()** call, and it contains many of the items discussed above.

*VLControlInfo.number* is the number of the VLControlInfo.node that the info pertains to. There may be several controls of the same type on a particular node, but usually there is just one.

*VLControlInfo.numFractRanges* is the number of fraction ranges for a particular control. The names correspond 1-to-1 with the *rangeNames*, up to the number of range names, *numRangeNames*. That is, there may be fewer names than ranges, but never more.

### **VL Control Fraction Ranges**

The VL uses fraction ranges to represent the values possible for a control. A VLFractionRange generated by the VL is guaranteed never to generate a fraction with a zero denominator, or a fractional numerator or denominator.

For a VLProgressionType of VL\_LINEAR, *numerator.increment* and *denominator.increment* are guaranteed to be greater than zero, and the limit is always guaranteed to be {numerator,denominator}.base, plus some integral multiple of {numerator,denominator}.increment.

The type definition for fraction types in the header file is:

```
typedef struct {
   VLRange numerator;
   VLRange denominator;
} VLFractionRange;
```

#### VL Control Classes

The VL defines control classes for user-interface developers. The classes are hints only; they are the VL developer's idea of how the control is commonly represented in the real world.

```
#define VL_CLASS_NO_UI 0
#define VL_CLASS_SLIDER 1
#define VL_CLASS_KNOB 2
#define VL_CLASS_BUTTON 3
#define VL_CLASS_TOGGLE 4
#define VL_CLASS_DETENT_KNOB 5
#define VL_CLASS_LIST 6
```

In the list above, VL\_CLASS\_NO\_UI is often used for controls that have no user-interface metaphor and are not displayed in the video control panel or saved in the defaults file.

The VL controls can be read-only, write-only, or both. The VL includes these macros:

```
#define VL_CLASS_RDONLY
                             0x8000
                                        /* control is read-
only */
                              0x4000
                                        /* control is write-
#define VL_CLASS_WRONLY
only */
#define VL_IS_CTL_RDONLY(x) ((x)->class & VL_CLASS_RDONLY)
#define VL_IS_CTL_WRONLY(x) ((x)->class & VL_CLASS_WRONLY)
#define VL_IS_CTL_RW(x)
                             (!(VL_IS_CTL_RDONLY(x) &&
                                 VL_IS_CTL_WRONLY(x)))
to test these conditions:
#define VL_CLASS_MASK
                              0xfff
typedef unsigned long VLControlClass; /* from list above */
```

### **VL Control Groupings**

Like control class, control grouping is an aid for the user-interface developer. The groupings are the VL developer's idea of how the controls would be grouped in the real world. These groupings are implemented in the video control panel *vcp*.

```
The type definition for groupings is:
```

```
typedef char NameString[80];
#define VL_CTL_GROUP_PATH 9 /* Path Controls */
```

The maximum length of a control or range name is VL\_NAME\_SIZE.

Table 13-2 summarizes the VL control groupings.

 Table 13-2
 VL Control Groupings

Grouping	Includes controls for
VL_CTL_GROUP_BLENDING	Blending; for example, VL_BLEND_B
VL_CTL_GROUP_VISUALQUALITY	Visual quality of sources or drains; for example, VL_H_PHASE or VL_V_PHASE
VL_CTL_GROUP_SIGNAL	Signal of sources or drains; for example, VL_MUXSWITCH or VL_HUE
VL_CTL_GROUP_CODING	Encoding or decoding sources or drains; for example, VL_TIMING or VL_FORMAT
VL_CTL_GROUP_SYNC	Synchronizing video sources or drains; for example, VL_SYNC
VL_CTL_GROUP_ORIENTATION	Orientation or placement of video signals; for example, VL_ORIGIN
VL_CTL_GROUP_SIZING	Setting the size of the video signal; for example, VL_SIZE
VL_CTL_GROUP_RATES	Setting the rate of the video signal; for example, VL_RATE
VL_CTL_GROUP_WS	Specifying the windowing system of the workstation; for example, VL_WINDOW
VL_CTL_GROUP_PATH	Specifying the data path through the system; these controls, often marked with the VL_CLASS_NO_UI, are often internal to the VL, with no direct access for the user
VL_CTL_GROUP_SIGNAL_ALL	Specifying properties of all signals
VL_CTL_GROUP_SIGNAL_COMPOSITE	Specifying properties of composite signals
VL_CTL_GROUP_SIGNAL_COMPONENT	Specifying properties of component signals
VL_CTL_GROUP_SIGNAL_CLUT_COMPOSITE	Specifying properties of composite color lookup table (CLUT) controls
VL_CTL_GROUP_SIGNAL_CLUT_COMPONENT	Specifying properties of component CLUT controls
VL_CTL_GROUP_KEYING	Specifying properties of chroma or luma keying controls, such as VL_KEYER_FG_OPACITY
VL_CTL_GROUP_PRO	Specifying values not commonly found on the front panel of a real-world video device; for example, a wipe control
VL_CTL_GROUP_MASK	Masking optional bits to extract only the control group

### **Galileo Video Controls**

VL controls that are used only for Galileo Video and VINO on the Indy workstation fall into several categories:

- general controls for Galileo Video and VINO
- Galileo Video encoder and color-space conversion controls
- Galileo 601 Video digital breakout box controls
- VINO analog input controls
- IndyCam controls

**Note:** Galileo Video keying controls are documented in Chapter 15, "VL Blending."

Each category is explained separately in this section.

# **General Controls for Galileo Video**

The Galileo Video controls are summarized in Table 13-3 in alphabetical order.  $\,$ 

**Table 13-3**Galileo Video *vcp* Controls

Galileo Video	Purpose
VL_EV1_AGC_CONTROL	Sets automatic gain control speed for chrominance for composite or Y/C
VL_EV1_ALPHA_NOT_PIXEL Value:	Determines whether information is derived from the alpha out or the pixel out channel
VL_EV1_ANTI_DITHER Value: (0,1) where 0 = off, 1 = on	Removes interference between frequency components generated by dithered graphics images (Y/C and composite out only) and chrominance frequency present in video signals by using a notch filter in luminance
VL_EV1_APERTURE	Sets aperture factors for luminance for composite and Y/C inputs
VL_EV1_BANDPASS	Selects bandpass filters for luminance for composite and Y/C inputs
VL_EV1_BLANK_LINE Range: (0,63)	Sets first unblanked line on all analog video outputs
VL_EV1_BLEND_B_FLAT	Sets a flat backgroud color
VL_EV1_BLEND_B_U	Sets the U value of a flat background color
VL_EV1_BLEND_B_V	Sets the V value of a flat background color
VL_EV1_BLEND_B_Y	Sets the Y value of a flat background color
VL_EV1_BLEND_H_FILT	Controls blending
VL_EV1_BLEND_SHADOW_GAIN	Controls blending
VL_EV1_BLEND_SHADOW_OFFSET	Controls blending
VL_EV1_BLEND_SHADOW_ON	Controls blending
VL_EV1_BOTTOM_FLUTTER	For CCIR 601 (13.5 MHz) sampling only, compensates for horizontal phase jump on the selected line number $$
VL_EV1_C_GAIN Range: (0,255)	Adjusts burst and chrominance output level of composite and Y/C simultaneously
VL_EV1_CHROMA_BAND Value: enhanced = 0, standard = 1	Selects standard chrominance bandwidth of about 1.3 MHz or enhanced bandwidth (nonstandard) of about 2.5 MHz for composite and $\rm Y/C$ outputs

Table 13-3 (continued)	Galileo Video vcp Controls
------------------------	----------------------------

Galileo Video	Purpose
VL_EV1_CHROMA_DELAY	For CCIR 601 (13.5 MHz) sampling only, changes composite or Y/C chrominance delay without affecting luminance delay
VL_EV1_CHROMA_GAIN	Fine-tunes chroma gain for composite and Y/C inputs
VL_EV1_COLOR_IN_ON	
VL_EV1_COLOR_KILL_THRES	Controls level at which burst amplitude decides if composite or $Y/C$ input is color or monochrome when color mode is automatically set
VL_EV1_COLOR_OUT_ON	Makes composite or Y/C output into monochrome by turning off color burst and chrominance
VL_EV1_CORING	Selects coring levels for luminance for composite and Y/C inputs
VL_EV1_DEINTERLACE	For graphics to video, turns off interlace for flicker reduction; for video to graphics, interlaces video image
VL_EV1_DELAY_SYNC Range: (0,63)	Same as VL_EV1_H_OFFSET but with a narrow range: resolution in pixel clock steps
VL_EV1_DOMINANCE_FIELD Value: 0 = odd field, 1 = even field	Triggers on odd or even fields
VL_EV1_FILTER	Filters decimated video images to smooth jagged edges
VL_EV1_GENLOCK_SRC	
VL_EV1_H_OFFSET	Delays timing of entire video signal (sync and picture) relative to timing reference such as genlock; no effect in slave mode for output timing
VL_EV1_H_PHASE Range: (0,63)	In genlock output timing, provides a small-range horizontal phase adjustment for all outputs; in slave output timing mode, moves both h-phase and horizontal picture position for composite and $Y/C$ output (no effect on RGB or $Y/R-Y/B-Y$ in slave mode)
VL_EV1_H_PICTURE_POSITION	Controls horizontal position of input picture: 2 pixel steps for composite and $Y/C$ resolution; 1 pixel step for $Y/R-Y/B-Y$ input resolution
VL_EV1_LOCK_PORT0	
VL_EV1_LUMA_DELAY	Changes composite or Y/C luminance delay without affecting chrominance delay

Galileo Video	Purpose
VL_EV1_PEAK_WHITE	Sets expected peak white amplitude; sets setup/no setup (pedestal) on black for Y or $Y/R-Y/B-Y$ input
	Betacam: use 714 mV SMPTE and EBU: use 700 mV or no setup
VL_EV1_PREFILTER	Boosts luminance frequency response for composite and Y/C formats
VL_EV1_QUALITY	Allows locking to unstable video source, such as videotape recorders with no timebase correction
VL_EV1_RGB_GAIN Range: (0,63)	Adjusts output level of $Y/R-Y/B-Y$ simultaneously (no effect on sync pulse amplitude)
VL_EV1_SCH_PHASE Range: (0,255)	Adjusts SC-H phase +/- 180 degrees
VL_EV1_SUB_FREQ Range: (0,255); total range +/- 450 ppm	Provides fine adjustment of composite and Y/C output color subcarrier frequency
VL_EV1_SVHS_CHROMA  Values; the first is the default:  VL_EV1_COLOR_MODE_AUTO  VL_EV1_COLOR_MODE_COLOR  VL_EV1_COLOR_MODE_MONO	Selects automatic detection of color or monochrome from the burst, forces monochrome, or forces color
VL_EV1_SYNC_LEVEL Value: 286 mV = 0, 300 mV = 1	Selects sync pulse amplitude on G/Y output
VL_EV1_SYNC_SLAVE	Selects the slave sync mode when genlocking is not required. In this mode, the Galileo board is slaved to the input source. This is the best mode for capturing images because it provides the highest capture rate.
VL_EV1_TBC_MODE Value: (0,1) where 0 = off, 1 = on	When video input and output timings are not frequency-locked, removes small timing errors in analog video input; video input must go directly into a video framebuffer for proper operation; can also be used to achieve frame synchronization; makes no digital input unavailable
VL_EV1_TRIGGER_LINE Range: (0,100)	Determines line number on which trigger event happens
VL_EV1_TRIGGER_POLARITY Values: 0 = negative, 1 = positive	Determines whether a trigger event occurs on a 0 to 1 or a 1 to 0 transition

 Table 13-3 (continued)
 Galileo Video vcp Controls

Galileo Video	Purpose
VL_EV1_UV_LEVEL Value: high = 0, low = 1	Changes R-Y and B-Y analog output levels to accommodate different standards, such as Beta, SMPTE, and EBU; set to low for RGB out
VL_EV1_U_GAIN_ROUGH Range: (0,63)	Controls gain of B-Y component input in coarse steps
VL_EV1_U_GAIN_VERNIER Range: (0,63)	Controls gain of B-Y component input in fine steps
VL_EV1_V_GAIN_ROUGH Range: (0,63)	Controls gain of R-Y component input in coarse steps
VL_EV1_V_GAIN_VERNIER Range: (0,63)	Controls gain of R-Y component input in fine steps
VL_EV1_V_OFFSET	Delays timing of entire video signal (sync and picture) relative to timing reference such as genlock; no effect in slave mode for output timing
VL_EV1_VNOISE_REDUCER	Selects mode of vertical noise reduction
VL_EV1_YC_GAIN Range: (0,255)	Adjusts output level (sync, burst, luminance, and chrominance) of composite and Y/C simultaneously $$
VL_EV1_YG_SYNC Value: (0,1) where 0 = off, 1 = on	Turns sync pulse on or off

# Galileo Video IndyCam Controls

Table 13-4 lists the Galileo Video IndyCam controls.

 Table 13-4
 Galileo Video IndyCam Controls

Galileo Video Control	Values	Sets
VL_EV1_INDYCAM_AGCENA	0 = off 1 (default) = on	Automatic gain control (AGC)
VL_EV1_INDYCAM_AWBCTL	0 (momentary default) = off 1 = on	Automatic white balance
VL_EV1_INDYCAM_BLUE_BALANCE	0,255,255; set by camera's white balance	Blue balance
VL_EV1_INDYCAM_BLUE_SATURATION	1	
VL_EV1_INDYCAM_BRIGHTNESS	Read-only value determined by input from IndyCam	Luma level (read-only)
VL_EV1_INDYCAM_GAIN	0,255,255; set by AGC in camera	Gain
VL_EV1_INDYCAM_RED_BALANCE	0,255,255; set by camera's white balance	Red balance
VL_EV1_INDYCAM_SATURATION	0,170,170	Saturation
VL_EV1_INDYCAM_SHUTTER	VL_EV1_INDYCAM_SHUTTER_60 VL_EV1_INDYCAM_SHUTTER_100 VL_EV1_INDYCAM_SHUTTER_125 VL_EV1_INDYCAM_SHUTTER_250 VL_EV1_INDYCAM_SHUTTER_500 VL_EV1_INDYCAM_SHUTTER_1000 VL_EV1_INDYCAM_SHUTTER_2000 VL_EV1_INDYCAM_SHUTTER_4000 VL_EV1_INDYCAM_SHUTTER_10000	Shutter speed
	Default: VL_EV1_INDYCAM_SHUTTER_1000	
VL_EV1_INDYCAM_SHUTTER_SNAP		

# **Galileo Video Encoder and Color-Space Conversion Controls**

Encoder controls for Galileo Video encode digital video from the system into the analog data stream. The controls are summarized in Table 13-5.

 Table 13-5
 Galileo Video Encoder and Color-Space Conversion Controls

Encoder control	Color-space conversion control	Purpose
VL_EV1_ENC_BLANK	VL_EV1_CSC_BLANK	Sets digital level for blanking
VL_EV1_ENC_BLACK	VL_EV1_CSC_BLACK	Sets digital level for black
VL_EV1_ENC_WHITE	VL_EV1_CSC_WHITE	Sets digital level for white
VL_EV1_ENC_UVGAIN	VL_EV1_CSC_UVGAIN	Sets color difference gain factor
VL_EV1_ENC_QUANTIZE	VL_EV1_CSC_QUANTIZE	Sets number of quantization levels; set to maximum for no effect
VL_EV1_ENC_LOAD	VL_EV1_CSC_LOAD	Loads default table
N/A	VL_EV1_CSC_SUBADDR	Selects component table to load with custom setup

# **Galileo 601 Video Digital Breakout Box Controls**

General controls for the Galileo 601 Video Digital Breakout Box are summarized in Table 13-6.

 Table 13-6
 Galileo 601 Video Digital Breakout Box General Controls

Control	Purpose
VL_EV1_DBOB_INPUT	Selects serial or parallel input
VL_EV1_DBOB_INPUT2	Selects serial or parallel input/output
VL_EV1_DBOB_VBSELECT Value: 0 = normal, 1 = narrow	Determines whether vertical blanking information passes through the Galileo 601 Video option: normal setting blanks out the vertical blanking information; narrow setting passes it through
VL_EV1_DBOB_PIXEL_MODE Values: 0 = 13.5 (601CCIR), 1 = square	Sets Galileo 601 Video pixel format
VL_EV1_DBOB_PIXEL_FORMAT	Selects analog output format on digital breakout box outputs
VL_EV1_DBOB_LINE Values: 0 = 525, 1 = 625	Sets 525 (NTSC) or 625 (PAL) timing
VL_EV1_DBOB_DELAY Range: (0,255)	Sets analog output delay on Galileo 601 Video

See the <code>dmedia/cl\_cosmo.h</code> header file for Cosmo Compress video parameters (CL\_COSMO\_VIDEO\_\*) that work with the Galileo 601 Video Digital Breakout Box and the Compression Library.

### **Color-Space Conversion Controls**

Table 13-7 summarizes color-space conversion controls for the Galileo 601 Video Digital Breakout Box.

 Table 13-7
 Galileo Video Digital Breakout Box Color-Space Conversion Controls

Control	Purpose
VL_EV1_DBOB_CSC_BLANK	Sets digital level for blanking
VL_EV1_DBOB_CSC_BLACK	Sets digital level for black
VL_EV1_DBOB_CSC_WHITE	Sets digital level for white
VL_EV1_DBOB_CSC_UVGAIN	Sets color difference gain factor
VL_EV1_DBOB_CSC_QUANTIZE	Sets number of quantization levels; set to maximum for no effect
VL_EV1_DBOB_CSC_LOAD	Loads default table
VL_EV1_DBOB_CSC_SUBADDR	Selects component table to load with custom setup

#### **Galileo Video DAC Controls**

Table 13-8 summarizes the Galileo Video digital-to-analog converter (DAC) controls; all but the last two appear in the All submenu of the *vcp* Pro menu. The range for each control is (0,63).

 Table 13-8
 Galileo Video DAC controls

Control	Sets
VL_EV1_DBOB_DAC_0	Y or green channel gain
VL_EV1_DBOB_DAC_1	Y or green channel offset
VL_EV1_DBOB_DAC_2	B-Y or blue channel gain
VL_EV1_DBOB_DAC_3	B-Y or blue channel offset
VL_EV1_DBOB_DAC_4	R-Y or red channel gain
VL_EV1_DBOB_DAC_5	R-Y or red channel offset
VL_EV1_DBOB_DAC_6	Chroma gain; not in vcp (no UI)
VL_EV1_DBOB_DAC_7	Chroma offset (no UI)

# **VINO Controls**

This section describes the VINO controls.

# **VINO Video Control Panel Controls**

Table 13-9 lists the general VINO controls.

Table 13-9VINO vcp Controls

VINO	Purpose
VL_VINO_APERTURE; default 1	Sets aperture factors for luminance for composite and Y/C inputs
VL_VINO_BANDPASS; default 0	Selects bandpass filters for luminance for composite and Y/C inputs
VL_VINO_CHROMA_AGC; default 0	Sets automatic gain control speed for chrominance for composite or Y/C
VL_VINO_CHROMA_GAIN default: 44	Fine-tunes chroma gain for composite and Y/C inputs
VL_VINO_COLOR_KILL_THRES default: 30 dB	Controls level at which burst amplitude decides if composite or $Y/C$ input is color or monochrome when color mode is automatically set
VL_VINO_CORING; default 0	Selects coring levels for luminance for composite and Y/C inputs
VL_VINO_H_PICTURE_POSITION default: 244 (both NTSC and PAL)	Controls horizontal position of input picture: 2 pixel steps for composite and Y/C resolution; 1 pixel step for Y/R-Y/B-Y input resolution
VL_VINO_LUMA_DELAY; default 0	Changes composite or Y/C luminance delay without affecting chrominance delay
VL_VINO_PREFILTER; default 0	Boosts luminance frequency response for composite and Y/C formats
VL_VINO_COLOR_MODE  Values; the first is the default:  VL_VINO_COLOR_MODE_AUTO  VL_VINO_COLOR_MODE_COLOR  VL_VINO_COLOR_MODE_MONO	Selects automatic detection of color or monochrome from the burst, forces monochrome, or forces color
VL_VINO_VNOISE_REDUCER Values: normal: set 0 (the default) search: set 1 auto: set 2 bypass: set 3	Selects mode of vertical noise reduction

# **VINO Analog Input Controls**

Table 13-10 summarizes input controls specific to VINO.

 Table 13-10
 VINO Analog Input Controls

		o i		
Control	Range	Default: 60 Hz (NTSC)	Default: 50 Hz (PAL)	Use
VL_VINO_HREF_GENERATION	(0,1) 0 = off, 1 = on	1	1	Shifts the line 8 pixels to the right.
VL_VINO_PAL_SENS	Fraction range: 0,255,1	N/A	144	In PAL timing, the chroma modulation phase inverts every line. Dropouts off the tape can disrupt this pattern. Use this control to set the recovery time constant (maximum for poor quality tape).
VL_VINO_AUFD	(0,1) 0 = off, 1 = on	1	1	Sets automatic field detect.
VL_VINO_ZOOM_X	(0,1) 0 = off, 1 = on	0	0	Specifies zoom or decimation in x direction only, to maintain aspect ratio for capturing only even or odd fields.
VL_VINO_ALPHA	(0,255)	255	255	Sets value placed in user's framebuffers for the alpha value of the RGBA pixel format. Can be changed while capture is in progress. See Graphics Library documentation for typical uses.
VL_VINO_EVEN_OFFSET	0MaxHeight	0	0	Sets offset or clipping value separately for even fields; for example, if this control is set to 2 and VL_VINO_ODD_OFFSET is set to 0, the following lines are captured in NTSC: odd line 21, even line 26, odd line 23, even line 28.
VL_VINO_ODD_OFFSET	0MaxHeight	0	0	Sets offset or clipping value separately for odd fields.

# **VINO IndyCam Controls**

The controls for the IndyCam are summarized in Table 13-11.

Table 13-11IndyCam Controls

VINO	Values	Sets
VL_VINO_INDYCAM_AGCENA	0 = off 1 (default) = on	Automatic gain control (AGC)
VL_VINO_INDYCAM_AWBCTL	0 (momentary default) = off 1 = on	Automatic white balance
VL_VINO_INDYCAM_BLUE_BALANCE	0,255,255; set by camera's white balance	Blue balance
VL_VINO_INDYCAM_BRIGHTNESS	Read-only value determined by input from IndyCam	Luma level (read-only)
VL_VINO_INDYCAM_GAIN	0,255,255; set by AGC in camera	Gain
VL_VINO_INDYCAM_RED_BALANCE	0,255,255; set by camera's white balance	Red balance
VL_VINO_INDYCAM_SATURATION	0,170,170	Saturation
VL_VINO_INDYCAM_SHUTTER	VL_VINO_INDYCAM_SHUTTER_60 VL_VINO_INDYCAM_SHUTTER_100 VL_VINO_INDYCAM_SHUTTER_125 VL_VINO_INDYCAM_SHUTTER_250 VL_VINO_INDYCAM_SHUTTER_500 VL_VINO_INDYCAM_SHUTTER_1000 VL_VINO_INDYCAM_SHUTTER_2000 VL_VINO_INDYCAM_SHUTTER_4000 VL_VINO_INDYCAM_SHUTTER_10000	Shutter speed
	Default: VL_VINO_INDYCAM_SHUTTER_1000	

# Chapter 14

# **VL Event Handling**

This chapter explains how to handle video events.

# VL Event Handling

The VL provides several ways of handling data stream events, such as completion or failure of data transfer, vertical retrace event, loss of the path to another client, lack of detectable sync, or dropped fields or frames. The method you use depends on the kind of application you're writing:

- For a strictly VL application, use:
  - vlSelectEvents() to choose the events to which you want the application to respond
  - vlAddCallback() to specify the function called when the event occurs
  - your own event loop or a main loop (vlMainLoop()) to dispatch the events
- For an application that also accesses another program or device driver, or if you're adding video capability to an existing X or OpenGL application, set up an event loop in the main part of the application and use the IRIX file descriptor (FD) of the event(s) you want to add.

Topics in this chapter include:

- Querying VL Events
- Creating a VL Event Loop
- Creating a Main Loop with Callbacks

This chapter concludes with an example illustrating a main loop and event loops.

# **Querying VL Events**

General VL event handling routines are summarized in Table 14-1.

 Table 14-1
 VL Event Handling Routines

Routine	Use	
vlGetFD()	Get a file descriptor for a VL server	
vlNextEvent()	Gets the next event; blocks until you get the next event from the queue	
vlCheckEvent()	Like a nonblocking vlNextEvent(), this call checks to see if you have an event waiting of the type you specify and reads it off the queue without blocking	
vlPeekEvent()	Copies the next event from the queue but, unlike vlNextEvent(), does not update the queue, so that you can see the event without processing it	
vlSelectEvents()	Selects video events of interest	
vlPending()	Queries whether there is an event waiting for the application	
vlEventToName()	Gets the character string with the name of the event; for example, to use in messages	
vlAddCallback()	Adds a callback; use for VL events	
vlRemoveCallback()	Removes a callback for the events specified if the client data matches that supplied when adding the callback	
vlRemoveAllCallbacks()	Removes all callbacks for the specified path and events	
vlCallCallbacks()	Creates a handler; used when creating a main loop or using a supplied, non-VL main loop	
vlRegisterHandler()	Registers an event handler; use for non-VL events	
vlRemoveHandler()	Removes an event handler	

The event type is an integer. **vlEventToName()** allows you to get the character string with the name of the event, so that you can use the event name, for example, in messages.

Table 14-2 summarizes VL event masks.

**Table 14-2** VL Event Masks

Symbol	Meaning	
VLStreamBusyMask	Stream is locked	
VLStreamPreemptedMask	Stream was grabbed by another application	
VLAdvanceMissedMask	Time was already reached	
VLSyncLostMask	Irregular or interrupted signal	
VLSequenceLostMask	Field or frame dropped	
VLControlChangedMask	A control has changed	
VL Control Range Changed Mask	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 timing change on a Galileo Video node	
VLDefaultSourceMask	Default source changed	

Call **vlGetFD()** to get a file descriptor usable from *select*(2) or *poll*(2).

Call vlSelectEvents() to express interest in one or more event. For example: vlSelectEvents(svr, path, VLTransferCompleteMask);

Event masks can be or'ed together. For example:

Depending on whether you want to block processing or not, use **vlNextEvent()** (blocking) or **vlCheckEvent()** (nonblocking) to get the next event.

Use **vlPeekEvent()** to see what the next event in the queue is without removing it from the queue. For example, the part of the code that actually gets the event from the event loop uses **vlNextEvent()**, whereas another part of the code that just wants to know about it, for example, for priority purposes, uses **vlPeekEvent()**.

### **Creating a VL Event Loop**

You can set an event loop to run until a specific condition is fulfilled. The routine **vlSelectEvents()** allows you to specify which event the application will receive.

Using an event loop requires creating an *event mask* to specify the events you want. The VL event mask symbols are combined with the bitwise OR operator. For example, to set an event mask to express interest in either transfer complete or control changed events, use:

```
VLTransferCompleteMask | VLControlChangedMask
```

To create an event loop, follow these steps:

1. Define the event; for example:

```
VLEvent ev; ...
```

2. Set the event mask; for example:

```
vlSelectEvents(vlServer, path, VLTransferCompleteMask |
VLControlChangedMask)
```

3. Block on the transfer process until at least one event is waiting:

```
for(;;){
vlNextEvent(vlServer, &ev);
```

4. Create the loop and define the choices; for example:

```
switch(ev.reason){
      case VLTransferComplete:
      ...
      break;
    case VLControlChanged:
      ...
      break;
}
```

## **Creating a Main Loop with Callbacks**

**vlMainLoop()** is provided as a convenience routine for the application programmer and constitutes the main loop of VL applications. This routine first reads the next incoming video event; it then dispatches the event to the appropriate registered procedure. Note that the application does not return from this call.

Applications are expected to exit in response to some user action. There is nothing special about **vlMainLoop()**; it is simply an infinite loop that calls the next event and then dispatches it. An application can provide its own version of this loop, for example, to test a global termination flag or to test that the number of top-level widgets is larger than zero before circling back to the call to the next event.

To specify callbacks, that is, routines which are called when a particular VL event arrives, use **vlAddCallback()**. Its function prototype is:

Example 14-1 illustrates the use of vlAddCallback().

#### **Example 14-1** Using VL Callbacks

```
main()
      /* Set up the mask for control changed events and Stream preempted events */
   if (vlSelectEvents(vlSvr, vlPath, VLTransferComplete | VLStreamPreemptedMask))
         doErrorExit("select events");
   /* Set ProcessEvent() as the callback for VL events */
   \verb|vlAddCallback| (\verb|vlSvr|, vlPath|, VLTransferCompleteMask| | VLStreamPreemptedMask|,
                 ProcessEvent, NULL);
   /* Start the data transfer immediately (i.e. don't wait for trigger) */
   if (vlBeginTransfer(vlSvr, vlPath, 0, NULL))
        doErrorExit("begin transfer");
   /* Get and dispatch events */
   vlMainLoop();
}
/* Handle VL events */
ProcessEvent(VLServer svr, VLEvent *ev, void *data)
   switch (ev->reason)
      case VLTransferComplete:
        /* Get the valid video data from that frame */
           dataPtr = vlGetActiveRegion(vlSvr, transferBuf, info);
       /* Done with that frame, free the memory used by it */
             vlPutFree(vlSvr, transferBuf);
             frameCount++;
    break;
    case VLStreamPreempted:
        fprintf(stderr, "%s: Stream was preempted by another Program\n",
         _progname);
          docleanup(1);
    break;
    default:
    break;
}
```

Delete a callback with **vlRemoveCallback()** or **vlRemoveAllCallbacks()**. Their function prototypes are:

The functions **vlAddHandler()** and **vlRemoveHandler()** are analogous to **vlAddCallback()** and **vlRemoveCallback()**, respectively. Use them for non-VL events.

Example 14-2 illustrates how to create a main loop and event loops.

**Caution:** To simplify the code, this example does not check returns. The programmer should, however, always check returns.

#### **Example 14-2** VL Event Handling: *eventex.c*

```
/*===========An Event Driven Application========
* File:
                 eventex.c
* Usage:
                eventex
* Description: event demonstrates VL eventloop with the IRIS GL
* Functions:
                 IRIS Video Library functions used
                 vlOpenVideo()
                 vlGetNode()
                 vlCreatePath()
                 vlSetupPaths()
                 vlSetControl()
                 vlCreateBuffer()
                 vlRegisterBuffer()
                 vlRegisterHandler()
                 vlAddCallback()
                 vlSelectEvents()
                 vlMainLoop()
                 vlGetActiveRegion()
                 vlGetNextValid()
                 vlPutFree()
```

```
vlBeginTransfer()
                  vlEndTransfer()
                  vlDeregisterBuffer()
                  vlDestroyPath()
                  vlDestroyBuffer()
                  vlCloseVideo()
                  vlPerror()
 * /
#include <stdlib.h>
#include <stdio.h>
#include <gl/gl.h>
#include <gl/device.h>
#include <dmedia/vl.h>
* Function Prototypes
 * /
void error_exit(void);
void ProcessEvent(VLServer svr, VLEvent *ev, void *data);
void ProcessGlEvent(int fd, void *win);
void exit_capture(void);
 * Global Variables
* /
char *_progName;
VLBuffer buffer;
VLServer svr;
VLPath path;
VLNode src, drn;
int xsize;
int ysize;
/* Report errors */
void
error_exit(void)
    vlPerror(_progName);
    exit(1);
```

```
void
main(int argc, char **argv)
    VLControlValue val;
    int c;
    long win;
    _progName = argv[0];
    foreground();
    /* Connect to the daemon */
    if (!(svr = vlOpenVideo("")))
        error_exit();
    /* Set up a drain node in memory */
    drn = vlGetNode(svr, VL_DRN, VL_MEM, VL_ANY);
    /* Set up a source node on any video source */
    src = vlGetNode(svr, VL_SRC, VL_VIDEO, VL_ANY);
    /* Create a path using the first device that will support it */
    path = vlCreatePath(svr, VL_ANY, src, drn);
    /* Set up the hardware for and define the usage of the path */
    if ((vlSetupPaths(svr, (VLPathList)&path, 1, VL_SHARE,
        VL\_SHARE)) < 0)
        error_exit();
    /* Set the packing to RGB */
    val.intVal = VL_PACKING_RGB_8;
    vlSetControl(svr, path, drn, VL_PACKING, &val);
    /* Get the video size */
    vlGetControl(svr, path, drn, VL_SIZE, &val);
    xsize = val.xyVal.x;
    ysize = val.xyVal.y;
    /* Set up and open a GL window to display the data */
    prefsize(xsize,ysize);
    win = winopen("Eventex Window");
    RGBmode();
    pixmode(PM_TTOB, 1);
    gconfig();
```

```
* Allow these key presses, mouseclicks, etc to be
 * entered in the event queue
qdevice(ESCKEY);
qdevice(WINSHUT);
qdevice(WINQUIT);
/* Create and register a buffer for 1 frame */
buffer = vlCreateBuffer(svr, path, drn, 1);
if (buffer == NULL)
    error_exit();
vlRegisterBuffer(svr, path, drn, buffer);
/* Begin the data transfer */
if (vlBeginTransfer(svr, path, 0, NULL))
    error_exit();
* Specify what path-related events we want to receive.
* In this example we only want transfer complete events.
* /
vlSelectEvents(svr, path, VLTransferCompleteMask);
/* Set ProcessEvent() is the callback for transfer complete */
vlAddCallback(svr, path, VLTransferCompleteMask, ProcessEvent,
              NULL);
/* Set ProcessGlEvent() as the GL event handler */
vlRegisterHandler(svr, qgetfd(), (VLEventHandler)ProcessGlEvent,
                  (VLPendingFunc) qtest, (void *)win);
/* Loop and dispatch events */
vlMainLoop();
```

```
/* Handle video library events */
void ProcessEvent(VLServer svr, VLEvent *ev, void *data)
    VLInfoPtr info;
    char *dataPtr;
    switch (ev->reason)
        case VLTransferComplete:
           info = vlGetNextValid(svr, buffer);
            if(!info)
                break;
            /* Get a pointer to the frame */
            dataPtr = vlGetActiveRegion(svr, buffer, info);
            /* Write the data to the screen */
            lrectwrite(0,0, xsize-1, ysize-1, (ulong *)dataPtr);
            /* Finished with frame, unlock the buffer */
            vlPutFree(svr, buffer);
        break;
        default:
            printf("Got Event %d\n", ev->reason);
        break;
    }
```

```
/* Handle graphics library events */
void ProcessGlEvent(int fd, void *win)
    static short val;
    switch (gread(&val))
        /* Quit */
        case ESCKEY:
            if (val == 1) /* Respond to keydowns only */
                exit_capture();
        break;
        case WINSHUT:
        case WINQUIT:
            exit_capture();
        break;
        default:
        break;
    }
}
void
exit_capture()
    /* End the data transfer */
    vlEndTransfer(svr, path);
    /* Disassociate the ring buffer from the path */
    vlDeregisterBuffer(svr, path, drn, buffer);
    /* Destroy the path, free the memory it used */
    vlDestroyPath(svr,path);
    /* Destroy the ring buffer, free the memory it used */
   vlDestroyBuffer(svr, buffer);
    /* Disconnect from the daemon */
    vlCloseVideo(svr);
    exit(0);
```

# Chapter 15

**VL** Blending

This chapter explains how to use the VL to perform blending operations.

# **VL** Blending

This chapter explains how to combine video frame information and computer-generated graphics, if your equipment supports such operations.

Use the VL to perform three types of blending:

- Chroma keying: overlaying one image on another by choosing a key color. For example, if chroma keying is set to blue, image A might show through image B everywhere the color blue appears in image B. A common example is the TV weather reporter standing in front of the satellite weather map. The weather reporter, wearing any color but blue, stands in front of a blue background; keying on blue shows the satellite picture everywhere blue appears. Because there is no blue on the weatherperson, he or she appears to be standing in front of the weather map.
- Luma keying: overlaying one image on another by choosing a level of luminance. For example, to overlay bright text (such as a caption) on video, a graphics source is created with the text on a dark background. The video source is made to show through the dark areas of the graphics; the bright text remains on top of the video.
- Transitions: fades, tiles, and wipes, such as single, double, or corner wipes, for which you can set the angle or center.

This chapter explains how the VL performs blending. Topics in this chapter include:

- The VL Key Generator
- The VL Blend Node
- VL Blending Controls
- VL Keying

The chapter concludes with example application programs.

## The VL Key Generator

Blending in the VL is based on values that the *key generator* assigns to each pixel in the sources to be blended. The key generator generates a key for each pixel in the two source nodes (foreground and background):

- If luma keying is set, the key generator assesses the brightness of each pixel.
- If chroma keying is set, the key generator assesses the color of each pixel.
- If spatial, or transition, keying (fade, tile, wipe) is set, the key generator assesses the (x,y) coordinates for each pixel.

In addition, the key generator determines the alpha value (opacity) of a pixel and sets a value for it ranging from 0 (completely transparent) to 1 (completely opaque). This alpha value can be used downstream for further layering operations.

Key generation is usually for one alpha source only. The other alpha source can be constant or can come from a matte signal or from graphics on systems that support alpha rendering.

### The VL Blend Node

Blending takes place in the VL's internal *blend node*, which is created with the **vlGetNode()** function.

**Note:** Not all connections are possible on all video options.

The code fragment in Example 15-1 sets up source, drain and blend nodes.

**Example 15-1** Setting Up Source, Drain, and Blend Nodes

```
/* variable definitions */
VLServer svr;
VLPath path;
VLNode src, drn;
VLControlValue val;
int x, y, c;
uint w, h, bw{
/* Open a video device */
svr = vlOpenVideo("")
/* Set up drain nodes on the screen and video */
drn_scr = vlGetNode(vlSvr, VL_DRN, VL_SCREEN, VL_ANY);
drn_vid = vlGetNode(vlSvr, VL_DRN, VL_VIDEO, VL_ANY);
/* Set up source nodes on the screen and video */
src_scr = vlGetNode(vlSvr, VL_SRC, VL_SCREEN, VL_ANY);
src_vid = vlGetNode(vlSvr, VL_SRC, VL_VIDEO, vin);
/* Set up internal blending node */
blend_node = vlGetNode(vlSvr, VL_INTERNAL, VL_BLENDER,
                       VL_ANY);
```

Figure 15-1 diagrams the blender setup.

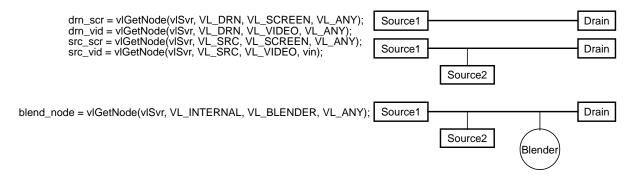


Figure 15-1 Setting Up the Blend Node

The blend node mixes the foreground and background video signals by applying a blend function to the foreground and background pixels.

Figure 15-2 diagrams the Galileo Video alpha blender.

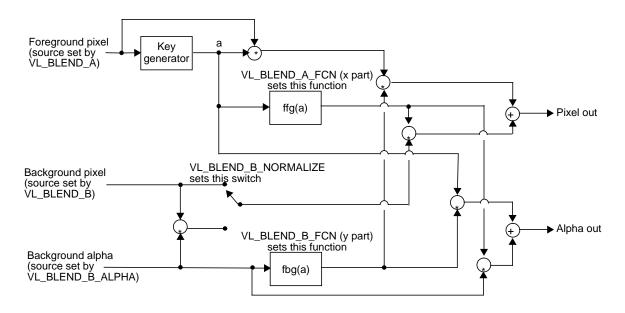
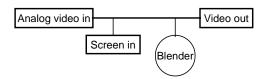


Figure 15-2 Galileo Video Alpha Blender

The blend node then sends the data to the drain node. For example, blending analog video with part of a graphics screen and sending it to video out can be diagrammed as shown in Figure 15-3.



**Figure 15-3** Blending Analog Video with Part of a Graphics Screen

Blending analog video with static frame data and sending it to video out can be diagrammed as shown in Figure 15-4.

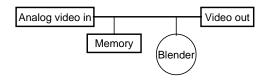


Figure 15-4 Blending Analog Video with Static Frame Data

Adding another drain, such as a screen location at which to preview the output, can be diagrammed as shown in Figure 15-5.

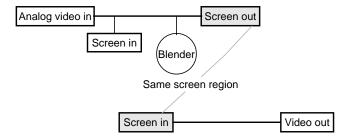


Figure 15-5 Adding Another Drain to Preview the Blend

In Figure 15-5, the source called Screen in and the drain called Screen out are shaded to indicate that although they are separate and distinct nodes, they overlap physically; that is, they are set for the same screen location.

## **VL Blending Controls**

The VL uses blending controls to set blending options.

All blending controls—that is, all the controls discussed in this chapter—apply only to blend nodes, except for VL\_EV1\_ALPHA\_NOT\_PIXEL, which applies to drain nodes. The order of blending and zooming depends on the node type: for a source, zooming takes place before blending; for a drain, blending takes place before zooming.

All controls are available for all platforms unless otherwise noted. See the *Sirius Video Owner's and Programming Guide* for Sirius Video blending controls.

**Note:** The reference "Galileo Video" includes Indigo<sup>2</sup> Video and Indy Video, unless otherwise noted.

Table 15-1 summarizes the VL controls for blending.

**Table 15-1** VL Blend Controls

Control	Values	Selects
VL_BLEND_A_FCN type intVal	VL_BLDFCN_ZERO VL_BLDFCN_ONE VL_BLDFCN_B_ALPHA (background alpha)/255 VL_BLDFCN_MINUS_B_ALPHA: 1 - ((background alpha) / 255)	Blend function that controls mixing of foreground signals
VL_BLEND_B_FCN type <i>intVal</i>	VL_BLDFCN_ZERO VL_BLDFCN_ONE VL_BLDFCN_A_ALPHA (foreground alpha)/255 VL_BLDFCN_MINUS_A_ALPHA 1 - ((foreground alpha) / 255)	Blend function that controls mixing of background signals
VL_BLEND_A type intVal	VLNode type, derived from vlGetNode(); must be one of the two source nodes	Input source for foreground image
VL_BLEND_B type intVal	VLNode type, derived from vlGetNode(); must be one of the two source nodes	Input source for background image

	Table 15-1 (continued)	VL Blend Controls
Control	Values	Selects
VL_BLEND_A_ALPHA type intVal	VLNode type, derived from vlGetNode(); must be one of the two source nodes	Input source for foreground alpha; cannot be used on Galileo Video, Indigo <sup>2</sup> Video, or Indy Video
VL_BLEND_B_ALPHA type intVal	VLNode type, derived from vlGetNode(); must be one of the two source nodes	Input source for background alpha
VL_BLEND_A_NORMALIZE type boolVal	(0,1) 0 = off, 1 = on	Follows Porter-Duff model (background pixels premultiplied by their corresponding alphas before blending)
VL_BLEND_B_NORMALIZE type boolVal	(0,1) 0 = off, 1 = on	Follows Porter-Duff model
VL_BLEND_OUT_NORMALIZE type boolVal	E(0,1) 0 = off, 1 = on	Follows Porter-Duff model

## VL Keying

For each kind of keying—luma keying, chroma keying, and transitions—further VL controls enable you to specify the properties of the blend.

**Note:** Keying parameters are implemented as device-dependent VL controls; this section explains Galileo Video (including Indigo<sup>2</sup> Video and Indy Video) keying controls.

The values for the Galileo Video "master" keyer control VL\_EV1\_KEYER\_MODE determine the type of keying performed:

- luma keying: VL\_EV1\_KEYERMODE\_LUMA
- chroma keying: VL\_EV1\_KEYERMODE\_CHROMA
- transitions, that is, fades, tiles, or wipes: VL\_EV1\_KEYERMODE\_SPATIAL

For example, the following fragment specifies that chroma keying is to be performed:

Keying controls fall into three groups:

- luma keying
- chroma keying
- fades, tiles, and wipes

Each type is explained separately in this section.

# **Galileo Video Luma Keying**

Luma keying is typically used to overlay a fixed image on video, such as the name and title of an individual being interviewed, a cable channel's logo, or a symbol that denotes an ongoing news story during a newscast. Figure 15-6 diagrams an application.

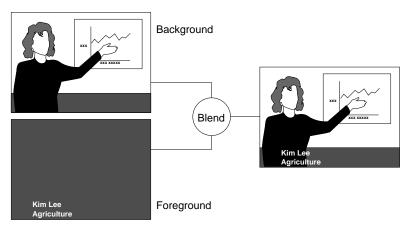


Figure 15-6 Luma Keying Application: Titling

The four Galileo Video luma keying controls are summarized in Table 15-2; each is of type **intVal**.

**Table 15-2** Galileo Video Luma Keying Controls

Control	Range	Sets
VL_EV1_KEYER_VALUE_LUMA	(0,255)	Central luma value. This control sets the luma value at which the background shows through the foreground.
VL_EV1_KEYER_RANGE_LUMA	A (0,255)	One-sided range of the center value. This control determines the range of luma values where the background shows through the foreground.
VL_EV1_KEYER_FG_OPACITY	(0,255)	Opacity of the foreground, thus limiting the value of foreground alpha at any point.
VL_EV1_KEYER_DETAIL	(-8,7)	Sharpness of transition between foreground and background allowing blurring of edges. The value -8 yields the most gradual transition, +7 the sharpest.

Figure 15-7 diagrams the relationships between these controls.

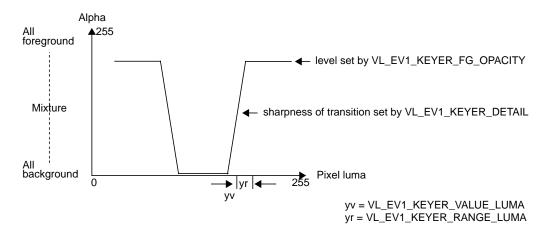


Figure 15-7 Relationships Between Galileo Video Luma Keying Controls

# **Galileo Video Chroma Keying**

Chroma keying overlays one image on another based on the color value. Figure 15-8 diagrams a common application.

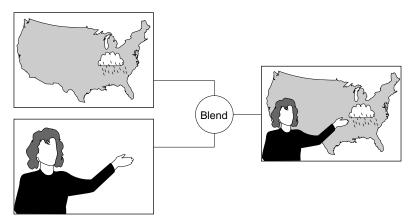


Figure 15-8 Chroma Keying Application: TV Weather Map

Table 15-3 summarizes the controls for Galileo Video chroma keying and gives their ranges. These controls are all of type **intVal**.

**Table 15-3** Galileo Video Chroma Keying Controls

Control	Range	Sets
VL_EV1_KEYER_VALUE_CHROMA_U	(-226,226)	Central U value at which the background shows through the foreground.
VL_EV1_KEYER_RANGE_CHROMA_U	(0,452)	One-sided range of U where the background shows through the foreground.
VL_EV1_KEYER_VALUE_CHROMA_V	(-179,179)	Central V value at which the background shows through the foreground.
VL_EV1_KEYER_RANGE_CHROMA_V	(0,358)	One-sided range of V where the background shows through the foreground.
VL_EV1_KEYER_DETAIL	(-8,7)	Sharpness of transition between foreground and background

**Note:** VL\_EV1\_KEYER\_FG\_OPACITY has no effect on Galileo Video in chroma key mode.

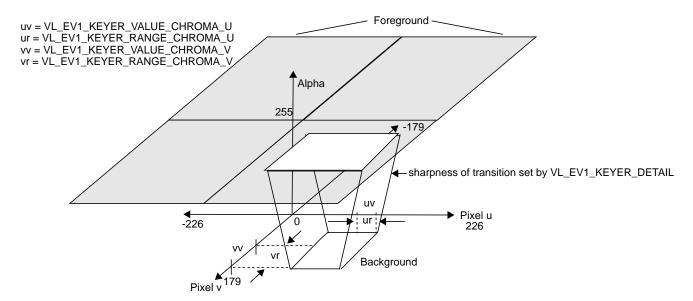


Figure 15-9 diagrams the relationships between these controls.

Figure 15-9 Relationships Between Galileo Video Chroma Keying Controls

## Galileo Video Fades, Tiles, and Wipes

The values used with the control VL\_EV1\_WIPE\_TYPE determine the type of blending performed:

- from all-foreground to all-background: VL\_EV1\_WIPETYPE\_FADE
- from all-foreground to all-background by randomly tiling screen with rectangles of a specified size: VL\_EV1\_WIPETYPE\_TILE
- wipe to cross the screen as a vertical, diagonal, or horizontal "front," with a specified angle: VL\_EV1\_WIPETYPE\_SINGLE
- wipe in two orthogonal directions simultaneously (two single wipes at the same time): VL\_EV1\_WIPETYPE\_DOUBLE
- wipe in two orthogonal directions, with the perpendicular position locked to the normal, or in-line position: VL\_EV1\_WIPETYPE\_CORNER

For example, the following fragment specifies that a fade is to be performed:

Fades, tiles, and wipes go from all-foreground (VL\_EV1\_WIPE\_POSN=0) to all-background (VL\_EV1\_WIPE\_POSN=1000), unless VL\_EV1\_WIPE\_INVERT control is set, in which case they go from all-background (VL\_EV1\_WIPE\_POSN = 0) to all-foreground (VL\_EV1\_WIPE\_POSN = 1000).

Table 15-4 summarizes controls common to all wipe types.

**Table 15-4** Controls for Fades, Tiles, and Wipes

Control	Values	Sets
VL_EV1_WIPE_POSN type fractVal	numerator (0,1000) denominator (1000)	Amount of progress of wipe, from none (numerator = 0) to full (numerator = 1000).
VL_EV1_WIPE_REPT type <i>intVal</i>	(0,15)	Number of repetitions of pattern in direction of wipe, usually louvers on single, corner, or double wipe, and length of one side of rectangles for a tile wipe.  Note: This control does not apply to fades.
VL_EV1_WIPE_INVERT type intVal	(0,1) 0 = off, 1 = on	Reversal of foreground and background regions of a wipe. When set to 0, wipes proceed from foreground (position = minimum) to background (position = maximum). When set to 1, wipes proceed from background (position = minimum) to foreground (position = maximum).  This value is buffered (does not go into effect) until another blending control is set.

Table 15-5 summarizes the controls specific to wipes or that work differently for wipes. Some of these controls work in conjunction with each other.

 Table 15-5
 Galileo Video Controls Specific to Wipes

Control	Values	Sets
VL_EV1_WIPE_ANGLE type intVal	VL_EV1_WIPEANGLE_E VL_EV1_WIPEANGLE_NE VL_EV1_WIPEANGLE_N VL_EV1_WIPEANGLE_W VL_EV1_WIPEANGLE_SW VL_EV1_WIPEANGLE_S VL_EV1_WIPEANGLE_S VL_EV1_WIPEANGLE_SE	Wipe vector direction, that is, the direction the wipe appears to be proceeding in as its position increases.  Note: VL_EV1_WIPEANGLE_N and VL_EV1_WIPEANGLE_S do not work for the wipe types VL_EV1_WIPETYPE_DOUBLE and VL_EV1_WIPETYPE_CORNER on Galileo Video.
VL_EV1_WIPE_FUZZ type intVal	(-8,7)	Sharpness of wipe transition band. As for VL_EV1_KEYER_DETAIL, -8 is most gradual, +7 is sharpest.
VL_EV1_WIPE_SYMMETRY type <i>intVal</i>	(0,1) 0 = off, 1 = on	Wipe symmetry (on or off) so that wipe proceeds in both directions at once from the center line. Effect depends on type of wipe: no effect for fades or tiling; enables VL_EV1_WIPE_CENT for single, double, and corner wipes; enables VL_EV1_WIPE_CENT_PERP control for double and corner wipes.
VL_EV1_WIPE_POSN_PERP type fractVal	numerator (0,1000) denominator (1000)	Amount of progress of wipe, from none (numerator = 0) to full (numerator = 1000), along a direction perpendicular to normal wipe position VL_EV1_WIPE_POSN.
VL_EV1_WIPE_CENT type fractVal	numerator (0,1000) denominator (1000)	Offset that is center of a symmetrical wipe along wipe position. 0 means center is where VL_EV1_WIPE_POSN is 0, and 1000 means center is where VL_EV1_WIPE_POSN is 1000. For this control to work for single, double, and corner wipes, VL_EV1_WIPE_SYMMETRY must be on.
VL_EV1_WIPE_CENT_PERP type fractVal	numerator (0,1000) denominator (1000)	Offset that is center of a symmetrical wipe along a perpendicular wipe position. 0 means center is where VL_WIPE_POSN_PERP is 0, and 1000 means center is where VL_WIPE_POSN_PERP is 1000. VL_WIPE_SYMMETRY must be on for this control to work for double and corner wipes.
VL_EV1_WIPE_REPT_PERP type intVal	(0,15)	Number of repetitions perpendicular to wipe direction for single, double, and corner wipes, and length of other side of rectangles for tile wipe.

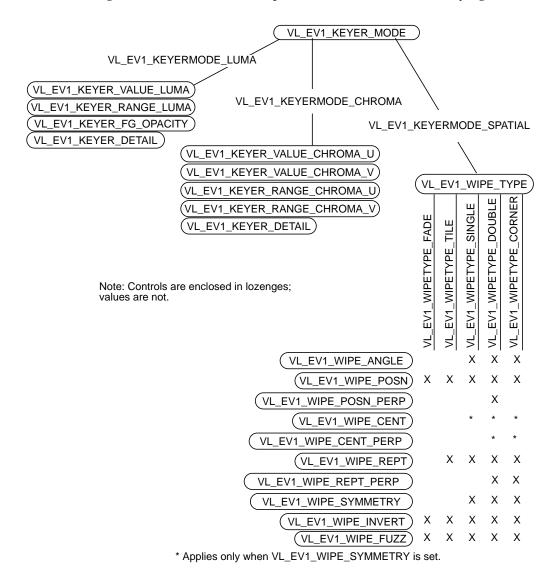


Figure 15-10 shows relationships between the Galileo Video keying controls.

**Figure 15-10** Galileo Video Keying Controls

## **VL Blending Examples**

This section explains two example programs from /usr/people/4Dgifts/examples/dmedia/video/vl:

- simpleblend.c
- simplewipe.c

Because the programs are lengthy, they are not duplicated here. Look at the source code in a separate window, or print them out to look at while you read their descriptions.

**Caution:** To simplify the code, these examples do not check returns. The programmer should, however, always check returns.

## **Blending Video and Graphics**

*simpleblend.c,* which blends video with graphics and outputs it to both a graphics window and video out. The program:

- constrains the window's aspect ratio
- checks that the device the user requested is in the device list
- sets up a path between the source (screen) and the drain (video)
- adds video source and a screen drain nodes to create the blend
- sets the keyer mode, keyer source, and blend controls
- displays the drain window and sets the video to appear in it
- specifies appropriate event handling
- starts data transfer
- specifies that video is updated if the user changes the size of the window

## **Creating a Simple Wipe Effect**

Like *simpleblend.c*, *simplewipe.c* blends video with graphics and outputs it to a graphics window and video out. When the user presses the  $\mathbf{w}$  key, it executes a wipe.

Specifically, in addition to doing everything that *simpleblend.c* does, *simplewipe.c*:

- sets up blend parameters (VL\_WIPE\_TYPE, VL\_WIPE\_ANGLE, VL\_WIPE\_CENT, VL\_WIPE\_REPT)
- calls a loop that sets the keyer mode to spatial and sets the position in the loop; **doswitchloop()** and **dowipe()** execute the loop
- checks for the w key and calls **dowipe()**, which in turn calls **doswitchloop()**

#### **PART FOUR**

# IndigoVideo Programming

Chapter 16, "Introduction to IndigoVideo Programming," introduces the IndigoVideo library and gives an overview of the features of the IndigoVideo board.

Chapter 17, "Getting Started with the IndigoVideo Library," describes basic concepts for using the IndigoVideo board, and presents a sample video application that displays live video input in a window.

Chapter 18, "Controlling the IndigoVideo Input Window," explains how applications can position and scale the video input. It also explains how to select different video sources, formats, and broadcast standards.

Chapter 19, "Producing IndigoVideo Output," explains how to encode a portion of your screen to video in real time. This chapter also covers single-frame output.

Chapter 20, "Capturing Video from IndigoVideo," explains how to capture frames of video to memory.

Chapter 21, "Handling IndigoVideo Events," explains how to handle video events, such as video parameters being changed by another process.

Chapter 22, "Using the IndigoVideo Utilities," explains how to use the IndigoVideo end-user tools.

# Chapter 16

# Introduction to IndigoVideo Programming

This chapter introduces the IndigoVideo Library, which is the software interface to the IndigoVideo board for Indigo Entry graphics workstations.

# Introduction to IndigoVideo Programming

The IndigoVideo board provides video input and output for IRIS Indigo workstations equipped with Entry Graphics. The IndigoVideo Library provides a software interface to the IndigoVideo board, enabling applications to:

- display live video in a window
- capture live video to system memory
- encode graphics to video in real time
- produce high-quality single-frame video output

The IndigoVideo Library provides a C language API; this part of this guide describes the use of those routines. The IndigoVideo library header file, /usr/include/svideo.h, is compatible with the ANSI-C standard; however, the IndigoVideo library does not comply with ANSI-C namespace conventions.

For an introduction to basic video concepts, read Chapter 11, "Video Basics," in Part III, "Video Programming," of this guide, and consult the Glossary at the end of this guide for definitions of video terms.

Part IV, "IndigoVideo Programming," presents the IndigoVideo library from a task-oriented perspective. Chapters are organized to cover topics in roughly the order you would be concerned about them as you write IndigoVideo programs.

# Using the IndigoVideo Examples

The code examples in this part of this guide are online in the directory /usr/people/4Dgifts/examples/dmedia/video/indigovideo. The README file in that directory gives an overview of the programs and instructions for compiling and running them. You must use the 4Dgifts login to compile and run these programs.

# **References for Video Programming**

For more information on video, consult these references:

- Television Engineering Handbook, Benson, K. Blair, McGraw-Hill (New York) 1986.
- *Television Technology: Fundamentals and Future Prospects*, Noll, A. Michael, Artech House (MA) 1988.
- Lenk's Video Handbook: Operation and Troubleshooting, Lenk, John D., McGraw-Hill (New York), 1991.
- Basic Television and Video Systems, Fifth Edition, Grob, Bernard, McGraw-Hill (New York), 1984.

# Chapter 17

# Getting Started with the IndigoVideo Library

This chapter explains basic concepts for programming with the IndigoVideo Library, including setting up video I/O through the IndigoVideo board.

# Getting Started with the IndigoVideo Library

This chapter describes the features and capabilities of the IndigoVideo board and presents an annotated sample program that displays live video input in a graphics window to help you get started with the IndigoVideo Library.

#### In this chapter:

- "IndigoVideo Basics" on page 385 describes the features and I/O functions of the IndigoVideo board.
- "A Simple Program for Getting Started with IndigoVideo" on page 392
  presents a simple IndigoVideo application that demonstrates the use of
  the most basic IndigoVideo Library routines.

For an introduction to basic video concepts, read Chapter 11, "Video Basics," in Part III, "Video Programming," and consult the Glossary at the end of this guide for definitions of video terms.

# IndigoVideo Basics

This section describes the IndigoVideo board and its I/O interface.

### IndigoVideo

The IndigoVideo board attaches to the Indigo Entry Graphics board in your Indigo workstation. On the back of the board is an I/O panel with a number of video connectors, which you can use to attach video devices to the IndigoVideo board. The IndigoVideo board translates video signals into a form usable by the IRIS Indigo workstation. It also does the reverse, translating graphics from the IRIS Indigo display into video signals.

#### **Broadcast Standards**

The IndigoVideo hardware supports the two most popular broadcast standards, the National Television Systems Committee (NTSC) composite video standard and the Phase Alternated by Line (PAL) standard.

#### Video and Videotape Formats

You can record video signals in a variety of videotape formats: S-VHS and Hi-8mm are two examples of common videotape formats. Although IndigoVideo doesn't distinguish between individual tape formats, you need to know what kind of connector your video equipment uses. The IndigoVideo board has two kinds of input connectors: *composite* and *S-Video*.

Most home VCRs use composite connectors. S-Video, on the other hand, carries the color and brightness components of the picture on separate wires; hence, S-Video connectors are also called Y/C connectors. Most S-VHS and Hi-8mm VCRs feature S-Video connectors.

In addition to composite and S-Video output, the IndigoVideo board provides analog RGB output, in which the image data is carried as three separate components: red, green, and blue intensity. The RGB output can be used in conjunction with an external encoder to produce formats that IndigoVideo does not support, or it can be used to drive an external video monitor.

See "Videotape Formats" in Chapter 11 for more information about videotape formats, which are summarized in Table 11-1 in that chapter.

#### IndigoVideo Data Formats

This section describes the image data formats used by IndigoVideo. IndigoVideo uses three formats for image data: 32-bit RGB, 8-bit RGB, and 4:1:1 YUV. You should note that data coming from or going to the IndigoVideo board is always ordered top-to-bottom. The IRIS Indigo, on the other hand, commonly stores image data with lines ordered bottom-to-top. Both IRIS Indigo and IndigoVideo store pixels from left to right within lines.

In the diagrams that follow, the bits are numbered from right to left, with the least significant (rightmost) bit numbered zero.

#### 32-bit RGB

IndigoVideo uses 32-bit RGB format for single frame output, and it is produced by some of the video capture convenience routines. This format could also be called 24-bit RGB data, since each 32-bit pixel consists of 24 bits of RGB data and 8 bits of unused space.

The format of these pixels is shown in Figure 17-1.

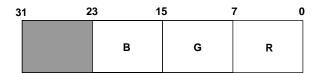
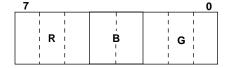


Figure 17-1 Format of 32-bit RGB Pixels

32-bit RGB is produced by the functions **svCaptureOneFrame()**, **svYUVtoRGB()**, and **svRGB8toRGB32()**. The IRIS GL **lrectwrite()** function also uses this format, but it expects rows of pixels to be ordered bottom-to-top unless the default ordering has been changed using the IRIS GL **pixmode()** function. As noted above, data sent to the IndigoVideo board using the **svPut24Frame()** function should be ordered top-to-bottom. The **svYUVtoRGB()** and **svRGB8toRGB32()** functions will return data ordered top-to-bottom unless the *invert* parameter is TRUE.

#### 8-bit RGB

The 8-bit RGB format stores 2 bits of blue and 3 bits each of red and green in an 8-bit pixel. The format of the 8-bit RGB pixels is shown in Figure 17-2.



**Figure 17-2** Format of 8-bit RGB Pixels

When you capture 8-bit RGB frames using IndigoVideo, you receive them with the fields uninterleaved, that is, the data contains all of the even lines (from top to bottom) followed by all of the odd lines.

#### 4:1:1 YUV

The 4:1:1 YUV format is much closer in form to the original video signal than the RGB formats described above. The YUV format is also much more complicated. Each pixel is described by three components: one luminance value (Y) and a pair of chrominance values (U and V); however, there are four luminance samples to each pair of chrominance samples has (hence 4:1:1 YUV). This means that sets of 4 pixels share the same chrominance values but have individual luminance values.

To further complicate matters, data from odd and even lines are interleaved within individual 32-bit words. The format of these words is shown in Figure 17-3.

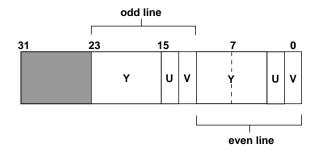


Figure 17-3 Format of YUV Data Words

Each 32-bit word consists of 8 bits of unused space, followed by one Y sample (8 bits) and 2 bits each of U and V from the odd line, then a Y sample and 2 bits each of U and V from the even line.

By collecting the chrominance data from 4 consecutive 32-bit words, you get a pair of chrominance samples for the odd line, and a pair of chrominance samples for the even line. The upper 2 bits of U and V come in the first word, the next highest set of bits comes in the next word, and so on. By putting these chrominance samples together with the 4 odd and 4 even luminance samples, you can reconstruct 8 YUV pixels. You can derive 8-bit red, green, and blue values from the YUV values using the following formulas:

$$R = 1.164 \times (Y - 16) + 1.596 \times (V - 128)$$

$$G = 1.164 \times (Y - 16) - 0.392 \times (U - 128) - 0.813 \times (V - 128)$$

$$B = 1.164 \times (Y - 16) + 2.017 \times (U - 128)$$

As with the other forms of video data, lines of YUV data are ordered top-to-bottom.

## IndigoVideo I/O

Figure 17-4 shows the IndigoVideo I/O panel, which is accessible from the back of the Indigo workstation.

### **IndigoVideo Board Input Port Specifications**

At the top of the panel are three pairs of input connectors, labeled 1, 2, and 3. Each pair is made up of a composite video connector and an S-Video connector. You can use only one of the connectors in a given pair at a time—if you connect a composite video source and an S-Video source to the same input pair, the IndigoVideo hardware will not be able to decode either source properly.

The three input port pairs on the IndigoVideo board have the following specifications:

- Composite VIDEO: 1.0 Vpp, 75, sync negative
- S-VIDEO: DIN 4-pin

Below the input connectors, there are three output connectors—one composite connector, one S-Video connector, and one RGB connector.

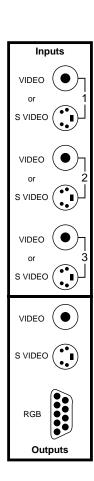
### **IndigoVideo Board Output Port Specifications**

The three output ports on IndigoVideo board have the following specifications:

- Composite VIDEO output: 1.0 Vpp, 75, sync negative
- S-VIDEO output: DIN 4-pin
- RGB output: 15 pin D-connector, .714 Vpp, 75

The sync specification is:

• Sync: RS-170 television sync: –4Vpp @ 75



**Figure 17-4** IndigoVideo I/O Ports

Figure 17-5 shows video equipment connected to the I/O panel of the Indigo Video board.

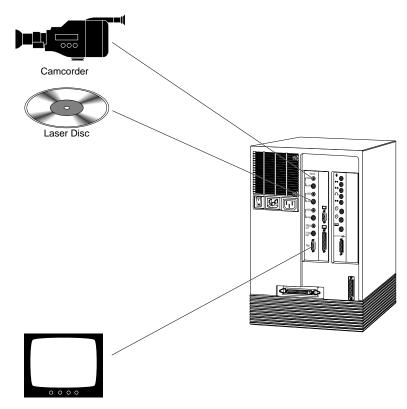


Figure 17-5 Connecting Video Equipment to the Indigo Video Board

The next two sections briefly describe how the IndigoVideo board operates.

### **Video Input**

Five basic steps are used to convert a video input signal to pixels that can be displayed on the Indigo workstation:

- 1. The IndigoVideo board synchronizes with the video source by looking for sync information in the video signal (see Figure 11-3).
- 2. The analog video signal is converted to a digital signal.

3. The digital signal is decoded into a series of pixels in 4:1:1 YUV *colorspace*.

A colorspace is simply a way of encoding color information. In RGB colorspace, a color is defined by its red, green, and blue levels. In YUV colorspace, a color is defined by its luminance and by two chrominance components that determine its color. 4:1:1 means that the chrominance components are *subsampled*; that is, for each four Y samples, there is only one U sample and one V sample.

- 4. The YUV pixels are converted to 24-bit RGB pixels.
- 5. The 24-bit pixels are dithered down to 8-bit pixels.

Before they can be displayed on the Indigo workstation, the video pixels must be reduced to 8-bit RGB pixels. IndigoVideo accomplishes this by dithering the video image. Dithering uses a small number of colors to simulate a larger number of colors using patterns of different colored pixels; for example, a pattern of red and yellow pixels can be used to represent orange.

6. The video image is scaled down, if necessary.

#### **Video Output**

In live video output mode, IndigoVideo receives 8-bit RGB pixels from the Indigo graphics subsystem. These 8-bit pixels are converted to 24-bit RGB pixels using a set of look-up tables. These 24-bit pixels drive a set of digital-to-analog converters, which produce the analog RGB output, and also a digital encoder, which produces composite and S-Video output. The digital encoder also generates a composite sync signal, which can be used to synchronize video devices with the IndigoVideo board. The timing for this signal can be generated by the IndigoVideo board itself, or it can be derived from either the input source or from a separate composite sync signal connected to input #3.

## A Simple Program for Getting Started with IndigoVideo

This section presents a sample program to help you get started with writing a simple video application. Source code for the sample programs is located in the <code>/usr/people/4Dgifts/examples/dmedia/video/indigovideo</code> directory, which is also referred to as <code>4Dgifts</code>. You must have the <code>svideodev</code> option installed to get the gifts source, <code>svideodev.sw.gifts</code>. You also must have the IRIS Development Option, <code>dev</code>, and the C language software, <code>c</code>, loaded before you can compile the sample programs (use the <code>versions</code> command to find out which software is loaded on your system). See the <code>Svideo Release Notes</code> for complete system software requirements. Login as <code>4Dgifts</code> to compile the examples and copy files to your home directory before modifying them.

Example 17-1 contains a listing of *simpleinput.c*, a program that opens up a window and displays live video input. This process has five basic steps:

- 1. Create a GL/X window.
- 2. Open the video device.
- 3. Set video parameters.
- 4. Associate video input with the GL/X window.
- 5. Wait for the user to quit.

To compile *simpleinput.c*, enter:

```
cc -o simpleinput simpleinput.c -lsvideo -lXext -lgl_s
```

You must link with the IndigoVideo Library (**-lsvideo**) to use the IndigoVideo software. The linking order is specific: **-lsvideo** must appear first, followed by **-lXext** to link with the X extensions Library.

Programs that use IRIS GL windowing and event handling must include *gl/gl.h* and *gl/device.h* and must link with the shared IRIS GL (**-lgl\_s**). Programs that use that use the X Window System must include *X11/Xlib.h* before including *svideo.h*, and must link with the X11 shared library (**-lX11\_s**).

**Note:** This program uses IRIS GL windowing and event handling; however, a mixed-model GL/X window and X11 event handling is recommended for greater portability.

**Note:** CPU-intensive programs that use IRIS GL windowing and event handling should do a **qtest()** followed by **sginap()** if no events are waiting.

**Example 17-1** Opening a Window to Display Live Video Input: *simpleinput,c* 

```
#include <stdlib.h>
#include <svideo.h>
#include <gl/gl.h>
#include <gl/device.h>
main()
    long win, dev, params[2];
    short val;
    SVhandle V;
    /* Step 1: Open window */
    prefsize(SV_NTSC_XMAX, SV_NTSC_YMAX);
    win = winopen("video test");
    /* Step 2: Open video device */
    if ((V = svOpenVideo()) == NULL) {
        svPerror("open");
        exit(1);
    }
    /* Step 3: Set video source */
    params[0] = SV_SOURCE;
    params[1] = SV_SOURCE1;
    if (svSetParam(V, params, 2) < 0) {</pre>
        svCloseVideo(V);
        svPerror("set param");
        exit(1);
    }
    /* Step 4: Associate video input with window */
    if (svBindGLWindow(V, win, SV_IN_REPLACE) < 0) {</pre>
        svPerror("bind gl window");
        svCloseVideo(V);
        exit(1);
    }
```

```
/* Step 5: wait for user to quit */
   qdevice(ESCKEY);
   qdevice(WINQUIT);
   gdevice(WINSHUT);
   while (1) {
        dev = qread(&val);
        switch (dev) {
        case ESCKEY:
           if (val)
                       /* exit on key up */
               break;
        case WINQUIT:
        case WINSHUT:
            svCloseVideo(V);
           exit(0);
            break;
    }
}
```

This program isn't very flexible—it assumes that a composite NTSC video source is connected to input #1—however, it demonstrates the basic principles of programming with the IndigoVideo Library. Here's a step-by-step description of what the program does:

- 1. The program opens a window in which to display the video. First it specifies the preferred size for the window using prefsize(), then it calls winopen() to create the window. The winopen() routine returns a window identifier, which is used in step 4 to associate the video input with the window. The symbolic constants SV\_NTSC\_XMAX and SV\_NTSC\_YMAX represent the dimensions of an NTSC video frame.
- The program opens the video device. The svOpenVideo() routine returns a video handle that is passed to other routines in the IndigoVideo Library. This step also demonstrates the use of the IndigoVideo error reporting routine, svPerror().
- 3. The program sets up the IndigoVideo board to receive video input from source #1. This is the default input, so you could omit this step; however, this step is included to demonstrate the use of the svSetParam() routine. The svSetParam() routine can be used to select input source, broadcast standard, and more. Note that it is not necessary to set these parameters before associating video input with a window; video parameters can be changed at any time. Also note that once set up, live video display requires no CPU intervention.

One of the arguments to **svSetParam()** is an array of long integers. Even-numbered elements of this array represent parameters to be changed, and the corresponding odd-numbered elements represent parameter values. In this program, the parameter being set is represented by the symbolic constant SV\_SOURCE, and the value is represented by the symbolic constant SV\_SOURCE1.

- 4. After the setup is complete, the program attaches the video input to the window, using the **svBindGLWindow()** routine. This routine takes three arguments: a video handle, a window identifier, and a third argument indicating how to display the video input. The sample program uses the symbolic constant SV\_IN\_REPLACE for the third argument, indicating that video should replace the contents of the window. The other possible values for the third argument are SV\_IN\_UNDER, for video underlay, SV\_IN\_OVER, for video overlay, and SV\_IN\_OFF, to deactivate video display in the window.
- 5. Finally, the program enters an IRIS GL event handling loop, displaying video input until the user elects to quit by pressing the <Esc> key. Once set up, live video display requires no CPU intervention. When the user exits, the program closes the video device by using the svCloseVideo() routine. This routine deallocates the data structures associated with the video handle.

More sophisticated event handling methods are presented later in this guide—new development should use X event handling rather than IRIS GL event handling. See Chapter 21, "Handling IndigoVideo Events," for more information on handling events.

## Chapter 18

# Controlling the IndigoVideo Input Window

This chapter explains how to set up IndigoVideo input, including:

- designating an input source
- creating and configuring an input window
- gaining exclusive use of the IndigoVideo board
- combining video and graphics

## Controlling the IndigoVideo Input Window

Chapter 17, "Getting Started with the IndigoVideo Library," showed you how to create a video input window. This chapter presents more details on the process and demonstrates how to control various aspects of the video display.

#### In this chapter:

- "Setting Input Parameters" on page 399 explains how to set input parameters that provide information about the input source and signal for the IndigoVideo board.
- "Querying Video Parameters" on page 401 describes how to obtain information about the IndigoVideo board status.
- "Positioning and Scaling the Video Input" on page 402 explains how to create and configure the video input window.
- "Preventing Other Programs from Using Video" on page 406 explains how to take control of the IndigoVideo board for exclusive use.
- "Combining Video and Graphics" on page 406 provides suggestions for using video with graphics, including overlays and underlays.

## **Setting Input Parameters**

You can use the **svSetParam()** routine to set a number of different video parameters. This section describes some of the parameters that affect the video input window. See *svGetParam*(3V) for a complete list of parameters.

**svSetParam()** takes three arguments: a video handle, an array of long integers specifying parameters and values, and an argument specifying the length of the array. The values in the array are interpreted in pairs: the first member of each pair represents the parameter to be changed; the second member represents the new value for that parameter. The code fragment in

Example 18-1 sets up the IndigoVideo board to receive PAL input over an S-Video connector.

**Example 18-1** Setting up the IndigoVideo Board for PAL Input

```
SVhandle V;
long param[4];
    /* . . . */
param[0] = SV_BROADCAST;
param[1] = SV_PAL;
param[2] = SV_VIDEO_MODE;
param[3] = SV_SVIDEO;
svSetParam(V, param, 4);
```

## Selecting an Input Source

You can plug up to three video inputs into the IndigoVideo board and select between them by changing the SV\_SOURCE parameter with **svSetParam()**. Set SV\_SOURCE to SV\_SOURCE1, SV\_SOURCE2, or SV\_SOURCE3. These constants correspond to the input connectors labeled 1, 2, and 3. The code fragment in Example 18-2 demonstrates setting up the input source.

## **Example 18-2** Selecting a Video Input Source

```
void setSource(SVhandle V, int source)
{
   long param[2];
   if (source < 1 || source > 3) {
        /* error */
   }
   param[0] = SV_SOURCE;
   switch (source) {
      case 1: param[1] = SV_SOURCE1;
        break;
      case 2: param[1] = SV_SOURCE2;
        break;
   case 3: param[1] = SV_SOURCE3;
        break;
}
svSetParam(V, param, 2);
}
```

## Selecting the Input Signal Type

To enable the IndigoVideo board to display video input correctly, you must set it to the correct broadcast standard and format. These may be set using **svSetParam()**. The broadcast standard parameter, SV\_BROADCAST, may be set to one of two values: SV\_NTSC for NTSC video, or SV\_PAL for PAL video. The SV\_VIDEO\_MODE parameter controls input format; it can be set to SV\_COMP for composite video input, or SV\_SVIDEO for S-Video input.

You can also set the default video mode and broadcast standard for each input source using the Video Control Panel. This procedure is explained in Chapter 22, "Using the IndigoVideo Utilities."

## Freezing and Restarting Video Input

Freeze the video input by setting the SV\_FREEZE parameter to TRUE. Doing so holds the current frame in the IndigoVideo frame buffer. Setting SV\_FREEZE to FALSE restarts live input.

## **Querying Video Parameters**

You can determine the value of a parameter by using the **svGetParam()** routine. This routine takes exactly the same arguments as **svSetParam()**, but instead of reading new parameter values from the array, it fills in the current parameter values in the appropriate places.

The code fragment in Example 18-3 displays the number of the currently selected input source.

#### **Example 18-3** Getting the Input Source Number

```
SVhandle V;
long param[2];
int source;
   /* ... */
param[0] = SV_SOURCE;
svGetParam(V, param, 2);
```

In addition to the input parameters already mentioned, two special readonly parameters can be used in conjunction with **svGetParam()** to get information about the video source (if any) connected to the current input:

#### SV SIGNAL STD

can be used to query the broadcast standard used by the current source. The parameter value returned is one of:

SV\_NTSC for NTSC input SV\_PAL for PAL input

SV\_NOSIGNAL if there is no signal on the current input source

## SV\_SIGNAL\_COLOR

can be used to query whether the current input signal contains color information. The returned value is either TRUE or FALSE.

## Positioning and Scaling the Video Input

By default, the IndigoVideo software displays video input at full resolution, with the origin of the video image at the upper left corner of the video input window. You can set the size and position of the video image using the **svSetSize()** and **svWindowOffset()** routines. If you call one of these routines after binding video to a window, you must rebind the video using either **svBindWindow()** or **svBindGLWindow()** for the change to take effect.

**Note:** The IndigoVideo positioning and scaling routines take arguments in pixels, not in IRIS GL coordinates. Because arguments to **svSetSize()** and **svWindowOffset()** are not in IRIS GL coordinates, they are not affected by the current IRIS GL transformation matrix.

## Setting the Size of the Video Image

You can set the size of the video image within certain constraints:

- The aspect ratio of the image (ratio of horizontal size to vertical size) must remain constant.
- The image can be scaled only in increments of 8 pixels of horizontal size and 6 pixels of vertical size.
- The image cannot be larger than the video frame size (640 by 480 pixels for NTSC, 768 by 576 pixels for PAL).

To allow the user to resize the video input window, you can use the IRIS GL minsize(), maxsize(), and stepunit() routines to constrain the window to a useful size.

Example 18-4 contains a listing of *sizeinput.c*, in /usr/people/4Dgifts/examples/dmedia/video/indigovideo, which implements a user-resizable video input window.

**Example 18-4** Creating a Scalable Video Input Window: *sizeinput.c* 

```
/*
 * Scalable GL Video Input Window
 */

#include <stdlib.h>
#include <gl/gl.h>
#include <gl/gl.h>
#include <gl/device.h>

main()
{
    short val;
    long win, dev, x, y;
    SVhandle V;

    /* Open window */
    minsize(80, 60);
    stepunit(8, 6);
    maxsize(SV_NTSC_XMAX, SV_NTSC_YMAX);
    keepaspect(SV_NTSC_XMAX, SV_NTSC_YMAX);
    win = winopen("video in");
```

```
if ((V = svOpenVideo()) == NULL) {
    svPerror("open");
    exit(1);
getsize(&x, &y);
svSetSize(V, x, y);
if (svBindGLWindow(V, win, SV_IN_REPLACE) < 0) {</pre>
    svPerror("bindwindow");
    svCloseVideo(V);
    exit(1);
}
/* Event loop */
qdevice(ESCKEY);
qdevice(WINQUIT);
qdevice(WINSHUT);
while (1) {
    dev = qread(&val);
    switch (dev) {
    case REDRAW:
        getsize(&x, &y); /* may have been resized */
        svSetSize(V, x, y);
        /* Re-bind window to scale input */
        if (svBindGLWindow(V, win, SV_IN_REPLACE) < 0){</pre>
            svPerror("bindwindow");
            svCloseVideo(V);
            exit(1);
        }
        break;
    case ESCKEY:
    case WINQUIT:
    case WINSHUT:
        svCloseVideo(V);
        winclose(win);
        exit(0);
        break;
}
```

In the preceding example, the window is constrained to sizes supported by the IndigoVideo software. The event loop handles the resizing of the video input. When the program receives a REDRAW event, which could indicate a size change, it determines the new size of the window, calls **svSetSize()** to scale the video input appropriately, and rebinds the video input to the window.

If you use **svSetSize()** to specify a size that IndigoVideo cannot produce, it will select the closest possible size. To determine what size will result from a given pair of arguments, use the **svQuerySize()** routine. The code fragment in Example 18-5 finds the closest match to the desired size and resizes the window accordingly.

**Example 18-5** Approximating the Requested Video Window Size

```
SVhandle V;
long win;
int x, y, new_x, new_y;
   /* ... */
svQuerySize(V, x, y, &new_x, &new_y);
prefsize(new_x, new_y);
winconstraints();
svSetSize(V, new_x, new_y);
svBindGLWindow(V, win, SV_IN_REPLACE);
```

## **Positioning the Video Image**

By default, the origin of the live video image is at the upper left corner of the live video input window. You can change this position using the **svWindowOffset()** routine. To do this, specify vertical and horizontal offsets, in pixels, from the upper left corner of the input window to the upper left corner of the video image. These values may be negative, meaning that you can use a small window to "pan" across the video image. The code fragment in Example 18-6 demonstrates the use of **svWindowOffset()**:

#### **Example 18-6** Specifying a Video Window Offset

```
SVhandle V;
int xoffset, yoffset, win;
   /* ... */
svWindowOffset(V, xoffset, yoffset);
svBindGLWindow(V, win, SV_IN_REPLACE);
```

**Note:** The live video image cannot be positioned such that any part of the image is off the edge of the screen. Thus, if you have a window in the upper left corner of the screen, negative window offsets will be ignored. The entire video image must remain within the screen dimensions, even if you are viewing only a small portion of the image.

## Preventing Other Programs from Using Video

To prevent other programs from changing video parameters while your program is running, you can request exclusive use of the IndigoVideo board by calling the **svUseExclusive()** routine with a value of TRUE for the *onoff* parameter, as demonstrated in Example 18-7.

**Example 18-7** Getting Exclusive Use of the IndigoVideo Board

```
SVhandle V;
int status;
    /* ... */
status = svUseExclusive(V, TRUE, SV_INPUT)
if (status == -1) {
    svPerror("Couldn't get exclusive use");
    /* error handling*/
}
```

While one process has exclusive use of the IndigoVideo board, any other process that makes a call to the IndigoVideo Library will receive an error.

To get out of exclusive use mode, call svUseExclusive() FALSE.

## **Combining Video and Graphics**

You can combine video and graphics in a window using either of two modes: video underlay or video overlay. In video replace mode, which has been used in this guide until now, all of the pixels in the video image are displayed. In video underlay and overlay modes, video pixels replace only selected graphics pixels. In video underlay mode, the decision whether to display a video pixel or a graphics pixel at a given location is based on the value of the graphics pixel. In video overlay mode, this decision is based on the value of the video pixel—meaning that in video underlay mode you

specify which parts of your graphic image should be replaced by video; in video overlay mode you specify which parts of the video image should be replaced by graphics.

Video underlay mode is typically used for such applications as video titling, where you want to display text or graphics superimposed over video. See Figure 15-6 on page 368 in Chapter 15, "VL Blending," for an illustration of this technique, but use the method described in this section for producing this effect with the IndigoVideo board.

You might use video overlay mode for a "TV weatherman" effect, superimposing a live video image over a computer-generated backdrop. See Figure 15-8 on page 370 in Chapter 15, "VL Blending," for an illustration of this technique, but use the method described in this section for producing this effect with the IndigoVideo board.

## Video Underlay Mode

In video underlay mode, video pixels replace graphics pixels that have a value of zero. In IRIS GL programs that use RGB mode, video pixels replace black graphics pixels. In X programs, and in IRIS GL programs that use color map mode, video replaces the color that is mapped to zero. There are advantages and disadvantages to both IRIS GL modes.

#### **RGB Mode IRIS GL Programs**

RGB mode allows you to take better advantage of the IRIS GL's special 3D effects such as lighting and shading; however, all black pixels are replaced by video, so you can't display black objects over video. Furthermore, in RGB mode, the Indigo workstation simulates 24-bit color using a dithering algorithm, which produces several black pixels in any expanse of a dark color. This means that dark objects appear partially transparent, which is usually the opposite of the desired effect. This effect is particularly noticeable in double-buffered mode. Getting the effect you want in RGB mode can require some experimentation.

A detailed description of IRIS GL lighting and shading routines is beyond the scope of this guide, but here are some hints for producing good-looking graphics in RGB mode:

- When drawing Gouraud shaded polygons, use fairly light colors—they appear more opaque than darker colors because light colors do not dither to black.
- When drawing lighted objects, use more than one light. With one light, objects are defined by the contrast between light and shadow; however, in video underlay mode, the shadows will look transparent. If you use two lights of contrasting colors on opposite sides of an object, you can define the object using the contrast between the two colors. For example, light a sphere from above using a white light, and fill the shadow by lighting it from below using a blue light. In some cases, using more than one light can affect the graphics performance.

The *Graphics Library Programming Guide* covers Gouraud shading and IRIS GL lighting in the chapters "Display and Color Modes," and "Lighting."

#### Color Map Mode IRIS GL Programs

Color map mode provides a certain flexibility in that it allows you to use any color you want without the side effects dithering can produce; however, it is much more difficult to produce lighted or Gouraud shaded polygons in color map mode. For applications that do not require these effects, you're better off using color map mode. The *Graphics Library Programming Guide* chapters listed in the previous section also describe lighting and Gouraud shading in color map mode.

Example 18-8 demonstrates the use of video underlay mode with IRIS GL color map mode. The program effectively clips the video input to a circle by drawing a circle of color zero on a white background.

**Example 18-8** Using IndigoVideo Underlay Mode

```
#include <gl/gl.h>
#include <svideo.h>
#include <gl/device.h>
```

```
void
drawScene(void)
   color(7);
   clear();
   color(0);
   arcf(320.0, 240.0, 200.0, 1.0, 0.0);
}
main(void)
   long win, dev;
   short val;
   SV_nodeP V;
   prefsize(640, 480);
   win = winopen("Video underlay test");
   ortho2(1, 640, 1, 480);
   if ((V = svOpenVideo()) == NULL) {
       svPerror("open video");
       exit(1);
   if (svBindGLWindow(V, win, SV_IN_UNDER) < 0) {</pre>
       svPerror("bind window");
       svCloseVideo(V);
       exit(2);
   }
   drawScene();
   qdevice(ESCKEY);
   qdevice(WINQUIT);
   qdevice(WINSHUT);
   while(1) {
       dev = qread(&val);
            switch (dev) {
             case ESCKEY:
             case WINQUIT:
              case WINSHUT:
                  svCloseVideo(V);
                  exit(0);
                  break;
              case REDRAW:
                  drawScene();
                  break;
   }
```

## Video Overlay Mode and Chroma Keying

In video overlay mode, video pixel values can be "keyed" out. Video pixels replace graphics pixels, except where the value of a video pixel matches one of the keyed values. This allows you to select certain colors in the video image to be replaced by graphics.

The IndigoVideo board has a 256-entry array of chroma keys, called the *chroma key map*. This array is indexed by pixel value (the 8-bit RGB pixels are treated as 8-bit unsigned integers), so entry zero in the chroma key map corresponds to pixels of value zero (black pixels). If the value of this entry is 1, black video pixels will be keyed out. If the value is zero, black video pixels will be displayed.

You can load a new chroma key map by using the **svLoadMap()** routine. The chroma key map is passed to **svLoadMap()** as a 256-entry array of *rgb\_tuple* structures, which are red, green, blue triplets. The *red* portion of the array is used for the chroma key map, and the rest of the array is ignored. The code fragment in Example 18-9 keys out black pixels by turning on the chroma key for pixel value zero.

#### **Example 18-9** Using Chroma Keying to Key Out Black Pixels

```
rgb_tuple chromamap[256];
SVhandle V;
    /* ... */
chomamap.red[0] = 1;
svLoadMap(V, SV_CHROMA_KEY_MAP, chromamap);
```

Example 18-10 and Example 18-11 contain listings of two *4Dgifts* programs that work together. Example 18-10, *voverlay.c*, demonstrates how to use IndigoVideo overlay mode. Example 18-11, *chromamap.c*, demonstrates how to use the chroma key map to set chroma key entries that *voverlay.c* can use.

## **Example 18-10** Using IndigoVideo Overlay Mode: *voverlay.c*

```
voverlay.c:
             "pool" ball that "bounces" around a 2-d "surface".
                  RIGHTMOUSE stops ball
                  MIDDLEMOUSE increases y velocity
                  LEFTMOUSE increases x velocity
 ^{\star} Adapted to show IndigoVideo overlay mode. To use, also compile and run
 * the chromamap.c example. As you set chroma key entries with chromamap,
 * the graphics generated by this program will begin to appear.
#include <stdio.h>
#include <svideo.h>
#include <gl/gl.h>
#include <gl/device.h>
long xmaxscrn, ymaxscrn;
                                /* maximum size of screen in x and y
                                                                              * /
#define XMIN 100
#define YMIN 100
#define XMAX 900
#define YMAX 700
long xvelocity = 0, yvelocity = 0;
main()
    Device dev;
    short val;
    long sizex, sizey;
    initialize();
```

```
while (TRUE) {
      while (qtest()) {
          dev = qread(&val);
          switch (dev) {
                                /* redraw window re: move/resize/push/pop */
                case REDRAW:
                    reshapeviewport();
                    ortho2(XMIN - 0.5, XMAX + 0.5, YMIN - 0.5, YMAX + 0.5);
                    drawball();
                    break;
                case LEFTMOUSE:
                                               /* increase xvelocity */
                   if (xvelocity >= 0)
                        xvelocity++;
                    else
                        xvelocity--;
                    break;
                                        /* increase yvelocity */
                case MIDDLEMOUSE:
                    if (yvelocity >= 0)
                        yvelocity++;
                    else
                        yvelocity--;
                    break;
                                        /* stop ball */
                case RIGHTMOUSE:
                   xvelocity = yvelocity = 0;
                   break;
                case ESCKEY:
                   gexit();
                    exit(0);
            }
       drawball();
    }
}
initialize() {
   SVhandle V;
   long win;
   xmaxscrn = getgdesc(GD_XPMAX)-1;
   ymaxscrn = getgdesc(GD_YPMAX)-1;
   prefposition(xmaxscrn/4,xmaxscrn*3/4,ymaxscrn/4,ymaxscrn*3/4);
   win = winopen("voverlay");
   winconstraints();
   doublebuffer();
   gconfig();
```

```
shademodel(FLAT);
    ortho2(XMIN - 0.5, XMAX + 0.5, YMIN - 0.5, YMAX + 0.5);
    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    qdevice(RIGHTMOUSE);
    /* Open video device */
    if ((V = svOpenVideo()) == NULL) {
        svPerror("open");
        exit(1);
    /* Associate video input with this window */
    if (svBindGLWindow(V, win, SV_IN_OVER) < 0) {</pre>
        svPerror("bindwindow");
        exit(1);
    }
drawball() {
    static xpos = 500, ypos = 500;
    long radius = 50;
    color(BLUE);
    clear();
    xpos += xvelocity;
    ypos += yvelocity;
    if (xpos > XMAX - radius ||
        xpos < XMIN + radius) {</pre>
        xpos -= xvelocity;
        xvelocity = -xvelocity;
    if (ypos > YMAX - radius ||
        ypos < YMIN + radius) {</pre>
        ypos -= yvelocity;
        yvelocity = -yvelocity;
    }
    color(YELLOW);
    circfi(xpos, ypos, radius);
    swapbuffers();
```

Example 18-11 contains a listing of *chromamap.c*, a program that demonstrates how to manipulate the chroma key map for programs that use video overlay.

**Example 18-11** Using the Chroma Key Map: *chromamap.c* 

```
chromamap.c
   This program demonstrates how to manipulate the IndigoVideo
   chroma key map for programs that use the video overlay feature.
   It shows the colors that correspond to the 256 entries in the map,
   where 0 is the lower left corner and 255 is the upper right.
   Clicking the left mouse button over a color toggles the value for that
   entry. An X mark in a box means that color will be keyed out (i.e.,
   the underlying graphics will show through.) A pull-down menu can
   be used to clear or set all of the entries. The program initializes
   all entries to 0 when it starts up.
   To demonstrate keying, compile and run the voverlay program in
   this directory. As you set entries in the key map, the graphics
   in voverlay will begin to appear. If you set all of the entries,
   only the graphics will appear. If you clear all of the entries,
   only video will appear.
#include <stdio.h>
#include <gl/gl.h>
#include <ql/device.h>
#include <svideo.h>
static SVhandle V;
static SVcolorMap keymap;
static long rgb8to32[256];
#define grey9() cpack(0x00E0E0E0)
#define grey7() cpack(0x00B0B0B0)
#define grey5() cpack(0x00808080)
#define bordercolor()
                       grey5()
static long xsize, ysize;
static long xorg, yorg;
#define YSIZE
                        16
#define XSIZE
                        16
```

```
#define BORDERSIZE
                      0.25
#define MOUSEXMAP(x) ( ((XSIZE+2*BORDERSIZE)*((x)-xorg))/(xsize) )
                     ( ((YSIZE+2*BORDERSIZE)*((y)-yorg))/(ysize) )
#define MOUSEYMAP(y)
static void
drawX(int i, int j)
    grey7();
    move2i(i,j);
    draw2i(i+1,j+1);
   move2i(i,j+1);
    draw2i(i+1,j);
static void
drawborder(int i, int j)
    bordercolor();
   move2i(i,j);
    draw2i(i+1,j);
    draw2i(i+1,j+1);
    draw2i(i,j+1);
    draw2i(i,j);
static void
drawcolor(int i, int j)
    cpack(rgb8to32[(j*XSIZE)+i]);
    rectfi(i,j,i+1,j+1);
static void
showmap(void)
    int i, j;
    /* Clear background */
    grey9();
    clear();
    ortho2(-BORDERSIZE, XSIZE+BORDERSIZE, -BORDERSIZE, YSIZE+BORDERSIZE);
```

```
/* Draw colored boxes for the 256 RGB colors */
    for (j=0; j<YSIZE; j++) \{
        for (i=0; i<XSIZE; i++) {
            drawcolor(i,j);
            if (keymap[i+(j*XSIZE)].red) {
                drawX(i,j);
        }
    }
    /* Draw borders around all the boxes */
   bordercolor();
    if ((xsize/XSIZE)>4) {
        for (j=0; j<=YSIZE; j++) \{
            move2i(0,j);
            draw2i(XSIZE,j);
        for (j=0; j<=XSIZE; j++) {</pre>
            move2i(j,0);
            draw2i(j,YSIZE);
    }
}
static void
fillmap(int fill)
    int i;
    for (i = 0; i < SV_CMAP_SIZE; i++)</pre>
       keymap[i].red = fill;
    showmap();
    if (svLoadMap(V, SV_CHROMA_KEY_MAP, keymap) < 0)</pre>
        printf("load map failed\n");
}
```

```
main(void)
    short val;
    int menu;
    int r, g, b;
    float mx, my;
    /* Open video device */
    if ((V = svOpenVideo()) == NULL) {
        svPerror("open");
        exit(1);
    }
    /* Create mapping of 8-bit RGB to 32-bit equivalents */
    for (r=0; r<8; r++) {
        for (b=0; b<4; b++) {
            for (g=0; g<8; g++) {
                 rgb8to32[(r<<5)|(b<<3)|g] =
                      ((r << 5) | (r << 2) | (r >> 1))
                      (((g<<5)|(g<<2)|(g>>1)) << 8)
                      (((b << 6) | (b << 4) | (b << 2) | b) << 16);
        }
    }
    keepaspect(XSIZE, YSIZE);
    winopen("chromamap");
    RGBmode();
    gconfig();
    qdevice(LEFTMOUSE);
    qdevice(MOUSEX);
    qdevice(MOUSEY);
    qdevice(MENUBUTTON);
    menu = defpup("chromamap %t|clear all|set all|exit");
    getsize(&xsize,&ysize);
    getorigin(&xorg,&yorg);
    /* Put map in known state */
    fillmap(0);
```

```
while (1) {
    switch(qread(&val)) {
        case REDRAW:
            reshapeviewport();
            getsize(&xsize,&ysize);
            getorigin(&xorg,&yorg);
            showmap();
            break;
        case MOUSEX:
            mx = MOUSEXMAP(val) - .25;
            if (mx < 0.0)
                mx = 0.0;
            else if (mx \ge XSIZE)
                mx = XSIZE-1;
            break;
        case MOUSEY:
            my = MOUSEYMAP(val) - .25;
            if (my < 0.0)
                my = 0.0;
            else if (my >= YSIZE)
                my = YSIZE-1;
            break;
        case LEFTMOUSE:
            /* Toggle the entry's key */
            if (val) {
                int i = (int)mx + (int)my * XSIZE;
                keymap[i].red = !keymap[i].red;
                drawcolor((int)mx, (int)my);
                if (keymap[i].red) {
                    drawX((int)mx, (int)my);
                drawborder((int)mx, (int)my);
                if (svLoadMap(V, SV_CHROMA_KEY_MAP, keymap) < 0)</pre>
                    printf("load map failed\n");
            break;
```

```
case MENUBUTTON:
    if (val) {
        switch (dopup(menu)) {
            case 1:
                 fillmap(0);
                     break;
            case 2:
                      fillmap(1);
                      break;
            case 3:
                      exit(0);
        }
    }
    break;
```

## Chapter 19

# **Producing IndigoVideo Output**

This chapter explains how to set up IndigoVideo output, including:

- designating an output area
- outputting one frame at a time

# Producing IndigoVideo Output

Producing live video output from the IndigoVideo board is simple. The IndigoVideo board constantly encodes a portion of the screen to video unless output has been explicitly turned off. You can also use IndigoVideo to produce single-frame output. In single-frame output mode, both live input and live output are disabled.

#### In this chapter:

- "Selecting the IndigoVideo Live Output Area" on page 423 explains how to designate a portion of the screen to be output to video.
- "Setting Output Parameters" on page 426 explains how to set up the output configuration.
- "Generating Single-frame Output" on page 426 explains how to output video one frame at a time.

## Selecting the IndigoVideo Live Output Area

Select the portion of the screen to be output using the **svOutputOffset()** routine. This routine specifies the upper left corner of the output "window." The broadcast standard parameter (SV\_BROADCAST) determines the size of the output area—640 by 480 pixels for NTSC, 768 by 576 pixels for PAL. **svOutputOffset()** does not create an actual window, and the output area is not in any way delineated on the Indigo workstation monitor.

Example 19-1 demonstrates setting the location of the IndigoVideo output area.

**Example 19-1** Setting the Location of the IndigoVideo Output Window

```
#include <stdio.h>
#include <stdlib.h>
#include <svideo.h>
 * Simple Output Window
 * Change output location of NTSC video window
main(int argc, char *argv[])
   int xstart, ystart;
   SVhandle V;
   if (argc != 3) {
      fprintf(stderr, "Usage: %s x_start y_start\n",
      argv[0]);
      exit(1);
   xstart = atoi(argv[1]);
   ystart = atoi(argv[2]);
   /* open video device */
   if ((V = svOpenVideo()) == NULL) {
      svPerror("open");
      exit(1);
   }
   /* change location of output window */
   svOutputOffset(V, xstart, ystart);
   exit(0);
}
```

You can use this program to record the output of any graphics program to videotape.

To select an output window:

1. Use *xwininfo*(1) to get the location of the desired window on the screen by entering:

```
xwininfo
```

2. Place the mouse cursor in the window that you want information about and click any of the mouse buttons. *xwininfo* displays information about the selected window, including the coordinates of its upper left corner:

```
xwininfo ==> Window id: 0x3400001 (Hubert)
==> Absolute upper-left X: 14
==> Absolute upper-left Y: 156
...
```

3. You can then use these coordinates as arguments to *vout* to place the output area over the selected window:

```
vout 14 156
```

To incorporate video output into a program, you can use the code fragment from Example 19-2 to place the upper left corner of the output area at the upper left corner of the current IRIS GL window.

## **Example 19-2** Aligning a Video Output Area with an IRIS GL Window

```
long x, y, xsize, ysize, screensize;
SVhandle V;
    /* ... */
getorigin(&x, &y);
getsize(&xsize, &ysize);
screensize = getgdesc(GD_YPMAX);
y = screensize - (y + ysize);
svOutputOffset(V, x, y);
```

Your program should recalculate the output offset whenever a "REDRAW" event is received so that the program relocates the output area whenever you move the window.

## **Setting Output Parameters**

This section explains how to set parameters that affect live video output.

## **Turning Output On and Off**

The SV\_VIDEO\_OUTPUT parameter can be used to turn live video output on and off. The default value for SV\_VIDEO\_OUTPUT is TRUE. Setting SV\_VIDEO\_OUTPUT to FALSE disables live video output.

## **Synchronizing Output with Input**

The IndigoVideo board normally synchronizes video input and output so that they have the same field rates. This is necessary if you are doing any input; however, when you are using the IndigoVideo board for output only, you will get better results if you disable this synchronization by setting SV\_SLAVE to FALSE. The default value for SV\_SLAVE is TRUE.

## **Filtering Output**

When encoding dithered graphics to video, you may encounter vertical bands on the composite output. You can eliminate these by setting SV\_OUTPUT\_FILTER to TRUE. The default value for SV\_OUTPUT\_FILTER is FALSE.

## **Generating Single-frame Output**

In single-frame output mode, the IndigoVideo board is configured as a 24-bit RGB framebuffer, and both live input and live output are disabled. Sending the board a still frame automatically reconfigures the board into single-frame output mode.

Use the **svPutFrame()** routine to send a still frame to the IndigoVideo board. The data must be in the format used by the IRIS GL **lrectwrite()** routine (32-bit pixels, ordered 0x00BBGGRR, that is, 1 empty byte, followed by 1 byte each of blue, green, and red).

The rows of pixels must be ordered top-to-bottom. The video capture routines also use this 32-bit RGB format. See Chapter 20, "Capturing Video from IndigoVideo," for information on the video capture routines, and "IndigoVideo Data Formats" in Chapter 17 for information on the data formats used by IndigoVideo.

Exit single-frame mode by calling **svPutFrame()** with a NULL frame pointer.

Example 19-3 reads an RGB image file and sends it to the IndigoVideo board as a still frame.

**Example 19-3** Sending a RGB Image as a Still Video Frame

```
* Use video card as a 24 bit RGB framebuffer.
* ipaste SGI image file to screen, vpaste image file to
video
 * /
#include <stdio.h>
#include <stdlib.h>
#include <svideo.h>
#include <gl/image.h>
static void imgerror(char *);
static long getvideoparam(SVhandle, long);
static void sgiimage_to_buf(IMAGE *, unsigned long *,
                            unsigned long, unsigned long);
int main(int argc, char *argv[])
   unsigned long *rgb_buf, x_size, y_size;
   IMAGE* ip;
   SVhandle V;
   int pal_mode = 0;
   char line[30];
   if (argc != 2) {
      fprintf(stderr, "Usage: %s imagefile\n", argv[0]);
      exit(1);
   i_seterror(imgerror);
```

```
/* Open video device */
   if ((V = svOpenVideo()) == NULL {
      svPerror("open");
      exit(1);
   /* Size image according to broadcast standard */
   pal_mode = (getvideoparam(V, SV_BROADCAST) == SV_PAL);
   if (pal_mode) {
      x_size = SV_PAL_XMAX;
      y_size = SV_PAL_YMAX;
      x_size = SV_NTSC_XMAX;
      y_size = SV_NTSC_YMAX;
   /* Open image */
   if((ip = iopen(argv[1], "r")) \le (IMAGE*)0) 
      fprintf(stderr, "could not open image file %s\n",
              argv[1]);
      exit(1);
   }
   rgb_buf = (unsigned long *)malloc(x_size * y_size *
   sizeof(long));
   /* Convert to RGB buffer */
   sgiimage_to_buf(ip, rgb_buf, x_size, y_size);
   iclose(ip);
   /* Output 24-bit RGB image */
   if (svPutFrame(V, (char *)rgb_buf) < 0) 
      svPerror("putframe");
      svCloseVideo(V);
      exit(1);
   printf("Type <Enter> to exit:");
   (void) gets(line);
   exit(0);
static void
imgerror(char *s)
   fputs(s, stderr);
```

{

```
/* center SGI image file in buffer */
static void
sgiimage_to_buf(IMAGE *ip, unsigned long *rgb_buf, unsigned
long bxsize, unsigned long bysize)
   short *red, *green, *blue, *r, *g, *b;
   int bxstart, ixstart, bystart, iystart;
   int iy, by, x, nx, ny;
   unsigned long *rgb;
   red = malloc(ip->xsize * sizeof(short));
   green = malloc(ip->xsize * sizeof(short));
   blue = malloc(ip->xsize * sizeof(short));
   bzero(rgb_buf,bxsize*bysize*sizeof(long));
   if (ip->xsize > bxsize) {
      bxstart = 0;
      ixstart = (ip->xsize - bxsize)/2;
      nx = bxsize;
   } else {
      ixstart = 0;
      bxstart = (bxsize - ip->xsize)/2;
      nx = ip->xsize;
   if (ip->ysize > bysize) {
      bystart = 0;
      iystart = (ip->ysize - bysize)/2;
      ny = bysize;
   } else {
      iystart = 0;
      bystart = (bysize - ip->ysize)/2;
      ny = ip->ysize;
   for (iy=iystart, by=bystart; iy<iystart + ny; iy++, by++)</pre>
      getrow(ip, red, iy, 0);
      getrow(ip, green, iy, 1);
      getrow(ip, blue, iy, 2);
      rgb = &rgb_buf[(by*bxsize) + bxstart];
      r = &red[ixstart];
      g = &green[ixstart];
      b = &blue[ixstart];
      for (x = 0; x < nx; x++) {
           *rgb++ = (*b++ << 16) | (*g++ << 8) | *r++;
   }
```

```
free(red);
free(green);
free(blue);

return 0;
}

static long
getvideoparam(SVhandle V, long arg)
{
    long pvbuf[2];
    pvbuf[0] = arg;
    if (svGetParam(V, pvbuf, 2) < 0)
        svPerror("svGetParam");
    return pvbuf[1];
}</pre>
```

## Chapter 20

Capturing Video from IndigoVideo

This chapter explains how to capture video using the IndigoVideo board.

# Capturing Video from IndigoVideo

This chapter explains how to capture video using the IndigoVideo board. The IndigoVideo Library provides these three methods for capturing video:

- single frame
- burst mode
- continuous mode

## In this chapter:

- "Captured Video Data Formats" on page 434 provides a brief introduction to the data formats for capturing video.
- "Capturing a Single Video Frame" on page 436 describes the single-frame capture method. You can easily capture a single frame of video with just one call to the convenience routine svCaptureOneFrame().
- "Capturing Video Frames in Burst Mode" on page 439 describes the burst-mode capture method. In burst mode, IndigoVideo captures a buffer full of sequential frames at full frame rate in most formats. You can use burst mode capture in conjunction with a computercontrollable video device to read in segments of video and stop the deck while your program processes the buffer full of frames.
- "Capturing Video Frames in Continuous Mode" on page 441 describes
  the continuous-mode capture method. In continuous mode,
  IndigoVideo captures frames of video into a queue at less than full
  frame rate. In this mode, you can capture reduced size and reduced
  frame rate video directly to disk, allowing you to collect images for a
  movie without a computer-controllable video device. In continuous
  capture mode, you can capture frames at no more than half the normal
  frame rate.
- "Using Data Conversion Routines" on page 444 describes how to convert video data for graphics display.

## **Captured Video Data Formats**

This section describes the data formats used by the IndigoVideo Library capture routines. For more information on these formats, see "IndigoVideo Data Formats" in Chapter 17. The data formats are listed below, along with the symbolic constants that the IndigoVideo Library uses to identify them.

#### SV RGB8 FRAMES

This is the default format. In this mode, IndigoVideo captures 8-bit RGB frames. The fields that make up these frames are not interleaved; that is, all of the odd lines in the image come first, followed by all the even lines. 8-bit RGB frames are dithered unless the SV\_DITHER parameter is set to FALSE.

#### SV\_YUV411\_FRAMES\_AND\_BLANKING\_BUFFER

In this mode, the IndigoVideo board captures full-size YUV frames, complete with the data carried in the blanking interval of the video signal. YUV format provides the best resolution and the most accurate color representation of the available options. The blanking portion of the video signal is sometimes used to carry extra information, such as the closed captioning provided on some television broadcasts. The blanking buffer takes the form of an extra 22 lines of data preceding the picture data; therefore, frames in this format are 640 by 502 pixels (NTSC) or 768 by 598 pixels (PAL). Use the **svFindVisibleRegion()** routine to find the first line of the image data following the blanking buffer.

#### SV\_RGB32\_FRAMES

This format is not produced directly by IndigoVideo, but is produced by the convenience routines **svCaptureOneFrame()** and **svYUVtoRGB()**. In these frames, each pixel is represented by a 32-bit word containing 24 bits of RGB data.

#### SV YUV411 FRAMES

This format consists of interleaved 4:1:1 YUV frames without the blanking data mentioned above. This format is not produced directly by the IndigoVideo hardware, so it cannot be captured in burst capture mode; however, the other capture routines produce this format. Use **svYUVtoRGB()** to convert YUV frames to 24-bit RGB.

Table 20-1 summarizes the storage requirements for the various data formats:

**Table 20-1** Pixel Sizes for Video Data

Format	Bytes per Pixel
8-bit RGB	1
32-bit RGB	4
YUV	2

Several of the capture routines take an svCaptureInfo structure as an argument. The fields in the svCaptureInfo structure are listed in Table 20-2.

**Table 20-2** Fields in the svCaptureInfo Structure

Field	Value
format	format (one of the symbolic constants listed above)
width	width of captured frames, in pixels
height	height of captured frames, in pixels
size	size of the capture buffer, in frames
samplingrate	used for continuous capture

Note that the *width* and *height* members of the svCaptureInfo structure are input-output parameters; that is, if you set them to an unsupported size such as 321 by 243 pixels, they will be set to the nearest approximation of the requested size by the capture routines.

If you are capturing 8-bit RGB frames, and you do not specify width and height, the IndigoVideo Library will use the width and height of the live video input window, if one is active. The video capture routines set the size of the video image to the requested size, so if you have a live video input window active, you may have to reset the size of the video image after capturing frames.

## Capturing a Single Video Frame

The IndigoVideo Library provides a convenient way to capture a single frame of video in any of the supported formats. The **svCaptureOneFrame()** routine captures data into a user-allocated buffer. This buffer must be big enough to hold a single frame in the specified format.

You can use the **svQueryCaptureBufferSize()** routine to determine the size of buffer required by **svCaptureOneFrame()**, then use **malloc()** to reserve a buffer of the appropriate size, as demonstrated in Example 20-1.

#### **Example 20-1** Determining the Capture Buffer Size

```
SVhandle V;
svCaptureInfo capInfo;
char *buffer;
int width, height, bufSize;
    /* ... */
capInfo.format = SV_RGB8_FRAMES;
capInfo.width = width;
capInfo.height = height;
(void) svQueryCaptureBufferSize(V, &capInfo, &bufSize);
buffer = malloc(bufSize);
```

Example 20-2 contains a listing of the /usr/people/4Dgifts/examples/dmedia/video/indigovideo program rgbgrab.c, which demonstrates the use of svCaptureOneFrame(). This program lets the user click the left mouse button to grab a frame of 8-bit RGB data and display it in a window. See oneframe.c for an example of capturing other data formats.

**Example 20-2** Grabbing a Single Frame of 8-bit RGB data: *rgbgrab.c* 

```
* Simple frame grabbing using video capture.
 * To use: click on the left mouse button in either window
 * to grab a frame and display it.
#include <stdio.h>
#include <stdlib.h>
#include <svideo.h>
#include <ql/ql.h>
#include <gl/device.h>
#define RGBBUFSIZE (SV_NTSC_XMAX*SV_NTSC_YMAX)
static char captureData[RGBBUFSIZE], rgbbuf[RGBBUFSIZE];
main()
    SVhandle V;
    long dev, live_win, still_win;
    short val;
    int w, h;
    /* Open window */
    foreground();
    prefsize(SV_NTSC_XMAX, SV_NTSC_YMAX);
    still_win = winopen("Grabbed frame");
    RGBmode();
    gconfig();
    pixmode(PM_SIZE, 8);
    prefsize(SV_NTSC_XMAX, SV_NTSC_YMAX);
    live_win = winopen("Live video");
    /* Open video device */
    if ((V = svOpenVideo()) == NULL) {
        svPerror("open");
        exit(1);
    /* Associate video input with this window */
    if (svBindGLWindow(V, live_win, SV_IN_REPLACE) < 0) {</pre>
        svPerror("bindwindow");
        svCloseVideo(V);
        exit(1);
    }
```

```
printf("Use leftmouse to grab frame\n");
    /* Event loop */
    qdevice(LEFTMOUSE);
    qdevice(WINQUIT);
    qdevice(WINSHUT);
    qdevice(ESCKEY);
    while (1) {
        dev = qread(&val);
        switch (dev) {
        case LEFTMOUSE:
            if (val != 1)
                               /* button-press */
                break;
            w = SV_NTSC_XMAX;
            h = SV_NTSC_YMAX;
            if (svCaptureOneFrame(V, SV_RGB8_FRAMES,
                        &w, &h, (char *)captureData) < 0) {
                svPerror("captureburst");
                exit(-1);
            }
            svInterleaveFields(TRUE, captureData, rgbbuf, w,
h);
            winset(still_win);
            lrectwrite(0, 0, w-1, h-1, (unsigned long *)
rgbbuf);
            winset(live_win);
            break;
        case ESCKEY:
            if (val)
                       /* exit on key up */
                break;
        case WINQUIT:
        case WINSHUT:
            svCloseVideo(V);
            winclose(live_win);
            winclose(still_win);
            exit(0);
            break;
        }
    }
}
```

## **Capturing Video Frames in Burst Mode**

Use the **svCaptureBurst()** routine to capture a contiguous series of frames into a previously allocated buffer. In burst mode, you can capture full-sized YUV frames with blanking buffers or 8-bit RGB frames at full or reduced size. Initiating a burst capture puts the IndigoVideo board into exclusive mode, which remains set until the capture is complete or an error occurs.

You must pass **svCaptureBurst()** a pointer to an svCaptureInfo structure, which determines the number and type of frames to be captured. This svCaptureInfo structure can also be passed to **svQueryCaptureBufferSize()** to determine how much memory to allocate for the capture buffer.

The fields in the svCaptureInfo structure are listed in Table 20-2. Remember that the width and height members of the svCaptureInfo structure are input-output parameters; if you set them to an unsupported size, they will be set to the nearest approximation of the requested size when you call svQueryCaptureBufferSize() or svCaptureBurst().

The final argument to **svCaptureBurst()** is an optional pointer to a *bitvector* for the SV\_RGB8\_FRAMES format, which can be used to determine whether any fields have been dropped during the capture. Fields must occasionally be dropped during capture to avoid visible tearing in the image due to scan rate conversion. The SV\_FIELDDROP parameter controls whether fields are dropped, see *svGetParam*(3) for details.

Every captured frame is represented by a pair of status bits (one for each field) in the bitvector. Each bit is set to either SV\_EVEN\_FIELD or SV\_ODD\_FIELD. Every frame should consist of an even field followed by an odd field; if this is not true, then one or more fields have been dropped during the capture. Use the SV\_GET\_FIELD macro, which is defined in *svideo.h*, to determine the even or odd value for a specific field.

The code fragment in Example 20-3 prints the even/odd values for the fields that make up the frames from a captured burst.

#### **Example 20-3** Using the SV\_GET\_FIELD Macro

```
for (f = 0; f < info->size; f++) {
  printf("%s-%s ",
    SV_GET_FIELD(framevec, 2*f) == SV_EVEN_FIELD ? "even" :
    "odd",
    SV_GET_FIELD(framevec, 2*f+1) == SV_EVEN_FIELD ? "even" :
    "odd");
}
putchar('\n');
```

The status bits are filled in starting with the least significant bit in the first byte of the bitvector. You must allocate one byte of bitvector for every four frames or fraction thereof. Use the SV\_BITVEC\_SIZE macro to calculate the size of the buffer in bytes for a given number of frames.

The format can be either 8-bit RGB or YUV with blanking buffer. This is because data is transferred directly from IndigoVideo to memory, and the IndigoVideo hardware produces only these two formats.

If you need 32-bit RGB frames, you can capture YUV frames and use **svFindVisibleRegion()** and **svYUVtoRGB()** to convert the data, as discussed in "Converting YUV Data to RGB" on page 444.

See the *burstcapt.c* sample program in */usr/people/4Dgifts/examples/dmedia/video/indigovideo* for a demonstration of how to capture and display a burst of frames in either format.

The number of frames that can be captured in burst mode is limited to what will fit in the memory buffer, which has a maximum size of 8 MB. The frame size affects how many frames will fit in the buffer, because larger frames have more data.

**Note:** Currently, **svCaptureBurst()** can only capture YUV frames with blanking data at half the full frame rate.

The code fragment in Example 20-4 shows how to use **svCaptureBurst()**.

#### **Example 20-4** Capturing Frames in Burst Mode

```
SVhandle V;
char *buffer, *bitVector;
int numberOfFrames=8, bufSize, bitVectorSize;
svCaptureInfo capInfo;
/* ... */
capInfo.format = SV_YUV411_FRAMES;
capInfo.width = SV_PAL_XMAX;
capInfo.height = SV_PAL_YMAX;
capInfo.size = numberOfFrames;
(void) svQueryCaptureBufferSize(V, &capInfo, &bufSize);
buffer = malloc(bufSize);
bitVectorSize = numberOfFrames / 4 + 1;
bitVector = malloc(bitVectorSize);
if (svCaptureBurst(V, &capInfo, buffer, bitVector) < 0) {</pre>
   svPerror("capture burst");
} else { /* process frames */
```

## **Capturing Video Frames in Continuous Mode**

In continuous capture mode, IndigoVideo writes frames of video into a queue while your program reads frames out of the queue. To enter continuous capture mode, call <code>svInitContinousCapture()</code>, which allocates memory for the capture queue and begins capturing frames. You can read frames from the queue using the <code>svGetCaptureData()</code> routine. Once you are done with a frame, you must release it using <code>svUnlockCaptureData()</code>. When you are done capturing frames, call <code>svEndContinuousCapture()</code> to leave continuous capture mode.

Continuous capture mode does not give you full frame rate—the maximum frame rate achievable in this mode is one half the normal frame rate. The *samplingrate* member of the svCaptureInfo structure specifies the number of frames seen for each frame captured. Thus, a sampling rate of two captures gives you every other frame, and a sampling rate of four captures gives you every fourth frame.

## **Entering Continuous Capture Mode**

Use **svInitContinousCapture()** to enter continuous capture mode.

The code fragment in Example 20-5 demonstrates how to initialize continuous capture. See the *contcapt.c* sample program in */usr/people/4Dgifts/examples/dmedia/video/indigovideo* for a complete program using continuous capture.

#### **Example 20-5** Initializing Continuous Capture Mode

```
SVhandle V;
int success;
int width=320, height=240, queueSize=16, samplingRate=2;
svCaptureInfo capInfo;
    /* ... */
capInfo.format = SV_RGB8_FRAMES;
capInfo.width = width;
capInfo.height = height;
capInfo.size = queueSize;
capInfo.samplingrate = samplingRate;
success = svInitContinuousCapture(V, capInfo);
```

svInitContinousCapture() allocates the frame capture queue and takes control of the IndigoVideo board using the svUseExclusive() routine. If another program already has exclusive use of the board, svInitContinousCapture() returns -1; if it succeeds, svInitContinuousCapture() returns 0. In this example, 8-bit RGB frames are being captured at half size and half frame rate.

Call **svEndContinuousCapture()** after you have completed video capture, to release control of the board. If you release control of the video board using **svUseExclusive()** during continuous capture mode, continuous capture mode is automatically terminated.

### **Accessing Captured Data**

To access the next frame in the video capture queue, use the **svGetCaptureData()** routine. When you're done with the frame, release it using **svUnlockCaptureData()**, so that IndigoVideo can reuse the memory.

The code fragment in Example 20-6 demonstrates how to access and release captured frames.

**Example 20-6** Accessing and Releasing Captured Frames

```
SVhandle V;
void *data;
long fieldID;
long curFrame, maxFrames;
   /* ... */
while (TRUE) {
   svGetCaptureData(V, &data, &fieldID);
   if (data == NULL) {
       /* no frame available yet */
      sginap(1);
   } else {
      /* process data */
      svUnlockCaptureData(V, data);
      curFrame++;
      if (curFrame == maxFrames) break;
}
```

The *data* parameter is pointed to the next frame in the queue, and the *fieldID* parameter is filled in to indicate the ID of the first field in the frame. If there are no frames in the queue, the *data* parameter is set to NULL (this will happen if you are processing frames faster than they are captured).

Because there are two fields in each frame, the *fieldID* value increases by 2 for each video frame. Therefore, for a sampling rate of S, the *fieldID* should increase by  $2 \times S$  between captured frames. If the *fieldID* increases by more than  $2 \times S$  between captured frames, you have missed one or more frames. This could be because the system load is high, or because the queue has filled up, forcing the IndigoVideo board to drop frames.

Although you don't have to unlock a frame before you get the next one, it's a good idea; if you keep a lot of frames locked, you will probably force the IndigoVideo software to drop frames. At any rate, you should unlock frames in the same order in which you received them. The IndigoVideo software writes the frames in order, and if it encounters a locked frame, it will block until the frame is unlocked.

The *data* parameter points to memory in the queue, which may be overwritten as soon as you release the element. Once you have the queue element, you'll probably want to write it to disk or copy it to a location where it won't be overwritten.

#### **Leaving Continuous Capture Mode**

To leave continuous capture mode, call **svEndContinuousCapture()**, as demonstrated below:

```
SVhandle V;
   /* ... */
svEndContinuousCapture(V);
```

## **Using Data Conversion Routines**

This section describes the convenience routines for converting data from IndigoVideo into other formats. For more information on the various data formats that IndigoVideo uses, see "IndigoVideo Data Formats" in Chapter 17.

#### Converting YUV Data to RGB

The IndigoVideo Library provides a utility routine, **svYUVtoRGB()**, for converting YUV frames to RGB format. This routine produces 32-bit pixels, as used by the IRIS GL **lrectwrite()** routine. Each pixel contains 24 bits of RGB data and 8 bits of unused space. (In the YUV format, a pixel takes up only 16 bits, so if you're trying to record frames to disk as fast as possible, you should store the YUV data directly and convert it later.)

Rows of pixels in the YUV frames are ordered top-to-bottom. This differs from the default ordering used by the IRIS GL **Irectwrite()** routine, which is bottom-to-top. If you set the *invert* parameter to TRUE, the **svYUVtoRGB()** routine will return an RGB frame with lines ordered bottom-to-top. If *invert* is FALSE, **svYUVtoRGB()** will not perform this inversion (this is useful because the X Window System expects the top-to-bottom ordering).

Alternately, if you want to display the frame, you can set up **lrectwrite()** to use top-to-bottom ordering by using the IRIS GL **pixmode()** routine, as shown in Example 20-7.

#### **Example 20-7** Setting Top-to-Bottom pixmode for YUV

```
boolean invert = FALSE;
int width, height;
long *yuv_buf;
long rgb_buf;
    /*... */
rgb_buf = malloc(width*height*sizeof(*rgbbuf));
if (svYUVtoRGB(invert, yuv_buf, rgb_buf, width, height)==-1)
{
    /* error */
}
pixmode(PM_TTOB, 1); /* pixel ordering top-to-bottom */
lrectwrite(0, 0, width-1, height-1, rgb_buf);
```

**Note:** This code may not work on some older IRIS workstations.

If you capture YUV frames with blanking data, you can use **svFindVisibleRegion()** to locate the start of the YUV image data, as demonstrated in Example 20-8.

#### **Example 20-8** Finding Image Data in YUV with Blanking Frames

```
SVhandle V;
void *frame_with_blanking, *frame;
long fieldID;
   /* ... */
svGetCaptureData(V, &frame_with_blanking, &fieldID);
svFindVisibleRegion(V, frame_with_blanking, &frame);
svUnlockCaptureData(frame_with_blanking);
   /* process frame */
```

Example 20-9 contains a listing of *vgrab.c*, in */usr/people/4Dgifts/examples/dmedia/video/indigovideo*, which demonstrates how to convert grabbed YUV frames to RGB images.

**Example 20-9** Grabbing YUV Frames to Save as RGB Images: *vgrab.c* 

```
vgrab.c
        Grab YUV frames, save as SGI RGB images
 * /
#include <stdio.h>
#include <gl/gl.h>
#include <gl/image.h>
#include <gl/device.h>
#include <svideo.h>
#define GRABFILE
                        "out.rgb"
#define RGBBUFSIZE
(SV_PAL_XMAX*SV_PAL_YMAX*sizeof(long))
static char rgbbuf[RGBBUFSIZE];
 * Dump rgb data to image file
 * /
void
dumpImage(char *data, int xsize, int ysize)
{
    IMAGE *image;
    short rbuf[SV_PAL_XMAX];
    short gbuf[SV_PAL_XMAX];
    short bbuf[SV_PAL_XMAX];
    int x, y, z;
    image = iopen(GRABFILE, "w", RLE(1), 3, xsize, ysize, 3);
    for (y=0;y<ysize;y++) {
        for(x=0;x<xsize;x++) {
            bbuf[x] = *(data+1);
            gbuf[x] = *(data+2);
            rbuf[x] = *(data+3);
            data += 4;
        putrow(image, rbuf, y, 0);
        putrow(image, gbuf, y, 1);
        putrow(image, bbuf, y, 2);
    iclose(image);
```

```
main(int argc, char **argv)
    short val;
    long livewin, stillwin, x, y;
    int width, height;
    SVhandle V;
    long param[2];
    int videoon = 1;
    /* Open video device */
    if ((V = svOpenVideo()) == NULL) {
        svPerror("open");
        exit(1);
    }
    /* Determine window size based on signal standard */
    param[0] = SV_BROADCAST;
    svGetParam(V, param, 2);
    if (param[1] == SV_PAL) {
        width = SV_PAL_XMAX;
        height = SV_PAL_YMAX;
    } else {
        width = SV_NTSC_XMAX;
        height = SV_NTSC_YMAX;
    }
    /* Open windows */
    foreground();
    prefsize(width, height);
    stillwin = winopen("Grabbed frame");
    RGBmode();
    gconfig();
    /* Set video window background to black */
    cpack(0x0);
    clear();
    maxsize(width, height);
    keepaspect(width, height);
    stepunit(8, 6);
    livewin = winopen("video in");
    RGBmode();
    gconfig();
    getsize(&x, &y);
    svSetSize(V, x, y);
```

```
/* Associate video input with livewin */
if (svBindGLWindow(V, livewin, SV_IN_REPLACE) < 0) {</pre>
    svPerror("bindwindow");
    exit(1);
}
printf("Click on left mouse button to grab frame\n");
qdevice(LEFTMOUSE);
qdevice(WINQUIT);
qdevice(WINSHUT);
qdevice(ESCKEY);
while (1) {
    switch (qread(&val)) {
        case LEFTMOUSE:
            if (val != 1)
                break;
            svCaptureOneFrame(V, SV_RGB32_FRAMES, &width,
                               &height, rgbbuf);
            winset(stillwin);
            lrectwrite(0, 0, width-1, height-1,
                       (unsigned long *) rgbbuf);
            winset(livewin);
            if (svSetSize(V, x, y) < 0) {
                svPerror("setsize");
                exit(1);
            /* Re-bind window to re-scale output */
            if (svBindGLWindow(V, livewin,
                SV_IN_REPLACE) < 0) {
                svPerror("bindwindow");
                exit(1);
            dumpImage(rgbbuf, width, height);
            printf("saved image to file %s\n", GRABFILE);
            break;
```

```
case REDRAW:
           reshapeviewport();
            getsize(&x, &y);
            svSetSize(V, x, y);
            /* Re-bind window to re-scale output */
            if (svBindGLWindow(V, livewin,
                SV_IN_REPLACE) < 0) {
                svPerror("bindwindow");
                exit(1);
            break;
        case ESCKEY:
            if (val)
                            /* exit on key up */
                break;
        case WINQUIT:
        case WINSHUT:
            winclose(stillwin);
            winclose(livewin);
            svCloseVideo(V);
            exit(0);
            break;
}
```

## **Using 8-bit RGB Capture Data**

The fields in a frame of 8-bit RGB data captured with **svGetCaptureData()** are not interleaved; all the even rows of pixels are stored before all the odd rows of pixels. In addition, the rows within the fields are ordered top-to-bottom. The IndigoVideo Library provides a convenience routine, **svInterleaveFields()**, to interleave, and optionally invert, the fields. It produces 8-bit RGB data rather than SV\_RGB\_FRAMES data.

Example 20-10 demonstrates how to invert fields and interleave them.

#### **Example 20-10** Interleaving 8-bit RGB Fields with Inversion

```
boolean invert = TRUE;
char *fields, *rgb8frame;
int width, height;
   /* ... */
frame = malloc(width*height);
svInterleaveFields(invert, fields, rgb8frame,width, height)
```

Once interleaved, you can display the 8-bit RGB data directly on an Indigo workstation with Entry graphics that has the *svideo* software installed, by using the **RGBmode()**, **pixmode()**, and **lrectwrite()** routines, as demonstrated in Example 20-11.

#### **Example 20-11** Displaying Interleaved 8-bit RGB Data

```
char *rgb8frame;
int width, height;
    /* ... */
RGBmode();
gconfig();
pixmode(PM_SIZE, 8);
lrectwrite(0, width-1, 0, height-1, rgb8frame);
```

To display the 8-bit RGB data on other systems, convert it to the more common 32-bit RGB format by using the **svRGB8toRGB32()** routine, which converts the data to 32-bit in addition to interleaving, and, optionally, inverting the fields, as demonstrated in Example 20-12.

#### **Example 20-12** Converting 8-bit RGB Capture Data to 32-bit RGB

```
char *rgb8fields;
long *rgb32frame;
int width, height;
boolean invert = TRUE;
    /* ... */
rgb32frame = malloc(width*height*sizeof(*rgb32frame));
svRGB8toRGB32(invert, rgb8fields, rgb32frame, width, height);
```

The **svRGB8toRGB32()** works on 8-bit RGB field data only in the format captured by the IndigoVideo board. Like **svInterleaveFields()**, it will interleave, and, optionally, invert the data (bottom-to-top) if the *invert* parameter is true.

If you don't want to convert the 8-bit RGB data, you can display it on any system by treating the 8-bit pixels as 8-bit color index values. In Example 20-13, the code fragment, from the /usr/people/4Dgifts/examples/dmedia/video/indigovideo program vmirror.c, sets up the IRIS GL color map to display 8-bit RGB data:

#### **Example 20-13** Setting up the IRIS GL Color Map to Display 8-bit RGB

# Chapter 21

# **Handling IndigoVideo Events**

This chapter explains how to handle events for the IndigoVideo board.

# Handling IndigoVideo Events

Programs that use live video need to be notified when the setup or status of the IndigoVideo board changes. This chapter explains how to use either the X Window System or IRIS GL event mechanisms to provide your application with notification of status changes to the IndigoVideo board.

The X and IRIS GL event mechanisms are different, but they serve the same purpose. X event handling methods distinguish between *video activity events* and *parameter change events*; IRIS GL event handling methods lump these event classes together as a single pseudodevice. Wherever possible, X event handling should be used rather than IRIS GL event handling to provide greater portability and flexibility. See the *IRIS IM Programming Guide*, which you can read online using the IRIS InSight viewer, for more details about X event handling versus IRIS GL event handling.

#### In this chapter:

- "IndigoVideo Event Handling Basics" on page 456 provides an overview of video events and the event-handling routines in the IndigoVideo library.
- "X Event Handling" on page 458 describes some event handling methods provided by the X Window System. To learn more about these methods, and to find out about other methods, consult the references recommended at the beginning of this guide.
- "IRIS GL Event Handling" on page 462 describes pure IRIS GL event handling. Skip this section if you are developing a new IndigoVideo application, and use the methods described in "X Event Handling" on page 458 instead.

## **IndigoVideo Event Handling Basics**

Programs receive video activity events only if they use live video, that is, only if they have executed an **svBindWindow()** or **svBindGLWindow()** call.

The four reasons for which the window system generates a video activity event are:

#### video started

This event indicates that video has started in this window. It is generated when your program succeeds in turning on video using **svBindWindow()** or **svBindGLWindow()**.

#### video stopped

This event indicates that video has stopped in this window. It is generated when your program turns off video using **svBindWindow()** or **svBindGLWindow()** with an argument of SV\_IN\_OFF.

#### video busy

Your program tried to turn on video in this window using **svBindWindow()** or **svBindGLWindow()**, but failed because another program had exclusive use of the IndigoVideo board.

#### video preempted

Your program had video running in this window, but it was preempted by another program calling **svBindWindow()** or **svBindGLWindow()**.

These event reasons are identified by global variables, listed in Table 21-1.

 Table 21-1
 Video Activity Event Variable Names

Reason	Variable Name
Video started	SvVideoStarted
Video stopped	SvVideoStopped
Video busy	SvVideoBusy
Video preempted	SvVideoPreempted

The five types of parameter change events are:

active attribute change

This event indicates that a process has given up live video input; for example, by exiting, or by calling **svBindWindow()** or **svBindGLWindow()** with an argument of SV\_IN\_OFF.

signal change

This event is generated whenever a process changes the broadcast standard and/or video mode by calling **svSetParam()**. Changing input sources can also generate this event, as IndigoVideo changes to the default broadcast standard and video mode for the new input source.

video frozen/unfrozen

This event is generated whenever a process freezes or unfreezes video by calling **svSetParam()**.

input source changed

This event is generated whenever a process changes the input source by calling **svSetParam()**.

other parameters changed

This event is generated whenever a process changes any other variable by calling **svSetParam()**.

These event reasons are identified by the global variables listed in Table 21-2.

**Table 21-2** Video Parameter Change Event Variable Names

Reason	Variable Name
Active attribute	SvActiveAttribute
Signal change	SvEncodingAttribute
Video frozen/unfrozen	SvFreezeAttribute
Source change	SvSourceAttribute
Other parameter change	SvParamChangeAttribute

## X Event Handling

To provide X event handling, you must include the appropriate X11 header files, and you must link your program with the X extensions library (-lXext), the X shared library (-lX11\_s), and any X toolkits that you use.

Before your program can receive video-related X events, you must call the **svSelectXEvents()** function.

There are two types of video events, indicated by the following variables:

#### SvVideoActivityEventNumber

Describes video events such as video starting, or video stopping, that affect only a specific window. The *reason* field of the video activity event is set to one of the values listed in Table 21-1. Only programs that use live video receive video activity events.

#### SvParamChangeEventNumber

Describes parameter change events that apply to board settings. The *attribute* field of the param change event is set to one of the values listed in Table 21-2. In addition, the *value* field is set to the new value of the changed parameter (if applicable). In the case of SvEncodingAttribute events, the *value* field is set to one of the values listed in Table 21-3.

 Table 21-3
 Encoding Attribute Values

Variable	Broadcast Standard	Video Mode
SvNTSCComposite	NTSC	Composite
SvNTSCSVideo	NTSC	S-Video
SvPALComposite	PAL	Composite
SvPALSVideo	PAL	S-Video

Example 21-1 contains a listing of *xevents.c*, which demonstrates the use of X events. First, an event mask is set up, to establish interest in exposure, key, and video related events. A connection to the X server is established and a video device is opened with the proper window size for the signal being received. The program prints status messages about the events as they occur.

**Example 21-1** X Event Handling for IndigoVideo events: *xevents.c* 

```
* xevents.c
 * This X11 program displays live video from the IndigoVideo board and shows
 * how to decode X11 video-related event information.
 * Hit the escape or the 'q' keys to exit.
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <svideo.h>
                                /* must be included after <X11/Xlib.h> */
/* We're interested in exposure, key and video-related events */
#define EVENTMASK
                   (ExposureMask | KeyPressMask | StructureNotifyMask)
main(int argc, char *argv[])
    Window
                   rootwin, win;
    Display
                   *display;
    XEvent
                    event;
                    screen, width, height;
    SVhandle
                    V;
    long
                    param[2];
    /* Open connection to X server */
    if ((display = XOpenDisplay(0)) == NULL) {
        fprintf(stderr, "%s: cannot connect to X server", argv[0]);
        if (getenv("DISPLAY") == NULL)
            fprintf(stderr,
                    ", 'DISPLAY' environment variable not set.\n");
        else
            fprintf(stderr, " %s\n", XDisplayName(0));
        exit(1);
    }
    /* Open video device */
    if ((V = svOpenVideo()) == NULL) {
        svPerror("open");
        exit(1);
    }
```

```
/* Determine the window size from the signal standard */
param[0] = SV_BROADCAST;
svGetParam(V, param, 2);
if (param[1] == SV_PAL) {
    width = SV_PAL_XMAX;
    height = SV_PAL_YMAX;
} else {
    width = SV_NTSC_XMAX;
    height = SV_NTSC_YMAX;
printf("Default window size: %d by %d\n", width, height);
/* Create appropriate-sized window */
screen = DefaultScreen(display);
rootwin = RootWindow(display, screen);
win = XCreateSimpleWindow(display, rootwin, 100, 100, width, height,
           5, BlackPixel(display, screen), BlackPixel(display, screen));
/* Set the window and icon names for the window manager before mapping it */
XStoreName(display, win, "X Video Event Handler");
XSetIconName(display, win, argv[0]);
XSelectInput(display, win, EVENTMASK);
XMapWindow(display, win);
/* Associate video with window */
if (svBindWindow(V, display, win, SV_IN_REPLACE) < 0) {</pre>
    svPerror("bindwindow");
    svCloseVideo(V);
    exit(1);
/* Receive video-related X events */
svSelectXEvents(V, display);
/* Event loop */
while (1) {
    XNextEvent(display, &event);
    if (event.type == Expose) {
        printf("Expose event\n");
        if (svBindWindow(V, display, win, SV_IN_REPLACE) < 0) {</pre>
            svPerror("bindwindow");
            svCloseVideo(V);
            exit(1);
        }
```

```
} else if (event.type == KeyPress) {    /* See if we're done */
                  *kev = (XKeyEvent *) &event;
   XKeyEvent
   KeySym
                    keysym;
                    buf[4];
   char
   XLookupString(kev, buf, 1, &keysym, 0);
   printf("Key pressed: '%c' (%d)\n", buf[0], buf[0]);\\
    if (buf[0] == 'Q' \mid | buf[0] == 'q' \mid | buf[0] == '\033') 
        printf("Quitting...\n");
        svCloseVideo(V);
        exit(0);
    }
} else if (event.type == SvVideoActivityEventNumber) {
    SVvideoActivityEvent *ev = (SVvideoActivityEvent *) & event;
    if (ev->reason == SvVideoStarted) {
        printf("Video started\n");
    } else if (ev->reason == SvVideoStopped) {
        printf("Video stopped\n");
    } else if (ev->reason == SvVideoBusy) {
        printf("Video busy\n");
    } else if (ev->reason == SvVideoPreempted) {
       printf("Lost video\n");
    } else {
       printf("unknown video activity (%d)?\n", ev->reason);
} else if (event.type == SvParamChangeEventNumber) {
    SVparamChangeEvent *ev = (SVparamChangeEvent *) &event;
    if (ev->attribute == SvActiveAttribute) {
        /* value always 0 */
        if (svBindWindow(V, display, win, SV_IN_REPLACE) < 0) {</pre>
            svPerror("bindwindow");
            svCloseVideo(V);
            exit(1);
        printf("Active attribute: re-bound video\n");
```

```
} else if (ev->attribute == SvEncodingAttribute) {
                printf("Encoding change: %d = ", ev->value);
                if (ev->value == SvNTSCComposite) {
                    printf("NTSC composite\n");
                } else if (ev->value == SvPALComposite) {
                    printf("PAL composite\n");
                } else if (ev->value == SvNTSCSVideo) {
                    printf("NTSC SVideo\n");
                } else if (ev->value == SvPALSVideo) {
                    printf("PAL SVideo\n");
                } else {
                    printf("?\n");
            } else if (ev->attribute == SvFreezeAttribute) {
                printf("Freeze attribute: %s\n", ev->value ? "on" : "off");
            } else if (ev->attribute == SvSourceAttribute) {
                printf("Input source change: %d\n", ev->value + 1);
            } else if (ev->attribute == SvParamChangeAttribute) {
                printf("Parameter changed\n"); /* value always 1 */
            } else {
                printf("unknown param attribute (%d) ?\n", ev->attribute);
        }
   }
}
```

## IRIS GL Event Handling

Handling IRIS GL events is fairly simple. To receive video-related events, use the IRIS GL **qdevice()** function to queue events from the VIDEO pseudodevice, and use **qread()** to read the events from the queue.

When you get an event from the event queue using the IRIS GL **qread()** function, you must pass the function a pointer to a short integer. **qread()** returns a value indicating the device that generated the event, and fills in the reason for the event in the space pointed to by the argument. In the case of a video event, the reason will correspond to one of the constants listed in Table 21-1 or Table 21-2.

The event loop in Example 21-2 handles video events, printing a message when video is preempted by another process and rebinding the video when it is released by another process.

**Example 21-2** Handling Video Events with IRIS GL Routines

```
long window_id, device;
SVhandle vidnode;
short reason;
/* ... */
qdevice(VIDEO);
qdevice(WINQUIT);
qdevice(WINSHUT);
while(TRUE) {
   device=qread(&reason);
   switch(device) {
      case VIDEO:
             if (reason == SvVideoPreempted)
                 printf("Lost video\n");
             else if (reason == SvActiveAttribute) {
                 svBindGLWindow(vidnode, window_id,
                 SV_IN_REPLACE);
                 printf("Re-bound video\n");
             break;
      case WINSHUT:
      case WINQUIT:
             exit(0);
```

# Chapter 22

# Using the IndigoVideo Utilities

This chapter explains how to use IndigoVideo utilities for creating video shell scripts and for creating movie files from video imported from IndigoVideo captures.

# Using the IndigoVideo Utilities

This chapter briefly describes the utilities and end-user tools that are available for running applications written for the IndigoVideo board. These utility programs provide a convenient way of accessing many of the IndigoVideo board's functions. In addition, you may find them helpful as a reference when debugging your IndigoVideo Library programs.

Two graphical user interface (GUI) tools are available for controlling video: Video Control Panel (*vpanel*) and Video Pro Panel (*vpro*). You can use the Video Control Panel by itself for viewing live video input, grabbing frames of video, and encoding graphics to video, or you can use *vpanel* in conjunction with your IndigoVideo Library programs. As long as your program does not use exclusive-use mode, you can use *vpanel* to switch input sources and control display effects, such as dithering and hue. The Video Pro Panel provides access to IndigoVideo's low-level parameters. All the parameters that can be set using the Video Pro Panel are listed on the *svSetParams*(3V) manual page.

Live video windows for input (*videoin*) and output (*videoout*) can be launched separately from a command line or from *vpanel*. See the *videoin*(1) and *videout*(1) man pages for details.

See the *Media Control Panels User's Guide*, which is accessible from the InSight viewer, for complete instructions on using the Video Control Panel, Video Pro Panel, and the live video windows. Be sure to read the *Svideo Release Notes* for important information about software updates and notes about special situations and workarounds.

This chapter explains how to use two other utilities: *svcmd*, an interactive shell-level tool that lets you access IndigoVideo Library routines from a shell command line or a shell script, and *svtomovie*, a program that lets you turn video input into movie files that you can play using the Movie Maker tool.

#### In this chapter:

- "Using svcmd, the IndigoVideo Shell-level Tool" explains how to use *svcmd*. The IndigoVideo shell-level tool provides a command-line interface to most of the functions of the IndigoVideo Library, including single-frame output (but not including DMA). You can use the shell-level tool to control IndigoVideo functions from a shell script.
- "Making a Movie File from IndigoVideo and Audio Input" on page 469
  explains how to use *svtomovie* to create movies that can be viewed with
  Movie Player.

# Using svcmd, the IndigoVideo Shell-level Tool

You can use the shell-level tool, *svcmd*, to control the IndigoVideo board from the command line. You can also use *svcmd* in shell scripts to automate repetitive tasks. You can use the shell-level tool in one of two ways: either by specifying a single operation on the command line, or by using *svcmd*'s interactive mode.

To get a listing of *svcmd* commands, enter:

```
svcmd -h
```

To execute a single command, use the syntax:

```
svcmd command [parameters]
```

For example, to set the input source to 1, enter:

```
svcmd inputsource 1
```

To start *svcmd* in interactive mode, use the **-i** flag. To exit *svcmd*, use the *quit* or *exit* command. For example, to set the input source to 2 and the video mode to composite, type:

```
svcmd -i
svcmd >> inputsource 2
svcmd >> videomode comp
svcmd >> quit
```

For more information on *svcmd*, see the *svcmd*(1) manual page.

# Making a Movie File from IndigoVideo and Audio Input

You can use IndigoVideo input and optional audio input to make a movie. Movies are files that can be played on the Indigo workstation from the Movie Player tool. The *svtomovie* program provides everything you need to make a movie using your IndigoVideo board, audio input if you wish, and a little imagination.

To start *svtomovie* from a shell command line, enter:

svtomovie [options] filename

to which the system replies:

svtomovie: Press <ENTER> when you are ready to collect video:

You must now "queue" the video source, that is, advance it to the location where the recording is to begin. You can use *vpanel* to set up the video source and *apanel* to set up the audio source—*svtomovie* responds to control from both of these tools.

When you press <**Enter**>, *svtomovie* collects video, and optionally audio, and writes it to the file you specified in *filename*.

When *svtomovie* finishes capturing the video (and audio) frames, it rewrites the movie file to a playable format. You can play your new movie with Movie Player or edit it with Movie Maker.

The *svtomovie* options are:

- -a Turn audio off; audio is on by default. This makes a silent movie and has the possible advantage of capturing more video data, because the audio capturing is disabled and is not using CPU bandwidth.
- -b Use burst mode for capture; continuous mode is the default. Burst mode is useful to get short bursts of 30 frames per second. It captures only 30 frames per second, and can capture directly to only a maximum 8MB memory buffer; hence, there is no audio, and, depending on the chosen size, a limited number of frames.
- **-d** Turn diagnostic messaging on; diagnostics are off by default. See *svtomovie*(1) for a list of diagnostics.

<b>-f</b> framerate	Specifies movie frame rate; the default rate is 15 frames per second. This option is ignored if <b>-b</b> is used. In continuous capture mode, the only legal values for this parameter are 15, 10, 6, 5, 3, 2, and 1. All other values will gather data at one of these rates but movie playback will be at the requested rate. In burst capture mode, the only legal value is 30. All other values will be ignored.
-m	Turn audio monitoring on; audio monitoring is off by default. This option is ignored if <b>-a</b> or <b>-b</b> is used.
-n numframes	Specifies number of frames of movie; 100 is the default.
-s	Use stereo audio input; mono is the default. This option is ignored if <b>-a</b> or <b>-b</b> is used.
-w width	Specifies the video width in pixels; 320 is the default. Height is chosen to preserve 8:6 aspect ratio.

#### **PART FIVE**

# **Compression Programming**

Chapter 23, "Introduction to the Compression Library," introduces the CL and describes its applications and features. It provides basic background information on compression technology and on digital audio and video data formats.

Chapter 24, "Getting Started with the Compression Library," describes how to use the three types of interfaces supplied by the CL.

Chapter 25, "Using Compression Library Algorithms and Parameters," explains how to use the CL algorithms and global parameters.

Chapter 26, "Customizing the Compression Library," explains how to add your own algorithms and parameters to the CL.

# Chapter 23

# Introduction to the Compression Library

This chapter introduces the Compression Library, which provides an algorithm-independent, extensible interface to image, audio, video, and movie compression.

# Introduction to the Compression Library

The Compression Library, *libcl.so*, provides a flexible, extensible, and algorithm-independent software interface for compressing and decompressing audio, video, and image data. Developers may also choose to incorporate the licensable built-in interface to third-party audio compression software from Aware, Inc., which is described in Appendix B, "Aware Scalable Audio Compression Software."

Using the Compression Library (CL) involves three concepts, each of which are discussed in a separate chapter in this part of this guide:

- using the application interface (API)
- using algorithms and parameters
- using/adding algorithms

#### In this chapter:

- "Overview of the Compression Library" on page 475 describes the features and applications of the CL and provides fundamental information essential for working with compression.
- "Compression Library Data Formats" on page 483 describes the data formats that you are likely to encounter when using the CL.

# Overview of the Compression Library

Compression is the process of shrinking the size of the data without changing its content significantly. Compact data can be stored more efficiently and can be transmitted faster than raw data. For example, certain compression methods can allow you to store 10 to 20 times as many compressed images in the space required to store a single uncompressed image. Compression extends the capabilities of digital media delivery and storage systems because it encodes data more efficiently.

# **Compression Library Applications**

Compression Library applications are far-reaching. The primary goal of the CL is to improve the data delivery and storage capabilities of applications that use digital media.

The Compression Library can be used with the Audio File Library, and data used by the IRIS MediaMosaic<sup>TM</sup> tools, Movie Player and Movie Maker. Other applications include:

- Information delivery and storage, including multimedia presentations, publications, interactive training, archiving, and annotation. For example, you can use MoviePlayer as the playback mechanism for an information delivery application. Showcase™ can be used as the base medium from which to launch separate executables of the MoviePlayer to play back prerecorded movies.
- Telecommunications (video/voice mail, phone, and teleconferencing)
   Compression allows faster transmission of data. This is especially useful when the data rate is limited by the transmission medium. Cost savings can also be realized when transmitting data over a medium where you are billed on the basis of either access time or number of bytes transferred.
- Animation previewing

Images can be compressed frame-by-frame, as they are rendered, for previewing 2D and 3D graphics animations in live action before recording to video tape. Previewing saves time for animators because they don't have to render and record a full-data animation to tape every time they want to check the motion sequence.

Movie (audio and video) editing

Movie editing can be done entirely in the digital domain using a tool such as MovieMaker, instead of editing a tape recording. Compression lets you store more data and decompress it as you open files for editing.

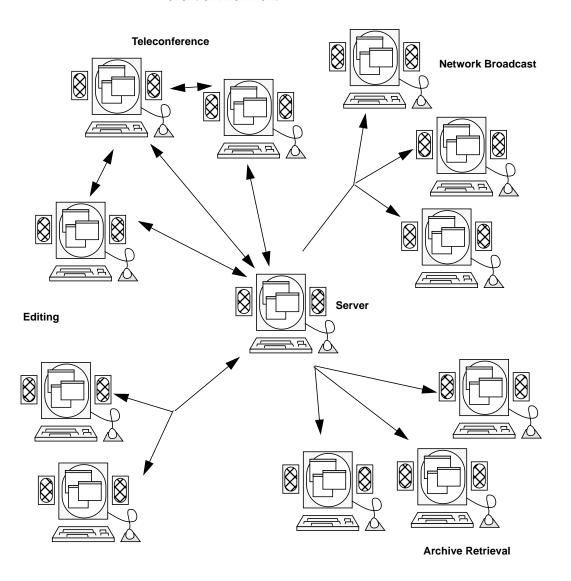


Figure 23-1 shows a few of the applications that are possible in a server-client environment.

**Figure 23-1** Server-Client Compression Applications

## **Compression Library Features**

The Compression Library features:

- algorithm independence
- hardware independence
- support of industry standard algorithms
- support of Silicon Graphics proprietary algorithms
- binary compatibility across Silicon Graphics platforms

Because the CL is algorithm-independent, you need to know only the basic application interface (API) to use any of the supplied algorithms. You can query the library for the available algorithms, and you can add your own algorithms to the library. A pass-through capability allows you to pass data through the routines without using an algorithm.

The *libcl* API provides facilities for working with audio, still images, sequential frames of data (movies), and a buffering mechanism for random access of compressed data.

The buffering facility allows independent buffering of compressed data and decompressed frames, with synchronous or asynchronous access, either external or internal to the library. Separate processes can be used for supplying data, compressing/decompressing, and retrieving data.

The API also uses a set of global state parameters, similar to those found in the Audio Library, *libaudio*, to establish and manipulate compression attributes.

## **Compression Library Basics**

This section introduces compression technology and compression standards. It provides useful background information that you should know before using the Compression Library.

#### **Lossy versus Lossless Compression Methods**

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. A *lossy* compression method does not preserve 100% of the information in the original method. Some methods incur more loss than others, so the amount of loss that can be tolerated by your application might affect your decision about which compression method to use.

**Note:** In general, video compression algorithms are designed to work on camera-generated images. Computer-generated images often contain text and line drawings that compression algorithms can't compress as well as smooth-shaded computer images, which approximate camera video.

#### **Compression Standards**

Standards provide a common ground for developers to share technology. Standards for the audio and video industries are constantly being developed and changed in response to new technology. The Compression Library supports these standards through the use of algorithms and parameters.

#### **Compression Library Algorithms**

Algorithms are provided within *libcl* for audio and video standards and for Silicon Graphics proprietary algorithms that have significant benefits. You can query the library for the available algorithms, and you can add your own algorithms to the library. Algorithms are grouped according to the type of data they operate on: still images, motion video, or audio.

#### **Still Image Algorithms**

Although any algorithm can be used for still 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 a compression standard for compressing full-color or grayscale digital images. JPEG is most useful for still images; it is usable, but slow when performed in software, for video. You can use the Cosmo Compress

option, a hardware JPEG accelerator, in conjunction with the Compression Library for compressing video to and decompressing video from memory or for compressing to and decompressing from a special video connection to Galileo Video, IndyVideo, or Indigo<sup>2</sup> video.

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.

The amount of compression and the quality of the resulting image are independent of the image data. The quality depends on the compression ratio. The Compression Library lets you select the compression ratio that best suits your application needs.

JPEG is designed for still images and is usable, but slow, for video. 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.

#### **Movie Algorithms**

For the best quality in a final movie, all image manipulation and storage should be with uncompressed images until the final movie is produced, at which time the images can be compressed. Repeatedly compressing, decompressing, and then recompressing images reduces the image quality.

The Compression Library supports the following algorithms for motion video compression/decompression:

#### CL\_MPEG\_VIDEO

Moving Pictures Expert Group is a standard that is designed for extreme compression of motion video while maintaining high image quality. It is a lossy algorithm that is capable of producing higher compression ratios than both JPEG and MVC1.

MPEG I is designed to give the best possible quality for a 1.2 million bits per second (Mbps) data rate for audio as well as video data. Other data rates are possible.

The quality depends on the sophistication of the encoder. Quality (subjectively evaluated) between VHS and S-VHS can be achieved for images whose frame size is  $352 \times 240$  with the 1.2 Mbps data rate, which is possible to obtain from a CD-ROM in real time.

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.

The compressed data stream is designed so that the video can be played forward or backward. This makes MPEG well suited for video publishing, where a video is compressed once and decompressed many times for playback.

CL MVC1

Motion Video Compressor 1 is a Silicon Graphics proprietary algorithm that is a good general-purpose compression scheme. It is a color-cell compression technique that works well for video, but can cause fuzzy edges in high-contrast animation. MVC1 is a fairly lossy algorithm that does not produce compression ratios as high as JPEG, but it is well suited to movies.

CL\_MVC2

Motion Video Compressor 2 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.

CL\_RLE

8-bit Run Length Encode is a lossless algorithm for compressing 8-bit RGB. It is the only algorithm currently available to directly compress 8-bit RGB data (CL\_RGB332). 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. The *libcl* implementation of RLE does not use a standard RLE method. This is a lossless compression method that uses run-length encoding (RLE). Run-length encoding compresses images by storing a color and its run-length (the number of pixels of that color)

every time the color changes. It is a good technique for animations where there are large areas that have identical colorsRun-length encoding replaces 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 repeat count representing the number of subsequent pixels of the same value.

CL\_RLE24 24-bit Run Length Encode is a lossless algorithm for

compressing 24-bit RGB.

CL\_RTR1 Real Time Record is a Silicon Graphics proprietary algorithm

designed for recording directly from a camera or VTR to disk or digital audio tape (DAT) by compressing on the fly. The quality achieved is dependent upon the processor performance and video hardware that is available.

#### **Audio Algorithms**

The Compression Library supports two audio algorithms that are based on international standards:

#### CCITT/TSB G.711 μ-law

compresses 16-bit audio to 8-bit audio using a geometric function that takes advantage of the fact that human hearing is more sensitive to differences at lower volume levels. It is designed for rapid compression and decompression at a 2:1 compression ratio.

#### CCITT /TSB G.711 A-law

compresses 16-bit audio to 8-bit audio using a different geometric function that takes advantage of the fact that human hearing is more sensitive to differences at lower volume levels. It is designed for rapid compression and decompression at a 2:1 compression ratio.

In America,  $\mu$ -law compression is generally used. In Europe, A-law is more prevalent.

# **Compression Library Data Formats**

This section provides a brief introduction to digital media formats. It describes the fundamental nature of digital data and introduces some basic terminology that you should know before using the Compression Library.

Many different formats exist for audio, image, and video data. The Compression Library supports the most common formats, but it doesn't restrict you to using one of these formats. In fact, you can define your own unique format to suit your application needs. For example, you can define a file format that contains interleaved frames of audio and video for a movie application, or you can define a file format that contains multiple tracks of audio data for an audio-mixing application.

The following sections describe some of the data formats you are likely to encounter when developing applications that use the Compression Library.

#### **Audio Data Formats**

Audio data occurs in a *stream*, which can be divided into units called *blocks*. Audio data can be *monaural* (mono), which has one channel embedded in the audio stream, or *stereo*, which has two channels embedded in the audio stream. In a stereo audio stream, the left and right channels are interleaved. The Compression Library provides support for both mono and stereo audio. Parameters are used to distinguish between the two data types.

Depending on the original source of the audio, it may have other distinguishing characteristics such as the resolution. See Part II, "Digital Audio and MIDI Programming," for more information about audio data formats

#### Image Data Formats

Image data is contained in a frame. You need to supply the height and width of an image frame when using the *libcl* routines that compress/decompress image and video data. The ordering of pixels within the frame depends upon the source of the data. Top-to-bottom is the default data orientation for

Compression Library routines. You can use the CL\_ORIENTATION parameter to specify how pixels are ordered.

The Compression Library works with data that is contained in frames. A frame is defined as a sample at one instant of time so that:

1 audio sample: mono 8 bit = 1 byte mono 16 bit= 2 bytes stereo 8 bit = 2 bytes stereo 16 bit = 4 bytes

1 video frame: *width* \* *height* \* *components* \* *bitsPerComponent/*8 = *n* bytes

#### **Video Data Formats**

Video data is a stream of sequential frames of image data. Some video formats have special frames called *keyframes* that contain information for a block of frames that is treated as a single unit. There are a variety of video formats. The Compression Library supports a set of formats for all algorithms.

Video data can be either color or black-and-white. If you are working with video data, you should be familiar with such terms as component video, composite video, chrominance, luminance, and RGBA data.

Implicit *color space conversion* occurs whenever the specified original format does not match the specified internal format, that is, the format that is compressed directly. Conversion from the original format to the internal format occurs on compression, and conversion from the internal format to the original format occurs on decompression. A different original format can be used on decompression than was used on compression.

**Note:** The parameter CL\_BEST\_FIT can be used when compressing to automatically choose the best internal format for a given original format.

The Compression Library supports these video formats:

CL\_RGBA R, G, B, and A data are 8-bit components packed into the 32-bit word as:

0xAABBGGRR

where:

AA contains the 1-byte alpha value. BB contains the 1-byte blue value. GG contains the 1-byte green value. RR contains the 1-byte red value.

RGBA component values range from 0 to 0xFF (255). For this format, *compressionFormat.components* should be set to

CL\_RGBX

R, G, B, and X (don't care) data are packed into the 32-bit word as for CL\_RGBA. Note that with this format, only the R, G, and B values are compressed.

CL\_RGB

R, G, and B data are packed into a 24-bit word. Note that with this format, the RGB triplets may cross the 32-bit word boundaries.

CL\_RGB332

R, G, and B data are packed into an 8-bit byte as:

0xrrrbbggg

where:

rrr is three bits of red. bb is two bits of blue. ggg is three bits of green.

## CL\_GRAYSCALE

Four 8-bit luminance bytes are packed in a 32-bit word.

CL\_Y Equivalent to CL\_GRAYSCALE.

CL\_YUV

Three 8-bit components, Y, U, and V, are packed into 24 bits

as

0xUUYYVV

where:

UU contains the chroma-blue value. YY contains the luminance value. VV contains the chroma-red value.

CL\_YCbCr

A synonym for YUV<sup>1</sup>. Y is for luminance, Cb (chroma-blue), and Cr (chroma-red) are for chroma.

CL\_YUV422

Two luminance components are packed into a 32-bit word with one U-V pair. In other words, the chroma components are sampled with half of the horizontal rate of the luma, which is known as 4:2:2 sampling. Two pixels are represented by this 32-bit word as (Y1, U1, V1) and (Y2, U1, V1). The order of the components is:

0xU1Y1V1Y2

where:

U1 contains the chroma-blue value. Y1 contains the first luminance value. V1 contains the chroma-red value. Y2 contains the second luminance value.

#### CL\_YUV422DC

(duplicate chroma) The chroma is subsampled by 2 vertically in addition to horizontally, and is packed the same as CL\_YUV422, except that U and V are duplicated on the odd lines. CL\_IMAGE\_WIDTH must be even when using this format. This format is convenient for storing 4:1:1 sampled data, which is analogous to 4:2:2 sampling with the addition of half-sampling of the chroma vertically. Sometimes 4:1:1 is used to indicate full vertical and one-quarter horizontal sampling.

The video specification of YUV and YCbCr dictates a scale factor for each component when converting between these formats. For convenience, the CL defines them as equal.

Table 23-1 shows the formats that are supported directly, that is, formats that do not require color conversion—for each algorithm that is currently implemented in *libcl*.

**Table 23-1** Video Formats Not Requiring Color Conversion

Algorithm	Format
UNCOMPRESSED	Any format
JPEG	CL_YUV and CL_GRAYSCALE
MVC1	CL_RGBX and CL_GRAYSCALE
MPEG	CL_YUV422DC
RLE	CL_RGB332
RTR1	CL_YUV,CL_YUV422,CL_YUV422DC,and CL_GRAYSCALE

## **Movie Data Formats**

The Compression Library supports the movie formats used by the Movie Maker and Movie Player tools.

#### **Header Formats**

Sometimes data is prefaced by a header that contains information about the data. The CL provides routines for extracting header information, which can also contain CL state parameters.

A typical header begins with a start code and a size:

```
Header Start Code
Header size (in bytes)
```

followed by parameter-value pairs such as those listed in Table 23-2.

 Table 23-2
 Parameters Contained in Header Data

Parameter	Information Supplied
CL_ALGORITHM_ID	Algorithm scheme
CL_ALGORITHM_VERSION	Version of the algorithm
CL_INTERNAL_FORMAT	Format of images immediately before compression
CL_NUMBER_OF_FRAMES	Number of frames in the sequence
CL_FRAME_RATE	Frame rate
CL_IMAGE_WIDTH	Width (image and video data only)
CL_IMAGE_HEIGHT	Height (image and video data only)

Other parameters are possible, see Chapter 25, "Using Compression Library Algorithms and Parameters," for a complete list of parameters available.

# Chapter 24

# Getting Started with the Compression Library

This chapter describes how to use the three interfaces that exist in the CL:

- still image, for compressing individual images
- sequential, for compressing data streams
- buffered, for independent compression and decompression

# Getting Started with the Compression Library

This chapter describes how to use the Compression Library API to compress and decompress image, audio, and video data. The Compression Library API has three basic interfaces:

- still image access
- sequential frame access
- buffered random frame access

In addition, the Compression Library supports Cosmo Compress, an optional hardware JPEG video codec, which connects to the Galileo family of video devices to allow real-time JPEG video capture and playback.

#### In this chapter:

- "Overview of the Compression Library API" on page 492 describes the three types of interfaces provided by the CL and explains how to use them. It also describes the CL error handling facility.
- "Using the Still Image Interface" on page 494 explains how to compress still images with a single call.
- "Using the Sequential Frame Interface" on page 497 explains how to compress or decompress sequential data using a compressor or decompressor.
- "Using the Buffering Interface" on page 506 explains how to use internal or external buffering to implement random access or multithreaded compression or decompression applications.
- "Programming with the Cosmo Compress JPEG Codec" on page 521 explains how to add Cosmo Compress support to your application.

# Overview of the Compression Library API

This section describes how each type of interface is used and provides error handling information.

## Still Image API

The single image method is designed to make still image compression as simple as possible. It is the simplest, yet most limited of the three. It consists of two calls, one for compression and one for decompression. No interframe compression/decompression, such as the method that takes advantage of similarities between frames in MPEG, is possible with this interface.

## **Sequential Access API**

The sequential interface is designed for audio/video-streaming applications where the input is live, or where there is no control over playback and when the amount of compressed data for each frame is known in advance; in fact, an error is reported if insufficient data is passed. This interface is more complex, requiring a series of compress or decompress calls to be encapsulated within an open-close block. Each compressor or decompressor keeps state information appropriate to the selected compression algorithm in parameters that you can query and set.

#### **Buffered Access API**

The buffered interface is designed for:

- VCR-like control over the audio/video stream
- maximum efficiency by buffering compressed data and uncompressed frames
- blocking and nonblocking access
- transparent buffering for hardware acceleration or for multiprocessor operation
- multithreaded applications

This interface includes the calls of the sequential interface, plus buffermanagement routines to access the compressed data and the uncompressed frame buffers.

The buffer management routines allow blocking and nonblocking access and accumulation of compressed data and decompressed frames. The compression or decompression modules can each be placed in separate processes. Separating the processes allows the compression or decompression process to get ahead a few frames, which is advantageous for algorithms such as MPEG, which compress the data using techniques that take advantage of similarities between frames, and it also facilitates hardware acceleration.

## About File I/O and Error Handling

In the CL, file I/O is handled by the caller. The CL has an error handler that prints error messages to *stderr*. Most CL routines return a negative error code upon failure.

You can override the default error handling routine and establish an alternate compression error handling routine using **clSetErrorHandler()**.

The function prototype for **clSetErrorHandler()** is:

```
CL_ErrFunc clSetErrorHandler(CL_ErrFunc efunc)
```

#### where:

*efunc* is a pointer to an error handling routine declared as:

void ErrorFunc(CLhandle handle, int code, const

char\* fmt,...)

The returned value is a pointer to the previous error handling routine.

The code fragment in Example 24-1 demonstrates how to silence error reporting for a section of code.

**Example 24-1** Using a Custom Error Handling Routine

# **Using the Still Image Interface**

A simple interface exists for compressing or decompressing still images with a single call. To compress a still image, use **clCompressImage()**, which compresses the data from the specified *frameBuffer*, stores the compressed image in *compressedData*, and stores its resulting size in *compressedBufferSize*.

Pass to **clCompressImage()** the compression scheme; the width, height, and format of the image; the desired compression ratio; pointers to reference the buffer containing the image and the buffer that is to store the compressed data; and a pointer to return the size of the compressed data.

You should allocate a buffer large enough to store the compressed data. In most cases, a buffer the size of the source image plus the maximum header size, which you can get by calling **clQueryMaxHeaderSize()**, is sufficient. When calculating the data storage of the source image, you can use the CL macro **CL\_BytesPerPixel()** to determine the number of bytes per pixel for certain packing formats.

The function prototypes for the compress and decompress image routines are:

#### where:

compressionScheme

is the compression or decompression scheme to use.

width is the width of the image.height is the height of the image.

originalFormat is the format of the original image to (de)compress. For

video, use CL\_RGB, CL\_RGBX, CL\_RGBA, CL\_RGB332,

CL\_GRAYSCALE, CL\_YUV, CL\_YUV422, or CL\_YUV422DC. For audio, use CL\_MONO or

CL\_STEREO\_INTERLEAVED.

#### compressionRatio

is the target compression ratio. The resulting quality depends on the value of this parameter and on the algorithm that is used. Use 0.0 to specify a nominal value. The nominal values for some of the algorithms are:

MVC1 5.3:1 JPEG 15.0:1 MPEG 48.0:1

*frameBuffer* is a pointer to the frame buffer that contains the

uncompressed image data.

#### compressedBufferSizePtr

is a pointer to the size, in bytes, of the compressed data buffer. If it is specified as a nonzero value, the size indicates the maximum size of the compressed data buffer. The value pointed to is overwritten by **clCompressImage()** when it returns the actual size of the compressed data.

compressedBufferSize

is the size of the compressed data in bytes.

compressedBuffer

is a pointer to the compressed data buffer.

Use **clDecompressImage()** to decompress an image. **clDecompressImage()** decompresses the data that is stored in *compressedBuffer*, whose size is *compressedBufferSize*, and stores the resulting image in *frameBuffer*.

The values of the state parameters that are used in conjunction with the other compression library calls have no effect on these routines, but their defaults do. The arguments *width*, *height*, *originalFormat*, and *compressionRatio* function the same as the state parameters by the same names but are given as direct arguments to facilitate the single-command interface.

Example 24-2 demonstrates how to compress and decompress a color image using the JPEG algorithm. The image is 320 pixels wide by 240 pixels high and its data is in the RGBX format.

#### **Example 24-2** Compressing and Decompressing a Single Frame

# **Using the Sequential Frame Interface**

This section describes how to work with sequential frames of audio or video data. See "Using the Buffering Interface" on page 506 for a description of how to work with nonsequential data, or for situations where the decompression rate is different from the compression rate.

## **Compressing a Sequence of Frames**

To compress sequential data and audio/video streams, use a *compressor*. A compressor is an abstraction that modularizes compression operations.

To compress a sequence of frames:

- 1. Open a compressor to establish the beginning of a sequence of compression calls.
- 2. Compress frames one at a time, storing the compressed data after each frame has been compressed.
- 3. Close the compressor to deallocate the resources associated with that compressor.

Each of these steps is discussed in detail in the following sections.

## **Opening a Compressor**

Call **clOpenCompressor()** to open a compressor for a given algorithm. Its function prototype is:

int clOpenCompressor(int scheme, CLhandle \*handlePtr)

#### where:

*scheme* is the compression scheme to use.

handlePtr is a pointer, which is overwritten by the returned handle of

the compressor that is used by subsequent calls to identify

the compressor.

More than one compressor can be open at a time. Use the handle that is returned in *handle* to identify a specific compressor.

#### Compressing Frames

After a compressor has been opened, call **clCompress()** to compress the data. Pass to **clCompress()** the handle returned by **clOpenCompressor()**, the number of frames to be compressed, and pointers to reference the frame buffer containing the data frames, the size of the data, and the location of the buffer that is to store the compressed data.

#### The function prototype for **clCompress()** is:

where:

handle

is a handle to the compressor

numberOfFrames

is the number of frames to compress: generally 1 for video data, an appropriate block size for audio data, or either CL\_CONTINUOUS\_BLOCK or

CL\_CONTINUOUS\_NONBLOCK in order to continue compression until either the frame buffer is marked done or clCloseCompressor() is called. With

CL\_CONTINUOUS\_NONBLOCK, the call to **clCompress()** returns immediately while the compression occurs in a separate thread; CL\_CONTINUOUS\_BLOCK blocks until compression is completed.

frameBuffer

is a pointer to the location of the buffer that contains the data that is to be compressed. Using a NULL argument here invokes the buffered interface that is described in "Using the Buffering Interface" on page 506. An error is reported if no buffer exists. Some compressors allow a value of CL\_EXTERNAL\_DEVICE, indicating a direct connection to an external audio or video source.

#### compressedDataSize

is a pointer to the returned size of the compressed data in bytes.

compressedBuffer

is a pointer to the location where the compressed data is to be written. Using a NULL argument here invokes the buffered interface that is described in "Using the Buffering Interface" on page 506.

Call **clCompress()** once to compress *numberOfFrames* sequential frames. **clCompress()** reads the raw data from the location pointed to by *frameBuffer* and writes the compressed data to the location pointed to by *compressedBuffer*. **clCompress()** returns either the number of frames successfully compressed, or in the case of CL\_CONTINUOUS\_NONBLOCK, returns SUCCESS immediately.

The size of the compressed data is stored in *compressedDataSize*, even if this size exceeds the COMPRESSED\_BUFFER\_SIZE state parameter. If COMPRESSED\_BUFFER\_SIZE is less than the actual size returned by **clCompress()**, then the data returned in *compressedBuffer* is not complete.

An application-allocated compressed buffer must be at least COMPRESSED\_BUFFER\_SIZE bytes. This parameter should be determined by calling clGetParams() after the frame buffer dimensions are defined by clSetParams(). It is not required to set the COMPRESSED\_BUFFER\_SIZE, because the default is the largest possible compressed data size, which is computed from the given parameters.

#### **Closing a Compressor**

To close a compressor, call **clCloseCompressor()** with the handle of the compressor you wish to close. This frees resources associated with the compressor.

The code fragment in Example 24-3 demonstrates how to compress a series of frames using the CL\_MVC1 algorithm. A compressor is opened, then a compression loop is entered, where frames are accessed one at a time and compressed using the selected algorithm, then written to a data buffer. The compressor is closed when all of the frames have been compressed.

## **Example 24-3** Compressing a Series of Frames

```
#include <dmedia/cl.h>
int pbuf[][2] = {
    CL_IMAGE_WIDTH,
    CL_IMAGE_HEIGHT, 0,
    CL_COMPRESSED_BUFFER_SIZE, 0
};
/* Compress a series of frames */
clOpenCompressor(CL_MVC1, &handle);
/* set parameters */
pbuf[0][1] = 320;
pbuf[1][1] = 240;
clSetParams(handle, (int *)pbuf, 4);
/* allocate the required size buffer */
clGetParams(handle, (int *)pbuf, 6);
compressedBuffer = malloc(pbuf[2][1]);
for(i = 0; i < numberOfFrames; i++)</pre>
    /* Get a frame from somewhere */
    clCompress(handle, 1, frameBuffer, &compressedBufferSize,
        compressedBuffer);
    /* Write the compressed data to somewhere else. */
clCloseCompressor(handle);
```

## **Decompressing a Sequence of Frames**

Decompressing sequential data and audio/video streams requires the use of a *decompressor*. A decompressor is an abstraction that modularizes decompression operations.

To decompress a sequence of frames:

- 1. Query the stream header to get the compression scheme used.
- 2. Open a decompressor to establish the beginning of a sequence of decompression calls.

- 3. Decompress frames one at a time, storing the decompressed data after each frame has been decompressed.
- 4. Close the decompressor to deallocate the resources associated with that decompressor.

Each of these steps is discussed in detail in the following sections.

## **Getting Stream Information**

To determine which *scheme* to pass to the decompressor, use **clQueryScheme()** to get the *scheme* from the 16 bytes of the stream header (see Table 24-1 for a list of typical header contents, and Table 24-2 for a list of additional video stream header contents). **clQueryScheme()** returns the scheme, or the (negative) error code when an error occurs.

Once you determine the scheme, you can open the decompressor and read the header using <code>clReadHeader()</code>, which returns the actual size of the header, or zero if none is detected. Use <code>clQueryMaxHeaderSize()</code>, which returns the maximum size of the header, or zero if none is detected, to determine the size of the header to send to <code>clReadHeader()</code>. You should free the space used for the header buffer when you are finished with it.

**clReadHeader()** is generally called before **clCreateBuf()** to help calculate the compressed buffer size. It uses the data passed to it without affecting the buffering. **clReadHeader()** also sets up any state parameters that can be determined from the header.

## The function prototypes are:

```
int clQueryScheme(void *header)
int clQueryMaxHeaderSize(int scheme)
int clReadHeader(CLhandle handle, int headerSize,void *header)
```

#### where:

header is a pointer to a buffer containing at least 16 bytes of the

header.

scheme is the decompression scheme to use.

handle is a handle to the decompressor.

headerSize is the maximum size of the header in bytes.

header is a pointer to a buffer containing the header.

A typical header begins with a start code and a size, followed by parameter-value pairs such as those listed in Table 24-1.

**Table 24-1** Typical Stream Header Contents

Parameter	Information supplied
CL_ALGORITHM_ID	Algorithm scheme
CL_ALGORITHM_VERSION	Version of the algorithm
CL_INTERNAL_FORMAT	Format of images immediately before compression
CL_NUMBER_OF_FRAMES	Number of frames in the sequence
CL_FRAME_RATE	Frame rate

In addition, video algorithms usually supply the width and height parameters listed in the header, as shown in Table 24-2.

 Table 24-2
 Additional Video Stream Header Contents

Parameter	Information Supplied
CL_IMAGE_WIDTH	Width
CL_IMAGE_HEIGHT	Height

The code fragment in Example 24-4 demonstrates how to query a stream header and read its contents.

**Example 24-4** Getting the Decompression Scheme from a Header

```
#include <cl.h>
...
int decompressionScheme;
...
/*
 * Determine the scheme from the first 16 bytes of the
 * header(from the beginning of video data)
*/
```

## **Opening a Decompressor**

Call **clOpenDecompressor()**, with the desired compression scheme and a pointer for returning a handle, to open a decompressor for a given algorithm. Its function prototype is:

```
int clOpenDecompressor(int scheme, CLhandle *handlePtr)
```

#### where:

scheme is the decompression scheme to use

handlePtr is a pointer to the returned handle of the decompressor that

is used by subsequent calls to identify the decompressor.

More than one decompressor can be open at a time. Use the handle that is returned in *handle* to identify a specific decompressor.

#### **Decompressing Frames**

After a decompressor has been opened, call **clDecompress()** to decompress the data. Pass to **clDecompress()** the handle returned by

**clOpenDecompressor()**, the number of frames to be decompressed, the size of the data, and pointers to reference the decompressed data and the frame buffer that contains the compressed frames.

## The function prototype for **clDecompress()** is:

where:

handle

is a handle to the decompressor.

#### numberOfFrames

is the number of frames to decompress: generally 1 for video data, an appropriate block size for audio data, or either CL\_CONTINUOUS\_BLOCK or CL\_CONTINUOUS\_NONBLOCK in order to continue decompression until either the frame buffer is marked done or clCloseDecompressor() is called. With CL\_CONTINUOUS\_NONBLOCK, the call to clDeCompress() returns immediately while the compression occurs in a separate thread; CL\_CONTINUOUS\_BLOCK blocks until compression is completed. Using a NULL argument invokes the buffered interface that is described in "Using the Buffering Interface" on page 506.

#### compressedDataSize

is a pointer to the returned size of the decompressed data in bytes.

#### compressedData

is a pointer to the location where the decompressed data is to be written.

## frameBuffer

is a pointer to the location of the frame buffer that contains the data that is to be decompressed. Some compressors allow a value of CL\_EXTERNAL\_DEVICE, indicating a direct connection to an external audio or video source. Using a NULL argument invokes the buffered interface that is described in "Using the Buffering Interface" on page 506. An error is reported if no buffer exists.

## **Closing a Decompressor**

To close a decompressor, call **clCloseDecompressor()** with the handle of the decompressor you wish to close.

The code fragment in Example 24-5 demonstrates how to decompress a series of  $320 \times 240$  (32-bit) RGBX frames by using the CL\_MVC1 algorithm. A decompressor is opened, then a decompression loop is entered, where frames are accessed one at a time and decompressed by using the selected algorithm, then written to a location such as the screen. The decompressor is closed when all of the frames have been compressed.

### **Example 24-5** Decompressing a Series of Frames

```
#include <cl.h>
int compressedBufferSize;
int compressedBuffer[320][240], frameBuffer[320][240];
int width, height, k;
static int paramBuf[][2] = {
   CL_IMAGE_WIDTH, 0,
    CL_IMAGE_HEIGHT, 0,
    CL_ORIGINAL_FORMAT, 0,
};
width = 320;
height = 240;
clOpenDecompressor(CL_MVC1, &decompressorHdl);
paramBuf[0][1] = width;
paramBuf[1][1] = height;
paramBuf[2][1] = CL_RGBX;
clSetParams(decompressorHdl, (int *)paramBuf,
             sizeof(paramBuf) / sizeof(int));
for (k = 0; k < numberOfFrames; k++)</pre>
{ /* Decompress each frame and display it */
 dataSize = GetCompressedVideo(k, frameSize, data);
  clDecompress(decompressorHdl, 1, dataSize, data,
             frameBuffer);
  lrectwrite(0, 0, width-1, height-1,
             (unsigned int *)frameBuffer);
/* Close Decompressor */
clCloseDecompressor(decompressorHdl);
```

# **Using the Buffering Interface**

Buffers are used to manage compression and decompression for data that is accessed randomly, or when it is necessary to separate the task into several processes or across multiple processors. Buffering allows the accumulation of compressed data to be independent of that of decompressed frames. The buffering interface can be used for multithreaded applications.

Buffers are implemented as ring buffers in *libcl*. A ring buffer contains a number of blocks of arbitrary size. It maintains a pointer to the buffer location, a size, and pointers to the *Head* of newest and *Tail* of oldest valid data. Separate processes can be *producing* (adding to the buffer) and *consuming* (removing from the buffer).

Figure 24-1 shows a conceptual drawing of a ring buffer.

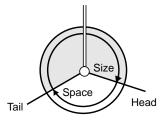


Figure 24-1 Ring Buffer

The circle represents the ring buffer. The shaded part of the circle contains frames or data, depending on the buffer type; the blank part is free space. The size of the data (or the number of frames) available and the size of the space (or the number of frames of space) are shown by the arrows within the circles. Head marks the location where new data or frames, depending on the buffer type, are inserted. Tail marks the location where the oldest data or frames, depending on the buffer type, are removed. The head and tail march around the circle as data or frames, depending on the buffer type, are produced and consumed. The double vertical bar at the top signifies the discontinuity between the end of the buffer and the beginning of the buffer in linear physical memory.

## Creating a Buffer

The buffer management routines allow buffer space to be allocated by the library (internal) or by the application (external). A buffer often already exists in memory where the frames exist (on compression) or need to be placed (on decompression). External buffering allows this to happen without having to copy the data to or from an internal buffer. An external buffer is managed entirely within *libcl* as a ring buffer.

Use clCreateBuf() to create an internal or external buffer. Use clDestroyBuf() to destroy an internal or external buffer. If clDecompress() or clCompress() is called with NULL for the compressed data or frame buffer parameters, then the buffer specified by clCreateBuf() is used. An error is reported if no buffer was created.

#### The function prototypes are:

int clDestroyBuf (CLbufferHdl bufferHdl)

where:

handle is the handle to the compressor or decompressor

*bufferType* specifies the type of the ring buffer, which can be either:

CL FRAME for a frame buffer

CL\_DATA for a data buffer

blocks specifies the number of blocks in the buffer

blockSize specifies the size in bytes of the block. This value is either 1

for data buffering or a multiple of the frame size for frame

buffering

bufferPtr is a pointer to a pointer to the ring buffer. If it points to a

NULL pointer, it specifies an internally allocated buffer, and

the value it points to receives the buffer pointer

bufferHdl is a handle to the buffer

The handle returned in *bufferHdl* is used in subsequent buffering calls, with which you can get the buffer handle, and get the compressor or decompressor handle.

Use **clQueryBufferHdl()** to get the buffer handle from a compressor or decompressor handle. Its function prototype is:

```
CLbufferHdl clQueryBufferHdl(CLhandle handle,
int bufferType, void **bufferPtr2)
```

Use **clQueryHandle()** to get the compressor or decompressor handle from a buffer handle. Its function prototype is:

```
CLhandle clQueryHandle(CLbufferHdl bufferHdl)
```

The code fragment in Example 24-6 demonstrates how to create and use an internal buffer.

# **Example 24-6** Creating and Using an Internal Buffer

```
#include <cl.h>
CLhandle handle;
CLbufferHdl bufferHdl;
void *buffer;
...
clOpenCompressor(CL_MVC1, &handle);

/* Create a buffer of 10 blocks of size 10000 */
buffer = NULL;
bufferHdl = clCreateBuf(handle, CL_DATA, 10, 10000, &buffer);
bufferHdl = clQueryBufferHdl(handle, CL_DATA, &buffer);
handle = clQueryHandle(bufferHdl);
...
clDestroyBuf(bufferHdl);
clCloseCompressor(handle);
```

The code fragment in Example 24-7 demonstrates how to create and use an external buffer.

## **Example 24-7** Creating and Using an External Buffer

```
#include <cl.h>
CLhandle handle;
CLbufferHdl bufferHdl;
void *buffer;
```

```
clOpenCompressor(CL_MVC1, &handle);

/* Create a buffer of 10 blocks of size 10000 */
buffer = malloc(10*10000);
bufferHdl = clCreateBuf(handle, CL_DATA, 10, 10000, &buffer);
bufferHdl = clQueryBufferHdl(handle, CL_DATA, &buffer);
handle = clQueryHandle(bufferHdl);
...
clDestroyBuf(bufferHdl);
clCloseCompressor(handle);
```

## **Managing Buffers**

The buffer management routines are used for both uncompressed (or decompressed) frames and compressed data. When used for compressed data, they return the number of blocks (of selectable byte size) of valid contiguous data (or free space for data). When used for frames, they return the actual number of valid contiguous frames (or free space for frames).

Use **clQueryFree()** to find out how much free space is available and where it is located.

Use **clUpdateHead()** to notify the library that data has been placed in the ring buffer and to update the head pointer.

Use **clQueryValid()** to find out how many blocks of valid data are available and where they are located.

Use **clUpdateTail()** to notify the library that valid data has been consumed from the ring buffer and that data is no longer needed.

Use **clDoneUpdatingHead()** to notify a decompressor that no more data will be arriving, in which case **clDecompress()** returns when the buffer empties.

The function prototypes are:

int clUpdateTail (CLbufferHdl bufferHdl, int amountToRelease)
int clDoneUpdatingHead (CLbufferHdl bufferHdl)

where:

bufferHdl is a handle to a compressor buffer.

space is the number of blocks of free space in the frame buffer to

wait for. If it is zero, then the current number of blocks of

space is returned without waiting.

freeData is a pointer to the returned pointer to the location where

data or frames can be placed.

wrap is the number of blocks that have wrapped to the beginning

of the ring buffer (see Figure 24-2 and the accompanying discussion). If it is greater than zero, then the end of the ring buffer has been reached and the routine return value will

not increase (on subsequent calls) until either

clUpdateHead() for free space or clUpdateTail() for valid

data has been called.

amountToAdd is the number of blocks of free space that were written by

the caller and are ready to be consumed by the library.

amount is the number of blocks of valid data in the data buffer to

wait for. If it is zero, then the number of blocks currently

available is returned without waiting.

validData is a pointer to the returned pointer to the location where

valid data can be retrieved.

amountToRelease

is the number of blocks of valid data that were consumed by

the call and can be reused by the library.

Each compressor or decompressor can have a (compressed) data buffer and a (uncompressed) frame buffer.

The block size for the uncompressed frame buffer must be a multiple of the size of one frame. This value, multiplied by the number of blocks specified, determines how many frames ahead a decompressor can get if you allow it to work ahead.

## Producing and Consuming Data in Buffers

Figure 24-2 on page 512 shows snapshots of the buffer state over time as a sequence of produce and consume processes operate on the buffer. Initially, the buffer is empty and both Head and Tail point to the beginning of the buffer. When Head and Tail are equal, the buffer is either empty or full—in this case, the buffer is empty. The library keeps track of whether the buffer is empty or full internally.

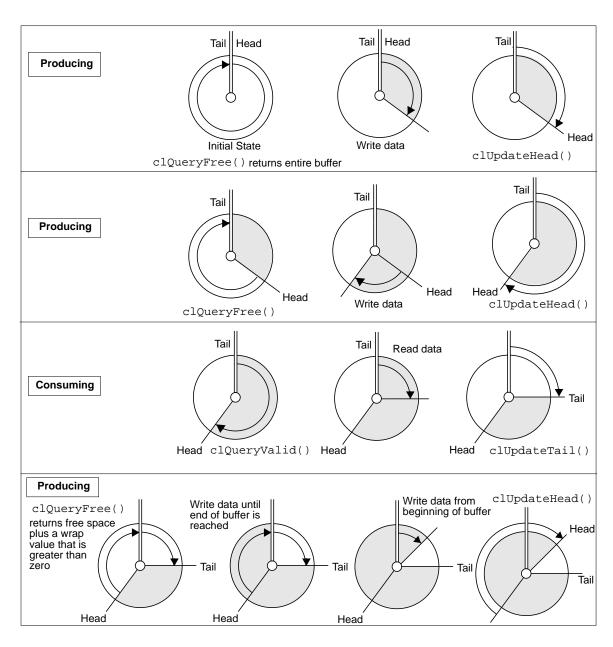
In the first frame of Figure 24-2, a process begins producing—adding data to the buffer. First, a call is made to **clQueryFree()** to determine how much free space is available. An amount equal to the entire buffer size is returned. Data is written to the buffer, then the location of Head is updated to point to the beginning of the next available free space.

In the second frame of Figure 24-2, the next call to **clQueryFree()** returns the free space that exists from Head to Tail. More data is written and the Head is updated once again.

In the third frame of Figure 24-2, a process begins consuming—taking data from the buffer. A call is made to **clQueryValid()** to determine the amount of valid data in existence. The size of the data that was written by the producers so far is returned. Data is read from the beginning of the buffer to the desired location, and Tail is updated to point to the next location containing valid data.

The final frame of Figure 24-2 shows what happens when the free space is not contiguous. When the next producer queries for the available free space, two pieces of free space exist—one on each side of the buffer discontinuity. The first piece of free space, which is from Head to the end of the buffer, is returned as usual. The second piece of free space, which is from the beginning of the buffer to Tail, is returned in the *wrap* argument. You can't write data across the buffer boundary, so it must be written to the buffer in two steps. First write the data until the end of the buffer is reached, then write the data from the beginning of the buffer until all of the data has been used. Head can then be updated to point to the next available free space.

The process for reading data across the frame discontinuity is analogous.



**Figure 24-2** Snapshots of Buffer State During Producing and Consuming Processes

Figure 24-3 shows the architecture of the buffer management. Rectangles represent code modules that can be placed in separate synchronized processes. The buffer management routines are shown within the boxes. Arrows show the flow of data from the modules to and from the buffers

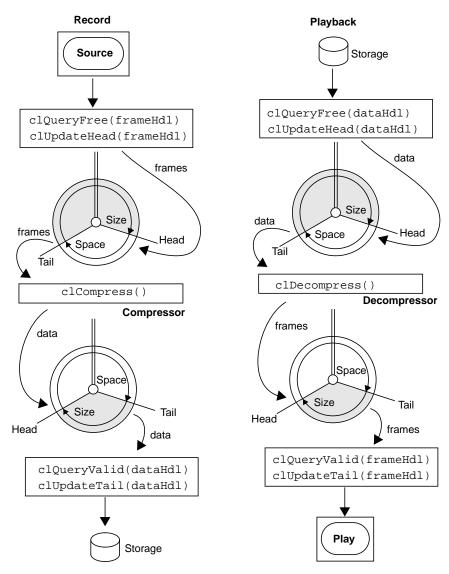


Figure 24-3 Flow of Data in a Buffered Compression and Decompression Scheme

## Creating a Buffered Record and Play Application

This section provides several examples of how to use buffering. Blocking and nonblocking playback and record examples are provided.

#### Creating a Basic Buffered Playback Application

The code fragment in Example 24-8 demonstrates how to use buffers for a playback application. The amount of space is queried, the data is read directly into the data buffer, and the decompressor is notified of the change. The data can then be decompressed and retrieved by querying the number of frames, displaying them directly from the frame buffer, then releasing the consumed frames.

## **Example 24-8** Using Buffers for Playback

```
#include <cl.h>
...
actualLen = clQueryFree(decompressorHdl, len, &buf, &wrap);
read(fd, buf, actualLen);
len = clUpdateHead(dataHdl, actualLen);

clDecompress(decompressorHdl, 1, 0, NULL, NULL);

actualNumberOfFrames = clQueryValid(frameHdl, numberOfFrames,
    &frameBuffer, &wrap);
ConsumeFrames(actualNumberOfFrames, frameBuffer);
numberOfFrames = clUpdateTail(bufferHdl,
actualNumberOfFrames);
```

**clUpdateHead()** indicates to the library that the data has been placed in the data buffer, but does not copy the data.

**clDecompress()** reads compressed data from the data buffer and writes uncompressed frames to the frame buffer. If space for a frame exists in the frame buffer, then it begins decompressing directly to the frame buffer. It consumes data from the data buffer until there is no more data, then it sleeps for a while and periodically continues to check for data until there is enough data. When it finishes decompressing a frame, it updates the frame buffer pointers and returns. **clDecompress()** does not return until decompression is complete or until an error occurs.

If no more data will be added to the buffer, the application can call **clDoneUpdatingHead()** so that the library will not stall.

**clQueryValid()** returns the pointer into the frame ring buffer. **clUpdateTail()** is required to free the internal frame buffer space, which you don't want to happen until after you consume it. The pointer to the next valid frame is kept internally, and only the actual number of frame buffers that have been decompressed are returned.

The *size* (or *numberOfFrames*) returned by the routines are for the contiguous data (or frames, depending on the buffer type). The *wrap* argument of the **clQuery()** routines returns the *actualLen* (or *numberOfFrames*) that have wrapped to the beginning of the buffer.

The frame accesses will not cross the buffer boundary, and the *wrap* argument does not need to be used if both:

- the allocated size of the frame ring buffer is a multiple of the size of a frame times the *numberOfFrames* that will be requested
- the same number of frames will always be requested

If the *len* (or *numberOfFrames*) passed to the **clQuery()** routines is greater than zero, the routine blocks until that much data (or that many frames) is available. If it is less than or equal to zero, then the routine returns immediately with whatever data is available. In either case, the buffer pointers are not adjusted until the **clUpdate()** routines are called.

## **Creating a Nonblocking Buffered Playback Application**

The code fragment in Example 24-9 demonstrates how to implement nonblocking playback.

## **Example 24-9** Using Buffers for Nonblocking Playback

```
actualLen = clQueryFree(decompressorHdl, 0, &buf, &wrap);
if((actualLen > MIN_READ_SIZE) || (wrap > 0)) {
   read(fd, buf, actualLen);
   len = clUpdateHead(decompressorHdl, actualLen);
}
/* Go do something else */
...
```

Each call to clQueryFree() returns the same <code>buf</code> pointer, but increasing values of <code>actualLen</code> until MIN\_READ\_SIZE is reached, whereupon <code>clUpdateHead(dataHdl)</code> updates the pointers and the next call to <code>clQueryFree()</code> returns a different <code>buf</code> pointer and a reset <code>actualLen</code>. If <code>wrap</code> becomes greater than zero, the end of the buffer has been reached and <code>actualLen</code> will not get any larger, so the amount remaining in the buffer must be consumed.

## **Creating a Buffered Record Application**

The code fragment in Example 24-10 demonstrates how to use buffers for recording.

#### **Example 24-10** Using Buffers for Recording

The amount of free space is queried, the frames are read directly into the frame buffer, and the compressor is notified of the change. The frames can then be compressed and the data can be retrieved by querying the amount of the data, consuming directly from the data buffer, then releasing the consumed data.

**clUpdateHead()** indicates that the frames have been placed in the frame buffer, but does not copy the data.

**clCompress()** reads from the frame buffer and writes to the data buffer. If a frame exists in the frame buffer, then it begins compressing directly from the frame buffer. It places compressed data in the data buffer until there is no more room, then it blocks until there is enough room. When it completes compression of a frame, it updates the frame buffer pointers and returns.

**clCompress()** does not return until compression is complete (or an error occurs).

**clQueryValid()** returns the pointer into the data ring buffer. **clUpdateTail()** is required to free the internal data buffer space, which you don't want to happen until after you consume it—in this case, by writing it. The pointer to valid data is kept internally, and **clUpdateTail()** returns only the actual number of bytes released.

The *amount/numberOfFrames* returned by the routines are for contiguous data/ frames. The *wrap* parameter of the **clQuery()** routines returns the *amount/numberOfFrames* that have wrapped to the beginning of the buffer.

If the allocated size of the frame ring buffer is a multiple of the size of a frame times the *numberOfFrames* that will be requested, assuming that the same number of frames is always requested, then the frame accesses will not cross the buffer boundary, and the *wrap* parameter does not need to be used.

If the *amount* passed to the **clQuery()** routines is greater than zero, then the routine blocks until that much data is available. If it is less than or equal to zero, then the routine returns immediately with whatever data is available. In either case, the buffer pointers are not adjusted until the **clUpdate()** routine is called.

#### Creating a Nonblocking Buffered Record Application

The code fragment in Example 24-11 demonstrates how to use buffers for nonblocking recording.

#### **Example 24-11** Using Buffers for Nonblocking Recording

```
actualLen = clQueryValid(dataHdl, 0, &buf, &wrap);
if((actualLen > MIN_READ_SIZE) || (wrap > 0)){
   write(fd, buf, actualLen);
   len = clUpdateTail(dataHdl, actualLen);
}
```

Each call to **clQueryValid()** returns the same *buf* pointer, but increasing values of *actualLen* until MIN\_READ\_SIZE is reached, whereupon **clUpdateTail()** updates the pointers, and the next call to **clQueryValid()** returns a different *buf* pointer and a reset *actualLen*. If *wrap* becomes greater

than zero, then the end of the buffer has been reached, and *actualLen* will not get any larger, so the amount remaining in the buffer must be consumed.

Note that the consuming, compressing or decompressing, and producing have been separated into different sets of calls. The most powerful use of the interface is to separate these functional groupings into shared processes using **sproc()** or to allocate them to separate (shared data) processors. See *sproc(*2) for more information about using **sproc()**.

The buffers are set up by **clCreateBuf()**. In order to use data input buffering, **clDecompress()** receives NULL for *compressedData*. In order to use frame output buffering, **clDecompress()** receives NULL for *frameBuffer*.

**clCompress()** reads from the frame buffer and writes to the data buffer. If a frame exists in the frame buffer, then it begins compressing directly from the frame buffer. It places compressed data in the data buffer until there is no more room, then it sleeps for a while and checks again until there is enough room. When it finishes compressing a frame, it updates the frame buffer pointers and returns. **clCompress()** does not return until compression is complete or until an error occurs.

# **Creating Buffered Multiprocess Record and Play Applications**

In the examples in the previous section, consuming, compressing or decompressing, and producing have been separated into different sets of calls. The most powerful use of the buffering interface is to separate these functional groups into shared processes using **sproc()** or to allocate them to separate (shared data) processors.

The code fragment in Example 24-12 demonstrates how to implement multiprocess playback. The functions in boldface can be implemented as separate processes.

## **Example 24-12** Using Buffers for Multiprocess Playback

```
ProduceDataProcess()
  actualLen = clQueryFree(dataHdl, len, &buf, &wrap);
  read(fd, buf, actualLen);
  len = clUpdateHead(dataHdl, actualLen);

DecompressProcess()
  clDecompress(decompressorHdl, 1, 0, NULL, NULL);

ConsumeFrameProcess()
  actualNumberOfFrames = clQueryValid(frameHdl,
    numberOfFrames, &frameBuffer, &wrap);
  lrectwrite(0, 0, width - 1, height - 1, frameBuffer);
  numberOfFrames =
clUpdateTail(frameHdl,actualNumberOfFrames);
```

The code fragment in Example 24-13 demonstrates how to use buffers for multiprocess recording. The functions in boldface can be implemented as separate processes.

## **Example 24-13** Using Buffers for Multiprocess Recording

```
ProduceFrameProcess()
  actualNumberOfFrames = clQueryFree(frameHdl,
      numberOfFrames, &frameBuffer, &wrap);
  lrectread(0, 0, width - 1, height - 1, frameBuffer);
  numberOfFrames = clUpdateHead(frameHdl,
      actualNumberOfFrames);

CompressProcess()
  clCompress(compressorHdl, 1, NULL, &compressedDataSize,
      NULL);

ConsumeDataProcess()
  actualBufSize = clQueryValid(dataHdl, bufSize,&buf, &wrap);
  write(fd, buf, actualBufSize);
  bufSize = clUpdateTail(dataHdl, actualBufSize);
```

This allows the application nonblocking access to compression and decompression. The application will almost always use <code>ProduceDataProcess()</code> for playback and the <code>ProduceFrameProcess()</code> for record, since the single process will block forever within <code>clDecompress()/clCompress()</code> if insufficient data or frames, depending on the buffer type, are supplied. The other processes can be made parts of the <code>main()</code> process. These processes could also be spread across multiple processors.

# **Programming with the Cosmo Compress JPEG Codec**

Cosmo Compress is an optional hardware JPEG accelerator for workstations equipped with Galileo Video options, including: Galileo Video for Indigo<sup>2</sup> and Indigo R4000 computers, Indigo<sup>2</sup> Video, Indy Video, and Sirius Video. Cosmo Compress is capable of compressing to and decompressing from memory, or directly through a special video connection to Galileo Video options.

Cosmo Compress JPEG is a lossy compression scheme based on psychovisual studies of human perception. Picture information that is generally not noticeable is dropped out, reducing the storage requirement anywhere from 2 to 100 times. Cosmo Compress implements a subset of the JPEG standard especially for video-originated images (baseline JPEG, interleaved YCrCb 8-bit components).

## **Cosmo Compress Basics**

See the *Cosmo Compress Execution Environment Release Notes* for important prerequisite information and installation instructions. Your workstation must be equipped with the following hardware and software components in order to be able to use Cosmo Compress:

- Cosmo Compress option and Cosmo Compress software
- Video option with output capability
- Iris Development Option software

You can program Cosmo Compress from the Compression Library (CL), using either the buffered or sequential interface along with JPEG-specific and Cosmo-specific CL parameters.

Cosmo Compress has four different modes of operation:

- compressing video from an external video device into a memory buffer
- decompressing video from a buffer to an external video device
- compressing an image stored in memory into another area of memory
- decompressing a stored compressed image into a buffer

To add Cosmo Compress support to your application:

1. Include the *dmedia/cl\_cosmo* header in order to get definitions for Cosmo Compress:

#include <dmedia/cl\_cosmo.h>

- 2. Set Cosmo Compress specific compression parameters:
  - Set image formats as described in "Cosmo Compress Image Formats" on page 522
  - Enable CL\_ENABLE\_IMAGEINFO as described in "Getting Compressed Image Information" on page 524.
- 3. Specify CL\_JPEG\_COSMO as the scheme argument for clOpenCompressor() when opening a compressor or clOpenDecompressor() when opening a decompressor. Only one application can have Cosmo Compress open at a time—an error will be returned to the program if another application has Cosmo Compress open.
- 4. Compress or decompress frames.

# **Cosmo Compress Image Formats**

This section describes CL image parameters supported by Cosmo Compress. Cosmo Compress works with video fields, 2 of which compose a video frame.

Cosmo Compress requires that input images have height and width dimensions that are multiples of 8 pixels because the JPEG compression algorithm processes images in blocks of 8x8 pixels. The CL associates two sets of image dimensions with an instance of a video compressor or decompressor: one set for uncompressed images (CL\_IMAGE\_WIDTH and CL\_IMAGE\_HEIGHT), and one set for compressed images (CL\_INTERNAL\_IMAGE\_WIDTH and CL\_INTERNAL\_IMAGE\_HEIGHT).

For memory-to-memory compression, CL\_IMAGE\_WIDTH always equals CL\_INTERNAL\_IMAGE\_WIDTH, and CL\_IMAGE\_HEIGHT always equals CL\_INTERNAL\_IMAGE\_HEIGHT.

Table 24-3 summarizes the image format parameters

 Table 24-3
 Cosmo Compress Image Format Parameters

Image Attribute	Description	Parameter(s)	Values
Pixel format	Cosmo Compress supports 32-bit RGB only for memory- memory transfers and YCrCb 4:2:2 only for video-memory transfers.	CL_ORIGINAL_FORMAT	CL_RGBX (memory-memory) CL_YUV (video-memory)
Interlacing	Cosmo Compress operates on interlaced NTSC or PAL video data for video-to-memory compression and memory-to-video decompression. Even and odd fields are compressed as separate images.	DM_IMAGE_INTERLACING	NTSC or CCIR(525): DM_IMAGE_INTERLACED_EVEN PAL or CCIR(625): DM_IMAGE_INTERLACED_ODD
Image orientation	Cosmo Compress compresses/decompresses images that have top-to-bottom orientation. By default, <b>lrectwrite</b> (3g) draws images with bottom-to-top orientation. Use <b>pixmode</b> (3g) to set graphics orientation to PM_TTOB in order to correctly display top-to-bottom images.	CL_ORIENTATION	CL_TOP_DOWN DM_TOP_TO_BOTTOM (for SGI movies)
Uncompressed image dimensions	Uncompressed image height (in pixels).	CL_IMAGE_HEIGHT	Range: 16–336, in multiples of 8. (NTSC must use either 240 or 248) Default: 248
	Uncompressed image width (in pixels )	CL_IMAGE_WIDTH	640 (NTSC), 720 (CCIR(525) and CCIR(625)), 768 (PAL). Default: 640
Compressed image dimensions	Compressed image height (in pixels)	CL_INTERNAL_IMAGE_HEIGHT	Range: 16–336, in multiples of 8.
	Uncompressed image width (in pixels)	CL_INTERNAL_IMAGE_WIDTH	320 (NTSC), 360 (CCIR(525) and CCIR(625)), 384 (PAL). Default: 320

## **Getting Compressed Image Information**

The CL provides a function exclusively for Cosmo Compress that lets you get information such as the size, timestamp, and a relative image index value for images (fields or frames) as they are compressed or decompressed through Cosmo Compress. When compressing from external video, the timestamp returned represents the time at which the first line of the uncompressed field arrived at the Cosmo Compress board.

To get compressed image information:

- 1. Call **clSetParam()** to set the CL\_ENABLE\_IMAGEINFO parameter to TRUE before compressing or decompressing any frames.
- 2. Call **clGetNextImageInfo()** to get a structure containing information about the compressed image:

handle specifies an open handle which is actively compressing

or decompressing

info is a pointer where a CLimageInfo structure is to be

placed

sizeofimageinfo specifies the size of the CLimageInfo structure in bytes

The CLimageInfo structure is defined in *dmedia/cl.h* and has the following fields:

```
typedef struct {
   unsigned size; /* size of compressed image in bytes */
   unsigned long long ustime; /* time in nanoseconds */
   unsigned imagecount; /* media stream counter */
   unsigned status; /* additional status information */
} CLimageInfo;
```

The *ustime* field returns a meaningful value only when compressing from or decompressing to an external device. The status field is reserved for future use.

**Note:** Currently, in order to get valid JPEG data, an application using the CL\_JPEG\_COSMO compressor must enable **clGetNextImageInfo()** by setting CL\_ENABLE\_IMAGEINFO, and then read a CLimageInfo structure corresponding to each compressed image, before calling **clCompress** to read the compressed image data.

When using the CL\_JPEG\_COSMO decompressor, you don't need to read CLimageInfo structures. When **clGetNextImageInfo()** is enabled, the CL uses a small internal buffer to queue the structures during decompression. When this queue fills, the oldest structures are overwritten by new ones.

**clGetNextImageInfo()** blocks only when it is waiting for the first valid decompressed field to exit the CL\_JPEG\_COSMO decompressor.

## Memory-to-Memory Compression and Decompression

You can use Cosmo Compress to compress images from a memory archive to a buffer. For example, you can use Cosmo Compress to compress images from a movie file to a buffer, and then insert the JPEG-compressed images into a movie file to create a compressed movie. Taking this idea a step further, you can then use Cosmo Compress to scale down the images as it decompresses them, in order to display thumbnail images similar to the ones in Movie Player. See <code>vidmemcomp.c</code> in <code>/usr/people/4Dgifts/examples/dmedia/compression/vidmemcomp</code> for example code that demonstrates these concepts.

## **Memory-to-Memory Compression**

To compress frames into memory using Cosmo Compress:

- 1. Set the CL image parameters to characterize the input image data.
- 2. Open a CL\_JPEG\_COSMO compressor.
- 3. Compress frames into memory. Each frame contains 2 fields.

When compressing images from memory into a buffer, Cosmo Compress supports image widths of 16–768 and image heights of 16–336, in multiples of 8. You cannot scale images when compressing into memory, therefore, CL\_IMAGE\_WIDTH equals CL\_INTERNAL\_IMAGE\_WIDTH, and CL\_IMAGE\_HEIGHT equals CL\_INTERNAL\_IMAGE\_HEIGHT.

The uncompressed data format must be 32-bit RGB (CL\_RGBX), and the uncompressed image size cannot be larger than PAL video.

**Note:** NTSC frames have a width of 243, but Cosmo Compress supports only input image widths that are multiples of 8. For NTSC, you must specify an image width of either 240 (causing the image to be cropped 3 lines from the bottom) or 248 (causing the image to be padded with 5 extra lines). Output image widths do not have to be multiples of 8.

Example 24-14 demonstrates memory-to-memory compression of NTSC video.

**Example 24-14** Cosmo Compress Memory-to-Memory Compression

```
#include <dmedia/cl.h>
    int pbuf[][2] = {
       CL_IMAGE_WIDTH, 0,
       CL_IMAGE_HEIGHT, 0,
       CL_COMPRESSED_BUFFER_SIZE, 0
   };
    /* Compress a series of frames */
   clOpenCompressor(CL_JPEG_COSMO, &handle);
   /* set parameters */
   pbuf[0][1] = 640;
   pbuf[1][1] = 240;
   clSetParams(handle, (int *)pbuf, 4);
    /* allocate the required size buffer */
   clGetParams(handle, (int *)pbuf, 6);
   compressedBuffer = malloc(pbuf[2][1]);
   for(i = 0; i < numberOfFrames; i++)</pre>
        /* Get a frame from somewhere */
        clCompress(handle, 1, frameBuffer,
                  &compressedBufferSize, compressedBuffer);
        /* Write the compressed data to somewhere else. */
   clCloseCompressor(handle);
```

After compressing the images, you can use **mvInsertCompressedImage()** to insert the compressed images into a movie file, as described in "Reading and Inserting Compressed Images" in Chapter 29. Since the JPEG images are stored in fields, you must read two fields for every frame.

### **Memory-to-Memory Decompression**

To decompress JPEG images from memory using Cosmo Compress:

- 1. Set the CL image parameters to characterize the output image data.
- Open a CL\_JPEG\_COSMO decompressor.
- 3. Decompress frames into a buffer. Each frame contains 2 fields.

You can shrink the images as they are decompressed, which is useful for displaying thumbnail images. When decompressing images from memory into a buffer, Cosmo Compress supports image widths of 16–768 and image heights of 16–336. Scaling can be arbitrary, that is, you can scale the image dimensions down by any amount, and the output image dimensions do not have to be multiples of 8. To shrink images as they are decompressed, make the uncompressed image dimensions (CL\_IMAGE\_WIDTH and CL\_IMAGE\_HEIGHT) less than the corresponding compressed image dimensions (CL\_INTERNAL\_IMAGE\_WIDTH and CL\_INTERNAL\_IMAGE\_HEIGHT).

# Compressing and Decompressing Video Through External Connections to Cosmo Compress

You can use Cosmo Compress as a real-time JPEG codec between your application and an external video device. This section explains how to use Cosmo Compress to compress images from an external video connection into memory and decompress JPEG images from memory to a video device.

## **Video-to-Memory Compression**

To capture video from an external video device using Cosmo Compress:

- 1. Connect the video device to the appropriate port. For example, use either analog port 1 or digital port 1. Video port connections are managed from the *videopanel* control panel.
- 2. Open a connection to the video server by calling **vlOpenVideo("")**.
- 3. Create the video transfer paths.
  - Get the source (VL\_SRC) node for the video signal connection by calling **vlGetNode()**.
  - Get the drain(VL\_DRN) node for the Cosmo Compress connection by calling **vlGetNode()**.
  - Create the path from source to drain by calling vlCreatePath().
  - Set up the path to share (VL\_SHARE) data by calling vlSetupPaths().

**Tip:** Cosmo Compress is not a video node; it is a separate device. Therefore the VL does not have a Cosmo VLnode for video paths. The port to which Cosmo Compress is connected (for example, the digital video output) is the video drain node.

- 4. Set the appropriate video synchronization mode. Use slave mode (VL\_EV1\_SYNC\_SLAVE) when capturing from the analog port; use internal mode (VL\_SYNC\_INTERNAL) when capturing from the digital port.
- 5. Set the CL parameters for image dimensions, quality factor, and compressed image information (CL\_ENABLE\_IMAGEINFO).
- 6. Open a CL\_JPEG\_COSMO compressor.
- 7. Call **clGetNextImageInfo()** to get a structure containing information about the compressed image.
- 8. Start the video transfer.
- 9. Use the CL buffered interface to compress frames by calling **clCompress()** with CL\_CONTINUOUS\_NONBLOCK as the *framecount* parameter and CL\_EXTERNAL\_DEVICE as the *frameBuffer* parameter.

**Note:** Instead of using CL\_CONTINUOUS\_NONBLOCK, you can call **clCompress()** from a separate thread within the program. **clCompress()** does not return until the transfer is complete.

See *cosmo\_capture.c* in /*usr/people/4Dgifts/examples/dmedia/dmrecord* for an example of capturing external video through Cosmo Compress.

Video fields entering Cosmo Compress from the direct video connection are captured into an array of field buffers. The field buffers support field widths from 640 to 768 and field heights from 16 to 336. Field dimensions depend on the video timing, as shown in Table 24-4.

Table 24-4 shows video field dimensions for the video formats supported by Cosmo Compress.

**Table 24-4** Cosmo Compress Video Field Dimensions

Video Format	Wldth (pixels)	Height (pixels)
NTSC	640	243
PAL	768	288
CCIR(525)	720	243
CCIR(625)	720	288

Lines in the field buffers following the end of valid video data are filled with indeterminate data (that is, they are not blacked out).

When the compressed image height is less than the height of the incoming video fields, the video fields are clipped from the bottom before they are sent to the compressor. When the compressed image height is greater than the height of the incoming video fields, additional lines of indeterminate data are appended to the valid video data before the data is sent to the compressor.

**Note:** NTSC fields have a height (243 pixels) which is not a multiple of 8. For NTSC capture, you can choose to have your application either throw away 3 lines from the bottom of each field (240 pixel height) or append 5 extra lines to the bottom of each field (248 pixel height) prior to compression.

You can scale the captured image to half-size before compressing it. This allows for an additional increase in data compression by factor of 4.

Specify vertical decimation by setting the compressed image height (CL\_INTERNAL\_IMAGE\_HEIGHT) to half the size of the uncompressed image height (CL\_IMAGE\_HEIGHT). Compressed image heights can range from 16 to 168 and uncompressed image heights can range from 32 to 336.

Specify horizontal decimation by setting the compressed image width (CL\_INTERNAL\_IMAGE\_WIDTH) to half the size of the uncompressed image width (CL\_IMAGE\_WIDTH) as indicated in Table 24-5.

**Table 24-5** Cosmo Compress Field Widths for Compression With Decimation

Video Format	CL_IMAGE_WIDTH (pixels)	CL_INTERNAL_IMAGE_WIDTH (pixels)
NTSC	640	320
PAL	768	384
CCIR(525)	720	360
CCIR(625)	720	360

During video compression from an external device, *CLimageInfo.imagecount* is initialized to 1 when the first field is received by the compressor after calling **clCompress()**. The count advances when a new field arrives. If the compression data buffer fills up, then a field will be dropped, but the *imagecount* continues to increase. An application can thus detect a dropped field by noticing a jump in the *imagecount* field of more than one. The *ustime* indicates the time the uncompressed field entered the compressor.

#### Memory-to-Video Decompression

The connections for decompressing from memory to an external video are set up similar to those for capturing video, except that a decompressor is opened. See <code>clInit.c</code> in <code>/usr/people/4Dgifts/examples/dmedia/dmplay</code> for example code that initializes the CL for JPEG decompression (optionally through Cosmo Compress) from memory to external video.

Video playback of the decompressed frames requires media synchronization. See *dmplay.c* and *streamDecompress.c* in */usr/people/4Dgifts/examples/dmedia/dmplay* for more information.

Uncompressed fields leaving the JPEG decompressor may optionally be scaled up by a factor of 2 in the horizontal and/or vertical dimensions. NTSC, PAL or CCIR(525)/CCIR(625) fields are then scanned out of the array of field buffers. Horizontal scaling is performed by pixel replication, vertical scaling is performed by line doubling. If the uncompressed fields leaving the decompressor have fewer lines than the field height required by the NTSC, PAL or CCIR(525)/CCIR(625) connection (after optional pistoling), additional lines of indeterminate (not blacked out) data will be scanned out of the field buffers to pad out bottoms of the uncompressed images. If the uncompressed fields leaving the decompressor have more lines than the NTSC/PAL/CCIR(525)/CCIR(625) field height (after optional pistoling), lines will be clipped from the bottom of the uncompressed images.

Specify horizontal scaling by setting the uncompressed image width (CL\_IMAGE\_WIDTH) that is twice the compressed image width (CL\_INTERNAL\_IMAGE\_WIDTH) as indicated in Table 24-6.

 Table 24-6
 Cosmo Compress Field Widths for Decompression

Video Format	CL_IMAGE_WIDTH (pixels)	CL_INTERNAL_IMAGE_WIDTH (pixels)
NTSC	640	320
PAL	768	384
CCIR(525)	720	360
CCIR(625)	720	360

Specify vertical scaling by setting the uncompressed image height (CL\_IMAGE\_HEIGHT) to twice the size of the compressed image height (CL\_INTERNAL\_IMAGE\_HEIGHT). Compressed image heights can range from 16 to 168 and uncompressed image heights can range from 32 to 336.

During video decompression to an external device, *CLimageInfo.imagecount* reflects the count of fields sent by the application to the decompressor. The *ustime* indicates the time that field left the decompressor. In certain situations, fields are repeated on output, in which case the *imagecount* will

remain the same, but the *ustime* will increase. Cosmo Compress decompression has a 1 frame delay through Galileo/IndyVideo before the field actually leaves the machine.

When transferring to or from external video, the video can be played continuously (default) or single-stepped a field or frame at a time. In either mode, the frame output is composed of either a single field replicated twice or two different fields. Specify the frame control by setting CL\_COSMO\_VIDEO\_TRANSFER\_MODE.

For continuous transfer, set CL\_COSMO\_VIDEO\_TRANSFER\_MODE to CL\_COSMO\_VIDEO\_TRANSFER\_AUTO\_1\_FIELD for the first field in a frame, and CL\_COSMO\_VIDEO\_TRANSFER\_AUTO\_2\_FIELD for the second field in a frame.

For manual control, set CL\_COSMO\_VIDEO\_TRANSFER\_MODE to CL\_COSMO\_VIDEO\_TRANSFER\_MANUAL\_1\_FIELD, and CL\_COSMO\_VIDEO\_TRANSFER\_MANUAL\_2\_FIELD.

In manual video transfer mode, the output frame can be set to either advance or repeat the current frame or field, as specified by CL\_COSMO\_VIDEO\_FRAME\_CONTROL.

You can control compression or decompression with CL\_COSMO\_CODEC\_CONTROL. Setting CL\_COSMO\_CODEC\_CONTROL to CL\_COSMO\_STOP halts compression or decompression. If **clCompress()** or **clDecompress()** was called with CL\_CONTINUOUS\_BLOCK, the function returns. If **clCompress()** or **clDecompress()** was called with CL\_CONTINUOUS\_NONBLOCK, the associated thread terminates.

## Controlling JPEG Compressed Image Quality

JPEG is a tunable algorithm—you can trade quality for compression ratio and vice-versa. You can specify a hint (CL\_COMPRESSION\_RATIO) for an approximate compression ratio or you can set more explicit quality factors, as described next.

The source image is compressed in three basic steps.

- 1. Data is transformed from spatial to frequency form in eight-by- eight blocks using a discrete cosine transform (DCT).
- 2. The frequency coefficients are filtered down by a linear quantization.
- 3. The coefficients are Huffman-encoded into a bit stream.

The process is reversed for decompression.

The quantization step controls the trade-off between image quality and size. A table called the *JPEG quantization table* is used to scale each of the 64 DCT coefficients. The luminance (Y) and the chrominance (Cr and Cb) components each use a separate table.

The CL provides two methods for controlling image quality from these quantization tables. You can specify an overall JPEG quality factor (CL\_JPEG\_QUALITY\_FACTOR) for scaling the default JPEG quantization tables or you can manually set the quantization tables CL\_JPEG\_QUANTIZATION\_TABLES.

The JPEG algorithm does not allow you to specify exact compression ratios (or bit rate targets), so the CL\_EXACT\_COMPRESSION\_RATIO parameter is not supported by the CL JPEG codecs.

#### Specifying a JPEG Quality Factor

You can use the CL\_JPEG\_QUALITY\_FACTOR parameter to specify a JPEG quantization table scale factor that represents a rough percentage of the image detail preservation. This is one method to control the image loss and therefore the compression ratio for the Cosmo Compress JPEG algorithm.

Each time the quality factor is set, the reference quantization tables are scaled and downloaded into the codec. The formula used to obtain the scale factor is:

```
scalefactor = 50/quality (quality < 50)
scalefactor = 2 - 2*quality/100; (otherwise)</pre>
```

The default quality is CL\_JPEG\_QUALITY\_DEFAULT, which represents a good-quality compressed image. A quality factor of 1 results in coarse quantization, a high compression ratio, and very poor image quality.

A quality factor of 100 results in the finest possible quantization, a low compression ratio (perhaps even image expansion), and near-perfect image quality. The most useful quality factor is typically in the range of 25–95.

To bypass scaling, specify CL\_JPEG\_QUALITY\_NO\_SCALE.

#### **Defining and Using Custom JPEG Quantization Tables**

You can customize the JPEG quantization tables by using the CL\_JPEG\_QUANTIZATION\_TABLES parameter. To set the tables, specify an unsigned short \*qtables[4] argument. For each j, qtables[j] must either be NULL or point to a unsigned short[64] area of memory that represents a JPEG-baseline quantization table in natural scanning order. These custom tables are then stored as reference tables, and then scaled versions of them based on the current CL\_JPEG\_QUALITY\_FACTOR are downloaded into the codec, becoming the tables associated with the ID j.

When getting the value of CL\_JPEG\_QUANTIZATION\_TABLES, the CL allocates the required memory and returns the currently used tables, as indicated by CL\_JPEG\_COMPONENT\_TABLES, scaled by the value of CL\_JPEG\_QUALITY\_FACTOR. Your application is responsible for freeing the memory allocated to return these tables.

You can specify the quantization tables on a per-component basis, by using the CL\_JPEG\_COMPONENT\_TABLES parameter. It specifies the IDs of the AC Huffman table, DC Huffman table, and quantization table to be used for each component. Currently, you cannot change this parameter for Cosmo Compress—it is set up for YUV422 processing. This setting uses AC Huffman table 0, DC Huffman table 0, and quantization table 0 for component 0; AC huffman table 1, DC huffman table 1, and quantization table 1 for components 1 and 2.

# Chapter 25

# **Using Compression Library Algorithms and Parameters**

This chapter describes how to use these components of the CL:

- algorithms for JPEG, MPEG, RLE, RTR, MVC1, and CCITT/TSB G.711 μ-law and A-law compression schemes
- parameters for configuring data and algorithms

# Using Compression Library Algorithms and Parameters

This chapter describes how to use the Compression Library algorithms and parameters.

# In this chapter:

- "Using the Compression Library Algorithms" on page 537 describes the algorithms available in the CL and explains how to use them.
- "Using the Compression Library Parameters" on page 543 describes the CL global parameters and explains how to use them.

# **Using the Compression Library Algorithms**

This section describes how to use the algorithms that are supplied with *libcl*. See Chapter 26, "Customizing the Compression Library," for information on adding and using your own algorithms.

To use one of the algorithms supplied with *libcl*, you need to select an appropriate algorithm for your application and specify it in the compress or decompress routines.

# **Choosing a Compression Library Algorithm**

Perhaps the most important aspect of developing an application that uses *libcl* is selecting the appropriate algorithm to use for the application. The algorithm affects the data size and quality and the rate of compression and decompression, so it is important to consider how an algorithm will affect the end result and to consider whether a particular algorithm will achieve the desired effect. A certain amount of experimentation may be necessary.

If you are interested in a particular quality level, you need to set the compression ratio to achieve that quality; if you are primarily interested in a particular data size or data rate, you need to set the compression ratio to achieve the desired data size or rate.

Here are some suggestions for typical application categories:

**Note:** The performance is quoted for Indigo workstations with 33Mhz MIPS<sup>®</sup> R3000<sup>®</sup> processors only.

Algorithms for Multimedia Information Delivery Applications

Video

The key factors to consider when choosing a video compression algorithm for multimedia applications are playback speed, data size or rate, and quality.

MPEG gives the best video quality for a given data size or rate, but playback speed is limited by the CPU. MVC1 is usually the best choice if MPEG is not fast enough. If an expensive frame-by-frame VCR is not available, recording in real time to disk is important, which can be done with RTR1.

Audio

 $\mu$ -law and A-law audio compression are appropriate for some movies. If higher quality is desired, a license for Aware Inc.'s audio compression can be obtained (see *Aware* (5)).

Algorithms for Telecommunications Applications

Video

The key factors to consider when choosing a video compression algorithm for video/voice mail, video teleconferencing, and other telecommunications applications are the combined compression-decompression speed, data size/rate, and to a lesser extent, quality.

MVC1 gives the best result for video of about 10 frames per second for a 160 by 120 frame size at the cost of a very high data rate. More performance can be achieved by using grayscale.

Audio Either

Either  $\mu$ -law or A-law audio compression at 8KHz can be used with satisfactory results, or the audio can be left uncompressed if the degradation in sound quality is such that it renders the voice data unusable.

Algorithms for Previewing Animations

Video The key factors when choosing a video compression

algorithm for previewing 2D and 3D animations are playback speed, quality, and, to a lesser extent, data size/rate. MVC1 gives the appropriate speed and

quality.

Audio Audio compression is usually not an issue for these

applications.

Algorithms for Editing Movies

Video The key factors to consider when choosing a video

compression algorithm for movie editing applications are decompression speed, image quality, data size/rate,

and compression speed.

For motion video applications, MVC1 is the best choice, especially when the playback is provided by

the MoviePlayer tool. MVC1 provides rapid

decompression. Playback speed can be traded off with image quality. When recording from video hardware to disk, recording in real time to disk is important if a frame-by-frame VCR is not available—leading to the

use of RTR1

Audio The current audio compression algorithms are not

particularly suited to editing. Uncompressed audio is

recommended.

Table 25-1 summarizes the compression and performance relationships of the image and motion video algorithms. Compression, decompression, and codec performance measurements are in frames per second (FPS), as measured for 320 by 240 frames on Indigo workstations with 33Mhz MIPS R3000 processors only.

 Table 25-1
 Capabilities of Image and Video Algorithms

Algorithm	Typical compression ratio from 24-bit RGB	Average bits per pixel	Bits per second at 15 frames per second	Bytes per frame compression	Compress	Decompress <sup>a</sup>	Codec
Uncompressed	1:1	24	27.65Mb	230.4KB			
RLE 8 bit	4.8:1	5	5.76Mb	48KB	6 FPS	11.5 FPS	3.9 FPS
MVC1	5.33:1	4.5	5.2Mb	43.2KB	3 FPS	25 FPS	2.8 FPS
MVC1 Grayscale	8:1	3	3.456Mb	28.8KB	7 FPS	28 FPS	5.6 FPS
RTR1	6:1	4	4.608Mb	38.4KB	$NYM^b$	2.5 FPS	2.0 FPS
RTR1 Grayscale	9:1	2.67	3.072Mb	25.6KB	NYM	8 FPS	NYM
JPEG	16:1	1.5	1.728Mb	14.4KB	1.1 FPS	1.8 FPS	0.7 FPS
MPEG	48:1	0.5	0.576Mb	4.8KB	<< 1 FPS	4.75 FPS	<<1 FPS

a. Decompressed frame per second is the measured performance, including reading the data from disk, decompressing it, and writing it to the

# **Querying Compression Library Algorithms**

This section explains how you can get a list of available algorithms for an audio or video compressor or decompressor, along with the name and type of algorithm, or find the identifier for an algorithm given its name. Other features of the algorithms can also be queried by the application at run time. Querying algorithms, rather than having hard-coded setups, makes it possible to have an algorithm-independent interface, which lets you take

b. NYM—Not Yet Measured.

advantage of future algorithms as they are implemented without redesigning your code.

# **Getting a List of Algorithms**

Use **clQueryAlgorithms()** to get a list of algorithms for the compressor or decompressor identified by *handle*. **clQueryAlgorithms()** returns the size of the buffer needed to contain the list of algorithms and their types.

If the size of the *algorithmTypeBuffer* is smaller than the returned value, a partial list of the algorithms and their types is returned, and you must enlarge the *algorithmTypeBuffer* in order to receive a complete list.

# The function prototype for **clQueryAlgorithms()** is:

#### where:

algorithmMediaType

is the media type of the algorithm, which can be either of the following values:

CL\_AUDIO specifies an audio algorithm
CL\_VIDEO specifies a video algorithm

# algorithmTypeBuffer

is a pointer to an array of *ints* into which **clQueryAlgorithms()** can write algorithm name/type pairs for each parameter associated with *handle*. The even (0,2,4,...) entries receive the algorithm name. The odd entries (1,3,5,...) receive the types. The returned types take on one of three values:

CL\_COMPRESSOR for compression
CL\_DECOMPRESSOR for decompression

CL\_CODEC for both compression and

decompression

bufferLength is the length of the buffer, in ints, pointed to by

paramValueBuffer. If bufferLength is zero, then

paramValueBuffer is ignored and only the return value is

valid.

# **Getting an Algorithm Scheme or Name**

Use **clQuerySchemeFromHandle()** or **clQuerySchemeFromName()** to return the algorithm scheme identifier used by the other compression functions. Use **clGetAlgorithmName()** to return the algorithm name. Their function prototypes are:

```
int clQuerySchemeFromHandle(CLhandle handle)
int clQuerySchemeFromName(int algorithmMediaType, char *name)
char *clGetAlgorithmName(int scheme)
```

# where:

handle is a handle to a compressor or a decompressor

algorithmMediaType

is the media type of the algorithm, which can be either of the

following values:

CL\_AUDIO specifies an audio algorithm
CL\_VIDEO specifies a video algorithm

name is the algorithm name scheme is the algorithm scheme

Example 25-1 demonstrates how to query the CL for a list of algorithms—in this case, video algorithms. The necessary buffer size is returned in the first call to **clQueryAlgorithms()**, then **malloc()** is used to allocate enough buffer space to store the returned list of video algorithms.

**Example 25-1** Getting a List of Compression Library Algorithms

```
#include <dmedia/cl.h>
#include <malloc.h>

int *buffer, bufferLength;
char *name;
/*
```

```
* Get a buffer containing all the video algorithms and types
*/
bufferLength = clQueryAlgorithms(CL_VIDEO, NULL, 0);
buffer = (int *)malloc(bufferLength * sizeof(int));
clQueryAlgorithms(CL_VIDEO, buffer, bufferLength);
scheme = clQuerySchemeFromName(handle);
name = clGetAlgorithmName(scheme);
```

# **Getting License Information**

Use **clQueryLicense()** to obtain license information about an algorithm. The returned message is text intended for inclusion in a message box that is displayed for a user, explaining how to license an algorithm. Failure returns the license error code.

# The function prototype is:

#### where:

scheme

is the algorithm scheme.

functionality

is the type of algorithm, which can be one of:

- CL\_COMPRESSOR for compression
- CL\_DECOMPRESSOR for decompression
- CL\_CODEC for both compression and decompression

message

is a pointer to a returned pointer to a character string containing a message.

# **Using the Compression Library Parameters**

The CL has a group of routines for working with a set of state variables called "parameters" that are unique for each instantiation. These routines are similar to a set of routines in the audio library. You can get and set parameters, either individually or as a group; however, all of the parameters have reasonable defaults that are algorithm-dependent and need not be set.

The Compression Library works with data that is contained in frames. A frame is defined as a sample in time so that:

1 audio sample: mono 8 bit = 1 byte

mono 16 bit = 2 bytes stereo 8 bit = 2 bytes stereo 16 bit = 4 bytes

1 video frame: width \* height \* components \* bitsPerComponent/8 = n bytes

# **Compression Library Parameter Definitions**

Parameters provide state information about frame characteristics, data formats, and algorithms for each compressor/decompressor.

These parameters provide information about image frame dimensions:

# CL\_IMAGE\_WIDTH

The spatial width of a sample (not relevant for audio); the video default is 320, and the audio default is 1.

# CL\_IMAGE\_HEIGHT

The spatial height of a sample (not relevant for audio); the video default is 240, and the audio default is 1.

These parameters describe data formats:

# CL\_ORIGINAL\_FORMAT

On compression, this is the format of the original audio or video. On decompression, this is the format that you want after decompression. The value is a symbolic constant from one of the following lists, depending on its data type:

Video values are: CL\_RGB, CL\_RGBX (default), CL\_RGBA, CL\_RGB332, CL\_GRAYSCALE, CL\_YUV, CL\_YUV422, or CL\_YUV422DC.

Audio values are: CL\_MONO or CL\_STEREO\_INTERLEAVED (default).

# CL\_INTERNAL\_FORMAT

Some video algorithms have several "natural" formats that can be compressed without color space conversion. This parameter allows the selection of one of these formats. The video default is algorithm-specific. Not relevant for audio.

#### CL\_COMPONENTS

A read-only value, as determined by CL\_ORIGINAL\_FORMAT, that indicates the number of components in the data. For example, audio is 1 for mono, and 2 for stereo, video is generally 1 for grayscale, and 3 or 4 for color. The audio default is 2; video default is 4.

# CL\_BITS\_PER\_COMPONENT

The number of bits per component. For example, audio data is either 8-bit or 16-bit, video is generally 8-bit. The audio default is 16; video default is 8.

#### **CL ORIENTATION**

Specifies the orientation of compressed data, which can be either:

CL\_TOP\_DOWN for pixels arranged top-to-bottom (default).

CL\_BOTTOM\_UP for pixels arranged bottom-to-top.

The orientation of compressed data is always top down. When specifying compression or decompression, the original format (or final format) of the data may be bottom up. Specify this inversion by setting the CL\_ORIENTATION parameter to CL\_BOTTOM\_UP instead of the default.

These parameters describe buffer sizes and characteristics:

# CL FRAME BUFFER SIZE

The maximum size, in bytes, of the frame buffer. If **clDecompress()** is called with *numberOfFrames* larger than 1, this value should be the frame size × *numberOfFrames*. Because this is almost always true for audio, CL\_FRAME\_BUFFER\_SIZE should be set when doing audio decompression.

# CL\_COMPRESSED\_BUFFER\_SIZE

The maximum size of the compressed data buffer. The default is calculated as the maximum possible size, taking into account all the factors such as algorithm, encoding method, data type, and so on. If you want to try to use a smaller buffer, you can set this value explicitly. If **clCompress()** is called with *numberOfFrames* larger than 1, this value should be the maximum compressed size of one frame × *numberOfFrames*. Because this is almost always true for audio, CL\_COMPRESSED\_BUFFER\_SIZE should be set when doing audio compression.

# CL\_NUMBER\_OF\_FRAMES

The number of frames in the sequence.

# CL\_BLOCK\_SIZE

The natural block size of the algorithm in samples. It is most efficient to specify *numberOfFrames* to be a multiple of the block size when calling **clCompress()** or **clDecompress()**.

# CL\_PREROLL

The number of blocks of frames that must be supplied to **clDecompress()** before decompressed frames will be returned.

#### CL\_FRAME\_RATE

The requested number of frames per second.

# CL\_FRAME\_TYPE

The decompressor fills in the frame type when it decompresses a frame. Frame type is one of:

CL\_KEYFRAME *frame* is a keyframe

CL\_INTRA equivalent to CL\_KEYFRAME

CL\_PREDICTED frame contains information about

its succeeding frames

CL\_BIDIRECTIONAL frame contains information about

frames that precede and succeed

it

# CL\_ACTUAL\_FRAME\_INDEX

The frame number of the frame most recently decompressed by **clDecompress()**.

These parameters control the compression ratio and/or quality:

# CL\_ALGORITHM\_ID

A parameter that can be queried to find out the scheme identifier of the algorithm of an open compressor or decompressor.

# CL\_COMPRESSION\_RATIO

The target compression ratio. Some algorithms (MVC1, JPEG, and MPEG) are tunable, that is, they allow quality to be traded for compression ratio.

#### CL EXACT COMPRESSION RATIO

A flag determines whether the compression ratio is a target or must be exact. Some algorithm implementations, such as for JPEG, can be only approximated and can never be exact. For algorithms that do support it, it is generally kept within a small range that over time is guaranteed to average out to the specified compression ratio.

CL\_SPEED

The relative speed of playback. A value of 1.0 for this single-precision floating point value means no change. When this value is greater than 1.0, the algorithm tries to use less CPU time by dropping frames or by reducing the quality.

JPEG has the following additional parameters:

# CL\_JPEG\_COMPONENT\_TABLES

Specifies the IDs of the AC Huffman table, DC Huffman table, and quantization table to be used for each component. Currently, this parameter cannot be changed directly, rather, it is set up automatically for processing the selected CL\_INTERNAL\_FORMAT.

YUV formats use AC huffman table 0, DC huffman table 0, and quantization table 0 for component 0; AC huffman table 1, DC huffman table 1, and quantization table 1 for components 1 and 2. RGB formats use tables AC table 0, DC table 0, and quantization table 0 for all components.

# CL\_JPEG\_QUANTIZATION\_TABLES

Sets or gets the quantization tables to be used. For setting, an unsigned short \*qtables[4] argument is specified as the argument to this parameter. For each j, qtables[j] must either

be NULL or point to a unsigned short[64] area of memory which represents a JPEG base-line quantization table in natural scan order. The user-specified tables are stored as reference tables, and scaled versions of them based on the current CL\_JPEG\_QUALITY\_FACTOR are downloaded into the codec and become the table associated with the ID *j*. For getting, the library allocates the memory as described above, and the tables returned to the user are those which are specified by CL\_JPEG\_COMPONENT\_TABLES as being in use and are scaled as indicated by CL\_JPEG\_QUALITY\_FACTOR. The user is responsible for freeing the memory.

# CL\_JPEG\_QUALITY\_FACTOR

A JPEG quantization table scale factor that represents a rough percentage of the image detail preservation. This is one method to control the image loss and therefore the compression ratio for the JPEG algorithm.

Each time the quality factor is set, the reference quantization tables are scaled and downloaded into the codec. The formula used to obtain the scale factor is:

Using a value of 1 results in coarse quantization, a high compression ratio, and very poor image quality. Using a value of 100 results in the finest possible quantization, a low compression ratio (perhaps even image expansion), and near-perfect image quality. The most useful quality factor is typically in the range of 25–95. The default quality is CL\_JPEG\_QUALITY\_DEFAULT, which represents a good-quality compressed image. Use the value CL\_JPEG\_QUALITY\_NO\_SCALE to bypass scaling.

When CL\_QUALITY\_FACTOR is set, the approximate value of CL\_COMPRESSION\_RATIO is calculated; when CL\_COMPRESSION\_RATIO is set, the approximate value of CL\_QUALITY\_FACTOR is calculated. When decompressing JPEG, clDecompress() fills in this value. The actual compression ratio is determined by the quality factor and the image content and therefore may not be exactly what you expect.

MPEG\_VIDEO has the following additional parameters:

# CL\_END\_OF\_SEQUENCE

An end of sequence flag. When the decompressor arrives at the end of the sequence, it sets this flag. The default is FALSE (0).

RTR has the following additional parameters:

# CL\_QUALITY\_LEVEL

Selects a quantization table. 6 is the highest compression ratio and therefore the worst quality; 0 (default) is the lowest compression ratio and therefore the highest quality.

Table 25-2 lists the parameters and their types, ranges, and defaults.

 Table 25-2
 Compression Library Parameters

Parameter	Range	Default
CL_ALGORITHM_ID	Current ID	Current ID
CL_ALGORITHM_VERSION	Current version	Current version
CL_BITS_PER_COMPONENT	0–32	Audio: 16 Video: 8
CL_BLOCK_SIZE	0–2 Billion	1, depends on algorithm
CL_COMPONENTS (read only)	0–8	Depends on original format
CL_COMPRESSED_BUFFER_SIZE	0–2 Billion	Maximum amount of compressed data needed for one frame

Table 25-2 (continued)	Compression Library Parameters		
Parameter	Range	Default	
CL_COMPRESSION_RATIO	Depends on original format and algorithm	Uncompressed 1.0:1  µ-law 2.0:1  A-law 2.0:1  Aware MultiRate I 2–4:1  Aware MPEG 2–4:1  MVC1 5.3:1  JPEG 15.0:1  MPEG Video 48.0:1  RLE 1.0:1  RLE24 1.5:1  RTR 5.0:1	
CL_ENABLE_IMAGEINFO (Cosmo Compress JPEG only)			
CL_END_OF_SEQUENCE (MPEG_VIDEO only)	0 (FALSE)–1 (TRUE)	0 (FALSE)	
CL_EXACT_COMPRESSION_RATIO	0 (FALSE)–1 (TRUE)	Audio: 0 (FALSE) Video: 0 (FALSE)	
CL_FRAME_BUFFER_SIZE	0–2 Billion	Size of one frame	
CL_FRAME_RATE	0–1 Million	Audio: 44100.0 Video: 30.0	
CL_FRAME_TYPE	0–2	Supplied by decompressor	
CL_IMAGE_HEIGHT	0–1Million	Audio: 1 Video: 320	
CL_IMAGE_WIDTH	0–1Million	Audio: 1 Video: 240	
CL_INTERNAL_FORMAT	0–max <sup>a</sup>	Audio: CL_STEREO_INTERLEAVED Video: depends on algorithm	
CL_INTERNAL_IMAGE_HEIGHT			

Table 25-2 (continued)	Compression Library Parameters		
Parameter	Range	Default	
CL_INTERNAL_IMAGE_WIDTH			
CL_JPEG NUM_PARAMS (JPEG only)	0–2 Billion	0 (unknown)	
CL_JPEG_QUALITY_FACTOR (JPEG only)	0–100	75	
CL_JPEG_QUANTIZATION_TABLES (JPEG only)			
CL_LAST_FRAME_INDEX	0–2 Billion	0	
CL_NUMBER_OF_FRAMES			
CL_NUMBER_OF_PARAMS			
CL_ORIENTATION	CL_TOP_DOWN to CL_BOTTOM_UP	CL_TOP_DOWN	
CL_ORIGINAL_FORMAT	0–max <sup>b</sup>	Audio: CL_STEREO_INTERLEAVED Video: CL_RGBX	
CL_PREROLL	0–2Billion	0, depends on algorithm	
CL_RTR_QUALITY_LEVEL (RTR only)	1–6	1	
CL_SPEED	0 (stopped) to 2 Billion, depends on algorithm	1.0 (real-time playback)	

a.  $max = CL_MAX_NUMBER_OF_ORIGINAL_FORMATS$ .

# **Setting and Querying Compression Library Parameters**

After a compressor or decompressor is opened, thus specifying the compression scheme to use, various parameters can be modified using clSetParams(). All of these parameters have reasonable defaults that are algorithm-dependent and need not be set. Some parameters, such as CL\_IMAGE\_WIDTH and CL\_IMAGE\_HEIGHT for video compression, should be set, but setting them is not required.

b. max = Enumerated type value that depends on the format.

# **Getting a List of Parameters and Parameter Types**

Use **clQueryParams()** to get a list of valid parameters and their types for a specified a compressor or decompressor. The compressor being queried is identified by its handle. Its function prototype is:

int clQueryParams(CLhandle handle,int \*paramValuebuffer, int
maxLength)

where:

handle

is the handle to a compressor or decompressor.

param Valuebuffer

is a pointer to an array of *ints* into which **clQueryParams()** can write parameter identifier/parameter type pairs for each parameter associated with the compressor or decompressor. The even (0,2,4,...) entries receive a string that is the parameter identifier. The odd entries (1,3,5,...) receive the parameter type. Parameter type is one of four values:

- CL\_RANGE\_VALUE, meaning that a parameter can assume a range of values in which the relative magnitude of the value is meaningful—that is, increasing values indicate increasing quantities of whatever this parameter controls, and vice-versa.
- CL\_ENUM\_VALUE, meaning that a parameter assumes values from an enumerated type. They have a limited range, but there is no inherent relationship between the range values.
- CL\_FLOATING\_RANGE\_VALUE, meaning that a
   parameter can assume a range of floating point values,
   in which the relative magnitude of the value is
   meaningful—that is, increasing values indicate
   increasing quantities of whatever this parameter
   controls, and vice-versa.
- CL\_FLOATING\_ENUM\_VALUE, meaning that a
  parameter assumes values from an enumerated type.
  They have a limited floating point range, but there is
  no inherent relationship between the range values.

maxLength

is the length of the buffer, in *ints*, pointed to by *paramValuebuffer*. If *maxLength* is zero, then *paramValuebuffer* is ignored and only the return value is valid.

**clQueryParams()** returns the size of the buffer, in *ints*, needed to hold all the parameter identifier/parameter type pairs for the compressor or decompressor identified by *handle*. The parameters are returned in the even locations of *paramValuebuffer*, and their types are returned in the odd locations.

If the size of the *paramValuebuffer* is smaller than the returned value, a partial list of the parameter identifier/parameter type pairs is returned, making it necessary to enlarge the *paramValuebuffer* in order to receive a complete list. To avoid this situation, you can obtain the correct size of the buffer by calling **clQueryParams()** with a NULL buffer pointer and a *maxLength* of 0 to return the actual buffer length without writing any data.

**clQueryParams()** also reports whether the parameter is one of a set of enumerated types, any integer number within a specific range, or any floating point number within a specific range. In each case, the values are numbers within the range returned by **clGetMinMax()** and have the defaults returned by **clGetDefault()**.

Example 25-2 demonstrates how to get a list of parameters for a specified compressor/decompressor.

**Example 25-2** Getting a List of Parameters for a Compressor/Decompressor

```
#include <dmedia/cl.h>
#include <malloc.h>

/*

* Get a buffer containing all the parameters for a specified
* compressor or decompressor.

*/

int *buf, bufferLength;
bufferLength = clQueryParams(handle, 0, 0);
buf = (int *)malloc(bufferLength * sizeof(int));
clQueryParams(handle, buf, bufferLength);
```

# Getting the Parameter ID that Corresponds to a Parameter Name

If you know the name of a parameter, but not its identifier, you can use **clGetParamID()** to get the identifier of a parameter from its name.

Its function prototype is:

```
int clGetParamID(CLhandle handle, char *name)
```

# **Getting and Setting Parameter Values**

You can get or set parameter values as a group or individually.

Use **clGetParams()** to return the current values for the parameters referenced in the *paramValuebuffer* array. The values are written into the odd locations of *paramValuebuffer* immediately after the corresponding parameters.

Use **clSetParams()** to set the current state of the parameters referenced in the *paramValuebuffer* array.

To change a state parameter:

- 1. Call **clQueryParams()** to find out which parameters are available.
- 2. Call **clGetParams()** to find out the current state.
- 3. Fill in the even entries of the *paramValuebuffer* array corresponding to the parameters to be changed and then call **clSetParams()**.

The function prototypes are:

#### where:

handle is a handle that identifies a compressor or decompressor.

# paramValuebuffer

is a pointer to an array of pairs of *ints*. The even elements of this array select the parameters to be read or changed. The subsequent odd elements are the current or new values of these parameters.

bufferLength

is the number of *ints* in the buffer pointed to by *paramValuebuffer*.

Alternatively, parameters can be changed individually with clSetParam() and clGetParam(). clGetParam() returns the current value of the parameter. clSetParam() returns the previous value of the parameter.

# The function prototypes are:

```
int clGetParam(CLhandle handle, int paramID)
int clSetParam(CLhandle handle, int paramID, int value)
```

where:

handle is a handle that identifies a compressor or decompressor.

paramID is the identifier of the parameter to get or set.

value is the new value of the parameter.

Example 25-3 demonstrates how to extract the current value of specific parameters from a list of parameters returned as a group. In this case, the current block size and preroll values are obtained from the list of parameters that are returned in *paramValuebuffer* from **clGetParams()**.

# **Example 25-3** Getting the Current Values of Selected Parameters

```
#include <dmedia/cl.h>
...
/* Get the block size and preroll */
int paramValueBuffer[][2] = {
CL_BLOCK_SIZE, 0,
CL_PREROLL, 0
};
clGetParams(handle, (int *)paramValueBuffer,
sizeof(paramValueBuffer) / sizeof(int));
/* paramValueBuffer[0][1] is the block size */
/* paramValueBuffer[1][1] is the preroll */
```

#### **Getting or Setting the Value of a Floating Point Parameter**

Some parameters, such as CL\_COMPRESSION\_RATIO and CL\_FRAME\_RATE, are floating point values. You don't have to cast expressions involving floating point values, because macros are provided within *libcl* that handle the conversions for you; even though a value is a *float* you can cast to an *int*. To set a floating point value, use the macro CL\_TypeIsInt(); to retrieve a floating point value, use the macro CL\_TypeIsFloat().

The argument must be a variable, because the type definitions in /usr/include/ dmedia/cl.h are:

Example 25-4 demonstrates how to use the *libcl* macros to get/set a floating point parameter value.

# **Example 25-4** Using Macros to Get or Set the Value of a Floating Point Parameter

```
float number;
number = 3.0;
...
clSetParam(handle, CL_COMPRESSION_RATIO,
CL_TypeIsInt(number));
number =
CL_TypeIsFloat(clGetParam(handle,CL_COMPRESSION_RATIO));
```

#### **Getting or Setting Individual Parameter Attributes**

You can query parameters individually to get the name, defaults, and range of valid values, given the parameter identifier and a handle.

Use **clGetName()** to return a pointer to a null-terminated string that supplies the English name of a parameter. Its function prototype is:

```
char* clGetName(CLhandle handle, int param)
where:
handle is a handle that identifies a compressor or decompressor.
param is a parameter identifier.
```

Use **clGetDefault()** to return the default value of the parameter specified by *param*. Use **clSetDefault()** to set the default value. Setting the default value is particularly useful when an algorithm has been added and new defaults need to be set.

# The function prototypes are:

```
int clGetDefault(CLhandle handle, int param)
int clSetDefault(int scheme, int paramID, int value)
```

#### where:

handle is a handle that identifies a compressor or decompressor.

paramID is a parameter identifier.

*scheme* is the identifier of the scheme for which to set the defaults.

value is the new default value associated with param.

Example 25-5 demonstrates how to get and set defaults for a parameter. In this case, the default for the CL\_ORIGINAL\_FORMAT parameter is set to CL\_RGBX for the specified decompressor.

# **Example 25-5** Getting and Setting Parameter Defaults

```
#include <dmedia/cl.h>
int default;
...
clOpenDecompressor(scheme, &handle);
...
default = clGetDefault(handle, CL_ORIGINAL_FORMAT);
clSetDefault(scheme, CL_ORIGINAL_FORMAT, CL_RGBX);
...
```

Use **clGetMinMax()** to get the maximum and minimum values for a parameter. Use **clSetMin()** and **clSetMax()** to set new minimum and maximum parameter values, or to establish the minimum and maximum values when adding a new algorithm.

# The function prototypes are:

#### where:

handle is a handle that identifies a compressor or decompressor.

paramID is a parameter identifier.

minParam is a pointer to the parameter into which clGetMinMax() can

write the minimum value associated with *paramID*.

maxParam is a pointer to the parameter into which clGetMinMax() can

write the maximum value associated with paramID.

scheme is the identifier of the scheme that is to have its minimum or

maximum value changed.

min is the new minimum value associated with paramID.max is the new maximum value associated with paramID.

Example 25-6 demonstrates how to get and set the minimum and maximum values of a particular parameter for the specified compressor or decompressor.

# **Example 25-6** Getting and Setting Minimum and Maximum Parameter Values

```
#include <dmedia/cl.h>
int oldMin, oldMax;
...
clOpenDecompressor(scheme, &handle);
6
...
clGetMinMax(handle, CL_ORIGINAL_FORMAT, &oldMin, &oldMax);
clSetMin(scheme, CL_ORIGINAL_FORMAT, CL_RGB);
clSetMax(scheme, CL_ORIGINAL_FORMAT, CL_RGB332);
...
```

# **Using Frame Type Parameters**

Some compression algorithms do not allow direct compression or decompression of an arbitrary frame. These algorithms—MPEG, CCITT H.261, and so on—have blocks of frames, where each frame can be decompressed only if all previous frames in the block have been decompressed. The frame at the beginning of the block is called a *keyframe*.

A frame can be queried for its status as a keyframe by using the CL\_FRAME\_TYPE state parameter. Legal values are CL\_KEYFRAME (or CL\_INTRA), CL\_PREDICTED, and CL\_BIDIRECTIONAL. Predicted frames use information from a previous keyframe, bidirectional frames use information from both previous and future reference frames, where a reference frame is either of the other two types—CL\_KEYFRAME or CL\_PREDICTED. The Compression Library interface allows keyframe control from the application.

Some algorithms contain only keyframes, such as JPEG, MVC1, RTR, RLE, G.711, and so on. MPEG Video is the only algorithm currently supported that has all three types of frames.

# Chapter 26

# **Customizing the Compression Library**

This chapter explains how to customize the CL, by adding:

- algorithms to accommodate new compression schemes.
- parameters to configure custom algorithms

# Customizing the Compression Library

Audio and video compression technology is constantly evolving. The Compression Library provides the flexibility to evolve with this changing environment by letting you customize and expand *libcl*. This chapter explains how to add your own algorithms to the library and how to add new state parameters to these algorithms to provide capabilities that are not addressed by the standard set of parameters. To ensure capability across applications, you must use the existing interface paradigm for your custom implementations; you cannot add to or change the API.

# In this chapter:

- "Adding Custom Algorithms to the Compression Library" on page 563 explains how to add algorithms to the CL.
- "Adding Custom Parameters to the Compression Library" on page 572 explains how to add parameters to the CL.

# **Adding Custom Algorithms to the Compression Library**

For compatibility, an algorithm that is to be added to the Compression Library must meet these requirements:

- Provide support for all three types of interfaces (single-image, sequential, and buffered) through the clCompress() and clDecompress() entry points.
- Support interfaces for the query routines **clGetParams()** and **clSetParams()**.
- Provide the ability to specify the worst-case size of the compressed data through the CL\_COMPRESSED \_BUFFER\_SIZE parameter so that the application can then allocate buffers of appropriate size.
- Report errors using **clError()**.

Use **clAddAlgorithm()** to add your own compression algorithms to *libcl*. **clAddAlgorithm()** adds compression algorithms to the library by passing function pointers to routines that are unique for each algorithm. When you call this added algorithm, some preprocessing is done, then the routines that have been passed to **clAddAlgorithm()** are called.

# The function prototype for **clAddAlgorithm()** is:

#### where:

maximumHeaderSize

is the maximum size of the stream header for the specified

algorithm.

name is a pointer to a string that contains the name of the

algorithm.

type is the type of the algorithm (CL\_AUDIO or CL\_VIDEO).

openCompressor is a pointer to the function that opens a compressor for the

new algorithm. The function must have the same

arguments as clOpenCompressor().

compress is a pointer to the function that compresses for the new

algorithm. The function must have the same arguments as

clCompress().

closeCompressor

is a pointer to the function that closes a compressor for the

new algorithm. The function must have the same

arguments as **clCloseCompressor()**.

openDecompressor

is a pointer to the function that opens a decompressor for the new algorithm. The function must have the same

arguments as clOpenDecompressor().

*decompress* is a pointer to the function that decompresses for the new

algorithm. The function must have the same arguments as

clDecompress().

closeDecompressor

is a pointer to the function that closes a decompressor for the new algorithm. The function must have the same

arguments as clCloseDecompressor().

readHeader is pointer to the function that reads the stream header for

the new algorithm. The function must have the same

arguments as clReadHeader().

*queryScheme* is a pointer to the function that identifies the scheme from

the stream header for the new algorithm. The function must

have the same arguments as **clQueryScheme()**.

*queryLicense* is a pointer to the function that determines whether there is

a license for the new algorithm. The function must have the

same arguments as **clQueryLicense()**.

getParams is a pointer to the function that gets compressor or

decompressor parameters for the new algorithm. The function must have the same arguments as **clGetParams()**.

setParams is a pointer to the function that sets compressor or

decompressor parameters for the new algorithm. The function must have the same arguments as **clSetParams()**.

compressionScheme

is a pointer to a value that is to receive the compression

scheme identifier.

Argument bounds checking is performed before these functions are called. For example, if **clCompress()** is called, each argument is checked for validity before passing control to the function passed to **clAddAlgorithm()**.

Added algorithms must support interfaces to **clGetParams()** and **clSetParams()**, which are supplied to **clAddAlgorithm()**. The algorithm implementation is notified of changes to, or requests for, parameter values

through these routines. This allows complete control of parameters to constrain or report errors upon setting or to calculate only when requested.

Algorithms and parameters, once added, will show up when queried with **clQueryAlgorithms()** and **clQueryParams()** respectively.

The algorithm implementation is required to specify the worst-case size of the compressed data for a frame through the CL\_COMPRESSED\_BUFFER\_SIZE parameter. This value must be calculated every time it is requested, using the current value of other parameters such as CL\_IMAGE\_WIDTH, CL\_HEIGHT, and CL\_INTERNAL\_FORMAT.

An algorithm can also specify a value for its most natural number of frames to process at a time in the CL\_BLOCK\_SIZE parameter.

The number of blocks that need to be processed before decompressed data begins to emerge is specified in the CL\_PREROLL parameter. For algorithms with a fixed compression ratio, this may allow the application to use the sequential interface.

Use **clSetUnique()** and **clGetUnique()** to allow the algorithm implementation to store and retrieve algorithm-specific information that is associated with each instantiation of a compressor or decompressor.

When **clOpenCompressor()** or **clOpenDecompressor()** is called, the implementation sets up the *unique* pointer that gets stored associated with the compressor handle. Other functions that require the information get it using that handle. **clSetUnique()** returns the previous unique pointer. **clGetUnique()** returns the current unique pointer.

# The function prototypes are:

```
void * clSetUnique(CLhandle handle, void *unique)
void * clGetUnique(CLhandle handle)
```

#### where:

handle is a handle to a compressor or decompressor.

*unique* is a pointer to unique data that is associated with *handle*.

When adding an algorithm, use **clSetMin()** and **clSetMax()** to set its minimum and maximum values, respectively. For example, you may need to set the bounds that define the legal range of the compression ratio. These settings take effect when either **clOpenCompressor()** or **clOpenDecompressor()** is called.

When adding an algorithm, use **clSetDefault()** to set defaults. For example, you may need to specify a default compression ratio. These defaults take effect when either **clOpenCompressor()** or **clOpenDecompressor()** is called.

Example 26-1 demonstrates how to add algorithms to the Compression Library.

**Example 26-1** Adding Algorithms to the Compression Library

```
#include <cl.h>
int
        scheme;
/* Add the new algorithm */
clAddAlgorithm("New Algorithm", CL_VIDEO,
    NEW_ALGORITHM_MAX_HEADER_SIZE,
    OpenNewCompressor, CompressNew, CloseNewCompressor,
    OpenNewDecompressor, DecompressNew, CloseNewDecompressor,
    ReadNewHeader, QueryNewScheme, QueryLicense,
GetNewParams,
    SetNewParams, &newScheme);
/* Compress a series of frames (same as always) */
clOpenCompressor(newScheme, &handle);
for(i = 0; i < numberOfFrames; i++)</pre>
    /* Get a frame from somewhere */
    clCompress(handle, i, 1, frameBuffer,
&compressedDataSize,
             compressedData);
    /* Write the compressed data to somewhere else. */
clCloseCompressor(handle);
```

# **Managing Buffers for Added Algorithms**

When you add an algorithm, you must mirror the normal use of the buffer management calls, that is, the calls for compression and decompression are swapped and the calling order is reversed. Example 26-2 sets up decompression buffering for added algorithms.

# **Example 26-2** Decompression Buffering

```
until numberOfFrames frames are decompressed:
   until space for a frame is available:
        actualNumberOfFrames = clQueryFree(frameHdl, 1, &frameBuffer, &wrap);
   until a frame is decompressed (and the compressed data is available):
        actualBufSize = clQueryValid(dataHdl, bufSize, &buf, &wrap);
        /* Decompress the data in "buf" and place the result in "frameBuffer" *
        actualSize = clUpdateTail(dataHdl, actualBufSize);
        actualNumberOfFrames = clUpdateHead(frameHdl, numberOfFrames);
```

Example 26-3 sets up decompression buffering for added algorithms.

#### **Example 26-3** Compression Buffering

```
until numberOfFrames frames are compressed:
   until a frame is available:
        actualNumberOfFrames = clQueryValid(frameHdl, 1, &frameBuffer, &wrap);
   until a frame is compressed (and space for the compressed data is available):
        actualSize = clQueryFree(dataHdl, size, &buf, &wrap);
        /* Compress the frame in "frameBuffer" and place the result in "buf" */
        actualLen = clUpdateHead(dataHdl, len);
        actualNumberOfFrames = clUpdateTail(frameHdl, numberOfFrames);
```

# **Reading Data Across Buffer Discontinuities**

When **clDecompress()** is called with non-NULL pointers for the *compressedData* or *frameBuffer* arguments, the data is available through the buffer management calls, so no special code is required for that case; however, care must be taken not to wait for data that will never arrive. For example, if insufficient data is passed into the *compressedData* or *frameBuffer* parameters, you don't want the application to block and wait forever for data.

To avoid this situation for *compressedData*, you can use **clReadData()**, which provides an interface that removes the need for the application to know about the discontinuity of the compressed data caused by using a ring buffer. Its function prototype is:

where:

bufferHdl is a handle to a compressor/decompressor.

requestedDataSize is the size of the requested data.

compressedData is a pointer to the returned pointer to the compressed

data.

Because it is often not known what the size of the compressed data is, clReadData() allows the algorithm to request data of arbitrary size, such as the next piece of data that it knows it needs. When the requested data crosses a discontinuity, it is automatically pieced together in a temporary buffer. A pointer to this temporary buffer is returned. If the size of the requested data is larger than what is present in the buffer, the routine blocks until the data arrives. Alternatively, if the compressed data were passed directly to clDecompress(), no more data would arrive, no matter how long it waited and whatever data was available would be returned.

**clReadData()** calls **clQueryValidData()** and **clUpdateTail(dataHdl)**. It blocks (unless the compressed data was passed directly to **clDecompress()**) until the requested amount of data has accumulated and, if necessary (at the end of the ring buffer), copies the data into a temporary buffer, to guarantee one contiguous buffer.

**clReadData()** returns the actual number of bytes read. **clDone()** returns the actual number of bytes updated. The requested data size is always returned unless there is an error. If the data requested crosses the discontinuity from the end to the beginning of the ring buffer, a temporary buffer is automatically created, the data from the ring buffer is copied to it, and its address is returned in the *compressedData* argument.

An algorithm has two parts: the compressed data and the bitstream that encapsulates it. For some algorithms, such as JPEG and MPEG, the bitstream is fairly complex and must be parsed in very small segments. clReadData()

is designed to be very efficient and can be used to read many small segments of a few bytes if so desired.

Use **clDone()** to update the consumed data read by **clReadData()**. Its function prototype is:

int clDone(CLbufferHdl bufferHdl, int amountToUpdate)

where:

bufferHdl is a handle to a compressor or decompressor.

requested Data Size

is the size of the requested data.

compressedData

is a pointer to the returned pointer to the compressed data.

amountToUpdate

is the amount that was consumed from the last read and therefore needs to be updated with a call to **clUpdateTail()**. If -1 is passed, the amount returned by the last call to **clReadData()** is used.

In each call to **clReadData()**, **clUpdateTail()** is called to release data from the previous call to **clReadData()**, and **clQueryValid()** is called to get the new data. **clDone()** is used at the end of the decompress routine (just before returning) to call **clUpdateTail()** for data used from the last read.

New algorithms should report errors with **clError()**. Generally, the format string starts with the routine name within which the error occurred, followed by a description of the error.

The buffer architecture for adding algorithms is shown in Figure 26-1. The routines called by the compressor and decompressor are shown.

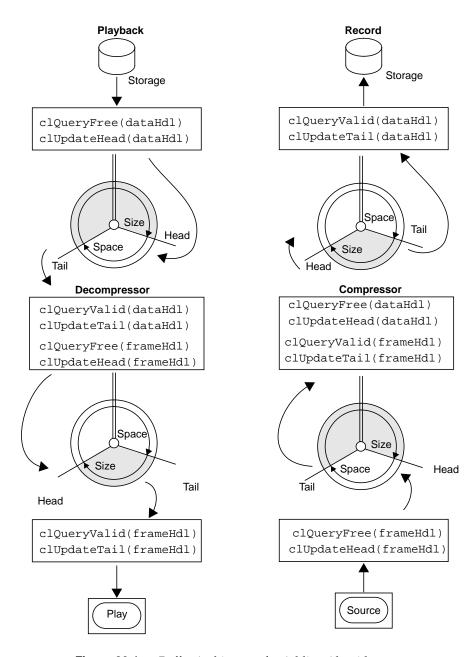


Figure 26-1 Buffer Architecture for Adding Algorithms

## **Adding Custom Parameters to the Compression Library**

You can add audio or video compression parameters to *libcl*. This is useful when using **clAddAlgorithm()** to add a new algorithm that uses parameters that don't exist in the default set of compression parameters. The application uses the new parameters as it would any of the other compression parameters. The functions for the new compression algorithm access the parameters in the same way as the application.

Use **clAddParam()** to add parameters to the library. Its function prototype is:

#### where:

scheme is the compression scheme to add a parameter to.

name is a pointer to a string that contains the name of the

parameter.

is the type of the parameter: CL\_ENUM\_VALUE,

CL\_RANGE\_VALUE, CL\_POINTER, CL\_FLOATING\_ENUM\_VALUE, or CL\_FLOATING\_RANGE\_VALUE.

min is the minimum value of the parameter.max is the maximum value of the parameter.initial is the default value of the parameter.

paramID is a pointer to an int value to receive the compression

parameter identifier.

The code fragment in Example 26-4 adds a new video algorithm to the CL.

### **Example 26-4** Adding Parameters to the Compression Library

```
#include <cl.h>
int paramID;
/* Add a new algorithm */
clAddAlgorithm("New Algorithm", CL_VIDEO,
   NEW_ALGORITHM_MAX_HEADER_SIZE,
   OpenNewCompressor, CompressNew, CloseNewCompressor,
   OpenNewDecompressor, DecompressNew, CloseNewDecompressor,
   ReadNewHeader, QueryNewScheme, GetNewParams,
SetNewParams,
   &newScheme);
/* Add the new parameter */
clAddParam(newScheme, "New Parameter", CL_RANGE_VALUE, 0,
100,
      75, &paramID);
/* Compress a series of frames (same as always) */
clOpenCompressor(newScheme, &handle);
clSetParam(handle, paramID, 55);
```

#### **PART SIX**

## **Movie Programming**

Chapter 27, "Introduction to the Movie Library," introduces the Movie Library and describes its applications and features.

Chapter 28, "Getting Started with the Movie Library," explains how to set up, compile, and debug Movie Library applications.

Chapter 29, "File I/O and Editing Movies with the Movie Library," explains how to perform movie file I/O and how to edit movies.

Chapter 30, "Playing Movies with the Movie Library," describes the Movie Library playback and event-handling facilities.

Chapter 31, "Using the Movie Library with QuickTime Movies," describes basic concepts for working with QuickTIme movies, and then it explains how to add QuickTime capability to a Movie Library application. It also describes the optional QuickTime compressor Library, which provides access to QuickTime compressors for Movie Library applications.

Chapter 32, "Using the Movie Library Sample Programs," describes the Movie Library sample programs.

## Chapter 27

# Introduction to the Movie Library

This chapter introduces the Movie Library, which provides a file-format-independent API for reading, writing, editing, and playing movies.

## Introduction to the Movie Library

The Movie Library is a collection of routines within the IRIS Media Libraries<sup>TM</sup> that provides a C language application programming interface (API) for reading, writing, editing, and playing movies on Silicon Graphics workstations. The API provides a uniform interface to movies of various formats and lets you convert movies from one format to another. This chapter describes the features and applications of the Movie Library.

## **Overview of Movie Library Features and Applications**

This section provides a quick overview of the features and applications of the Movie Library. You need not have specialized knowledge about digital media or synchronization methods to use the Movie Library.

## **Movie Library Features**

The main features of the Movie Library (*libmovie*) are:

- the ability to read, write, and play movie files
- a file-format-independent API
- file format conversion capabilities
- support for Silicon Graphics Movie format, versions 2.0 and 3.0
- support for Apple® Computer QuickTime™ movie format
- data compression and decompression
- asynchronous playback support
- flexible playback control
- support for movies embedded in applications software

## **Movie Library Applications**

You can easily integrate playback or creation of movies into your existing application without making extensive changes to the application's main event loop, or you can incorporate movies into an application that doesn't already have an event loop, because the Movie Library uses its own event queue.

Application categories include:

• 3D graphics animation

You can use the Movie Library to save 3D graphics images as a movie that can be used to provide rapid playback of computationally intensive graphics scenes.

scientific visualization

You can use the Movie Library to assemble a series of images from an IRIS Explorer<sup>TM</sup> application into a movie in order to look for patterns that aren't apparent when looking at the still images one by one.

• computer-based training (CBT)

You can use the Movie Library to play embedded movies to illustrate complicated procedures in interactive tutorials.

live video recording

You can use the Movie Library to convert video input from your video option board (IndigoVideo, Sirius Video , or Galileo Video , or Video from Indy Video into a movie.

digital video editing systems

You can build video editing applications on top of the low-level data handling routines provided by the Movie Library.

## **Using the Movie Library with Other Silicon Graphics Libraries**

Other libraries that your application can use in conjunction with the Movie Library include:

- Video Library
- ImageVision Library
- IRIS GL or OpenGL
- Audio File Library

The Movie Library uses the capabilities of the following libraries to play movies:

- Audio Library
- Compression Library
- Digital Media Library
- IRIS GL or OpenGL

Your application need not include the header file for the Digital Media Library, because it is included in *movie.h.* Normally, your application need not include the header files for the Audio Library, Compression Library, OpenGL, or IRIS GL unless it uses those libraries directly.

## Chapter 28

# **Getting Started with the Movie Library**

This chapter explains basic concepts for programming with the Movie Library, including movie file formats, movie and track properties and parameter getting and setting.

## Getting Started with the Movie Library

This chapter explains how to begin developing a Movie Library application. It presents terminology and describes the steps involved in developing, compiling, and debugging a Movie Library application. It also explains how to work with movie and track properties.

## **Movie Library Basics**

This section defines basic terms and explains fundamental concepts that you need to know in order to use the Movie Library, including the Movie Library programming model and movie file formats.

### **Definitions**

The definitions in this section provide a foundation for working with the Movie Library.

A digital *movie* is a file that contains tracks of image and audio (optional) data.

A *track* represents a data stream. Each track can contain only one type of medium, either an image sequence or an audio soundtrack. Currently, a maximum of one image track and one audio track is allowed.

Just as motion picture film is actually a sequence of individual photographs, the image track of a movie file is actually a sequence of still images, such as illustrations, camera images, or computer graphics, contained within individual *image frames* of uniform height and width. Some movies have special image frames called *keyframes* that contain information for a block of frames that is treated as a single unit.

The audio track contains digitized audio (samples) of narration, music, sound effects, and so on, that are synchronized to the frames of the image track.

An image frame contains a single image, composed of individual pixels. An audio frame is composed of one audio sample for each audio channel.

The *frame count* of a movie is the total number of image frames in that movie. Image frames are numbered from zero to one less than the frame count.

Figure 28-1 is a conceptual illustration of *somersault.mv*, a movie with one image track, a stereo audio track, and 13 frames (only 10 are shown).

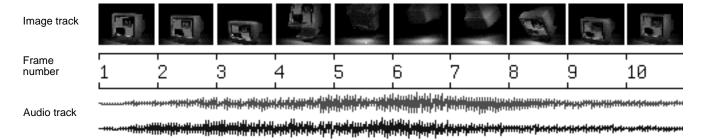


Figure 28-1 Typical Movie: somersault.mv

The *frame rate* of a movie refers to the number of image frames played per unit length of time. A typical frame rate is 15 frames per second (FPS), which means that each frame is displayed for 1/15 of a second.

Looping is the process of repeatedly playing a frame, a sequence of frames, or an entire movie. A movie loop is a useful display method for graphical sequences that cannot be rendered in real time or for any sequenced image data. When playing a movie, the movie loop can include the entire movie or a fragment of the movie, but you can only have one loop per movie. You cannot have a track loop. The Movie Library has three *loop modes*:

- nonlooping (play once through)
- looping (continuous play)
- swinging (repeatedly playing forward then backward)

## **Movie Library Programming Model**

The Movie Library programming model consists of:

- movie instances, which are handles to movie files
- tracks, which contain audio or images
- parameter-value lists, which contain movie and track properties

A movie instance is a handle that allows you to read, write, edit, and play a movie file. It contains information about the different tracks (audio and image) in a movie and is identified by a movie ID similar in nature to a file descriptor. An application can create several movie instances at once.

#### **Movie File Formats**

The Movie Library provides a file-format-independent API and lets you convert movies from one format to another. Currently supported formats are the Silicon Graphics movie format and the Apple Computer QuickTime movie format.

#### **Silicon Graphics Movie Formats**

Silicon Graphics uses a proprietary movie file format that has evolved over time to the current 3.0 version. You can read movies that use previous versions (1.0 and 2.0) of the file format—for example, movies that were created using the prerelease alphas of the Movie Library or previous releases of Movie Maker—but the Movie Library writes only Silicon Graphics version 3.0 movie files.

The Silicon Graphics version 3.0 movie format is a parameterized file format with data that is normally interleaved. It currently consists of a single image track and an optional audio track.

#### Apple Computer QuickTime File Format

See Chapter 31, "Using the Movie Library with QuickTime Movies," for information on working with QuickTime movies and on using the optional Silicon Graphics QuickTime Compressor Library, which provides Movie Library support for both Apple Animation and Apple Video compression.

See *Inside Macintosh: QuickTime* for information about developing QuickTime applications, and see the *QuickTime Starter Kit User's Guide* for information on how to use the Apple QuickTime utilities on the Macintosh.

### **Deciding Which Format to Use**

The destination system determines which movie format to choose when writing the final version of a movie file. Use the QuickTime format only if you require cross-platform compatibility between Silicon Graphics and Apple Macintosh computers.

## **Developing a Movie Library Application**

This section provides a nutshell description of how to develop a Movie Library application, including compiling and debugging instructions.

A good way to get started with the Movie Library is to look at the sample programs in the <code>/usr/people/4Dgifts/examples/dmedia/movie</code> directory. They demonstrate creating, editing, playing, and getting information about movies. See Chapter 32, "Using the Movie Library Sample Programs," for complete descriptions of the sample programs.

### **Outline for Developing a Movie Library Application**

Follow these steps to develop a Movie Library application:

- Open an existing movie file by calling either mvOpenFile(), mvOpenMem(), or mvOpenFD() or create a new movie instance by calling mvCreateFile(), as described in Chapter 29, "File I/O and Editing Movies with the Movie Library."
- 2. Once you have a movie instance, you can find existing tracks, add tracks to the movie, and delete tracks from the movie. If, in step 1, you created a new movie rather than open an existing movie, you must add an image track to it if one does not already exist.
- 3. Read and write images and audio from and to the movie tracks using the Movie Library file I/O functions, such as **mvReadFrames()** and **mvInsertFrames()** that are described in Chapter 29.

- 4. Use the Movie Library editing functions to efficiently implement editing tasks such as deleting part of a movie by calling mvDeleteFrames(), and copying data from one movie to another by calling mvPasteFrames(). Editing is described in Chapter 29.
- 5. To play a movie, configure a playback window and use **mvBindWindow()** to associate a movie with the playback window. You can play more than one movie simultaneously, in separate windows or in the same window. Playback is described in Chapter 30, "Playing Movies with the Movie Library." You can't edit a movie that you have opened for playback, so write the file before attempting to play it.
- When you have finished with a movie, call mvClose() to destroy the movie instance.

## **Compiling and Linking a Movie Library Application**

Movie Library applications must include the *movie.h* header file to obtain definitions for the library. When compiling your Movie Library application, follow these linking requirements in the order specified:

```
-lmovie -ldmedia -lcl -lawareaudio
```

### where:

-lmovie links with the Movie Library (libmovie)

-ldmedia links with the DM Library (libdmedia), which provides

parameter setting and ring buffering capabilities

-1cl links with the Compression Library, which provides

compression capability

**-lawareaudio** links with the part of the Compression Library that

provides Aware audio compression-decompression (codec) support. The part of the CL that provides Aware software codecs requires separate linking to accommodate end-user licensing. Currently, applications are required to link with this library even if it is not used, but licensing is optional for

the end user.

If your application uses the Movie Library to play movies, you must add the following to the end of the link line:

```
-lgl -laudio -lX11
```

#### where:

-lgl links with the IRIS GL

-laudio links with the Audio Library

-1x11 links with the X11 library

## **Debugging a Movie Library Application**

The Movie Library has two facilities designed to assist you in debugging your application:

- a debugging version of the Movie Library, libmovie\_d.a, which checks for improper usage of the Movie Library, such as out-of-bounds values and invalid parameters
- environment variables that allow you to emulate certain types of failures such as I/O errors, without actually causing failures

#### Using the Debugging Version of the Movie Library

The debugging version of the Movie Library checks for violations by setting assertions that rigidly state the requirements for a parameter or value. One typical assertion, worded in standard English, is: "Check that the current frame number is within the bounds of the number of frames in the movie," which is more precisely stated in the following assertion:

```
ASSERT( 0 <= frame < trackLength, "Frame number out of range")
```

To use the debugging version of the library, link with *libmovie\_d.a* as follows:

```
-lmovie_d
```

and then run your program.

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, *libmovie.a*, because the runtime checks imposed by the debugging library cause an undesirable performance and size overhead for a packaged application.

#### **Emulating I/O Failures**

When attempting to read data from a device such as a CD-ROM drive or over the network, two types of problems can occur:

- lost connections—either the CD (or other storage medium) is missing or defective or, when reading data over the network, the remote host is unreachable
- delays—either the seek time is too long or network delays are occurring

You can test how your application handles these types of I/O failures by setting the following environment variables to emulate failures without causing actual failures:

MVPLAYDBG\_VIDREAD simulates I/O errors when reading video
MVPLAYDBG\_AUDREAD simulates I/O errors when reading audio
MVPLAYDBG\_VIDNFS simulates delays when reading video
MVPLAYDBG\_AUDNFS simulates delays when reading audio

## **Setting and Getting Movie Properties**

Movies have certain inherent properties that provide information about their contents. Global movie properties apply to a movie as a whole; track properties apply only to tracks. Tracks have general properties that apply to all types of tracks, and specific properties that depend on the track medium.

Movie and track properties are stored in parameters, some of which you can set, others of which are read-only, because they are computed by the Movie Library from available information. You can also define your own parameters to represent properties not provided by the Movie Library.

This section explains how to set and get movie and track parameters using the universal mvSetParams() and mvGetParams() routines; then it explains how to use the Movie Library convenience routines for setting and getting individual movie and track parameters and for setting up defaults.

## Setting and Getting Movie and Track Parameters

The type MVid is used for both movies and for tracks so that universal set and get functions can work on both movies and tracks. The Movie Library also provides convenience routines for setting and getting movie parameters, described in "Setting and Getting Global Movie Properties" on page 593, and setting and getting track parameters, described in "Setting and Getting Track Properties" on page 601.

To set or change a group of parameters for a movie or track, make a parameter-value list containing the names of the parameters you want to set and the values you want them to have, then pass this list to **mvSetParams()**. Its function prototype is:

#### where:

movieOrTrack is the movie or the track whose parameters you want to set

params is a pointer to a parameter-value list that contains the

parameters and settings you want

### returnParamsSetOrNull

is a pointer to a parameter-value list into which the Movie Library loads those parameters and values that it recognized and was able to set; if <code>returnParamsSetOrNull</code> is NULL, the Movie Library will not return such a list

**Note:** Some parameters cannot be changed and some are not recognized by the Movie Library; verify the results by checking the list returned in *returnParamsSetOrNull*.

To retrieve a list of the current parameters for a given movie or track, call **mvGetParams()**. Its function prototype is:

```
DMparams* mvGetParams ( MVid movieOrTrack )
```

**mvGetParams()** returns the parameter list associated with the given movie or track. For movies, the parameter list contains the movie property settings: file format, looping mode, and so on. For tracks, the parameter list contains information about the format of the data in the track.

**Note:** The application should not call **dmParamsDestroy()** on the returned parameter-value list. The Movie Library reuses the same structure for each call to **mvGetParams()**.

See "Displaying Movie Parameters" in Chapter 32 for an example program called *mvinfo* that uses **mvGetParams()** to extract and print a list of movie and track parameters.

## **Setting and Getting Global Movie Properties**

Global movie properties apply to the movie as a whole.

You can set these global movie properties:

- comment
- loop limit
- loop mode
- title

The read-only (computed by the Movie Library) global movie properties are:

- file format, which is established when the movie is created
- optimization

You can set and get global movie properties individually, using the Movie Library mvSetParams() and mvGetParams() routines or the DM Library dmSetParams() and dmGetParams() routines.

You can make a default parameter-value list to use when creating a movie by calling **mvSetMovieDefaults()**, as described in "Creating a Default Movie Configuration" on page 596. If you set the default movie parameters by calling **mvSetMovieDefaults()**, then other parameters, such as the title, require setting only if they differ from the defaults.

This section explains how to set and get the global movie properties individually using Movie Library convenience routines that are built on top of the mvSetParams() and mvGetParams() calls.

## **Setting and Getting the Movie Comment**

To store a comment string in a movie file, call **mvSetComment()**. Its function prototype is:

```
DMstatus mvSetComment( MVid movie, const char* comment )
```

To retrieve the comment string that is stored in a movie file, call **mvGetComment()**. Its function prototype is:

```
const char* mvGetComment( MVid movie )
```

### **Setting and Getting the Default Movie Loop Mode**

The Silicon Graphics movie file format lets you store a default loop mode setting within a movie file. When opening a movie file, the Movie Library obtains the default loop mode from the movie if it is set; otherwise, it assumes MV\_LOOP\_NONE is the default loop mode.

To store a default loop mode setting in a movie file, call **mvSetLoopMode()**. Its function prototype is:

```
DMstatus mvSetLoopMode( MVid movie, MVloopmode mode )
```

#### where:

*mode* determines the default loop mode:

MV\_LOOP\_NONE sets the default mode to

play the movie once

MV\_LOOP\_CONTINUOUSLY sets the default mode to

play the movie repeatedly

MV\_LOOP\_SWINGING sets the default mode to

play the movie forward,

then backward repeatedly

You can set the playback loop mode independently of the default loop mode by using **mvSetPlayLoopMode()**, as described in "Looping" in Chapter 30.

To retrieve the loop mode setting that is stored in a movie file, call **mvGetLoopMode()**. Its function prototype is:

```
MVloopmode mvGetLoopMode( MVid movie )
```

#### **Setting and Getting the Default Movie Loop Limit**

You can store a default loop limit setting in the movie file that defines the default for the number of movie loops a movie will play if its loop mode is set to MV\_LOOP\_CONTINUOUSLY or MV\_LOOP\_SWINGING. To store a default loop limit in a movie file, call **mvSetLoopLimit()**. Its function prototype is:

```
DMstatus mvSetLoopLimit( MVid movie, int limit )
```

#### where:

limit

is the number of movie loops allowed; a value of 0 means that the movie will loop indefinitely

To retrieve the default loop limit setting that is stored in a movie file, call **mvGetLoopLimit()**. Its function prototype is:

```
int mvGetLoopLimit( MVid movie )
```

#### **Setting and Getting the Movie Title**

To store a title string in a movie file, call **mvSetTitle()**. Its function prototype is:

```
DMstatus mvSetTitle( MVid movie, const char* title )
```

To retrieve the title string that is stored in a movie file, call **mvGetTitle()**. Its function prototype is:

```
const char* mvGetTitle( MVid movie )
```

#### **Getting the Movie File Format**

To retrieve the file format of a movie file, call **mvGetFileFormat()**. Its function prototype is:

```
MVfileformat mvGetFileFormat( MVid movie )
```

mvGetFileFormat() returns the file format of the movie:

MV\_FORMAT\_SGI\_1 Silicon Graphics version 1 format MV\_FORMAT\_SGI\_2 Silicon Graphics version 2 format MV\_FORMAT\_SGI\_3 Silicon Graphics version 3 format

MV\_FORMAT\_QT Apple QuickTime format

#### **Getting the Movie Optimization Setting**

To determine whether a movie is optimized for playback, call **mvGetOptimized()**. Its function prototype is:

```
DMboolean mvGetOptimized( MVid movie )
```

**mvGetOptimized()** returns DM\_TRUE if the movie is optimized for playback performance and DM\_FALSE if it is not. Performing any editing operations on the movie that read, write, insert, or delete frames will disrupt the optimization and clear this flag. See "Optimizing a Movie File" in Chapter 29 for more information about movie optimization.

### Creating a Default Movie Configuration

To make a default parameter-value list that can be used to create a movie, call **mvSetMovieDefaults()**, which takes the desired format for the new movie file and sets the rest of the movie parameters (such as loop mode) to their default values. Its function prototype is:

```
DMstatus mvSetMovieDefaults ( DMparams* params, MVfileformat format)
```

*format* sets the file format of the movie:

MV\_FORMAT\_SGI\_3 Silicon Graphics version 3 format

MV\_FORMAT\_QT Apple QuickTime format

Table 28-1 lists the parameters and values set by **mvSetMovieDefaults()**.

**Table 28-1** Movie Defaults

Parameter	Value Type	Default
MV_COMMENT	String	Empty
MV_FILE_FORMAT	MVfileformat	format
MV_LOOP_LIMIT	Integer	0
MV_LOOP_MODE	MVloopmode	MV_LOOP_NONE
MV_OPTIMIZED	DMboolean	DM_FALSE

Example 28-1 is a code fragment that creates a parameter-value list called *params*, then initializes *params* to the movie defaults by calling **mvSetMovieDefaults()**, and then passes *params* to **mvCreateFile()** to configure a new movie file. After the movie has been created, the parameter-value list is destroyed by calling **dmParamsDestroy()**.

**Example 28-1** Creating and Initializing a Default Movie Parameter-value List

```
MVid movie;
DMparams* params;

if ( dmParamsCreate( &params ) != DM_SUCCESS ) {
    printf( "Out of memory.\n" );
    exit( 1 );
    }

if ( mvSetMovieDefaults(params, MV_FORMAT_SGI_3) !=

DM_SUCCESS ) {
    printf( "Out of memory.\n" );
    exit( 1 );
    }

if ( mvCreateFile("temp.mv", params, NULL, &movie) !=

DM_SUCCESS ) {
    printf( "Could not create movie.\n" );
    exit( 1 );
    }

dmParamsDestroy( params );
```

## Adding Your Own Parameters to the Movie Library

You can add your own movie parameters to the Movie Library to represent movie and track properties that do not exist in the parameters provided by the Movie Library. These parameters are global, meaning that any movie opened from within your application will recognize and have access to the parameters defined by the application. User-defined parameters exist as long as the application is running.

To add your own parameters to the Movie Library, call **mvAddUserParam()**. Its function prototype is:

```
DMstatus mvAddUserParam ( const char* paramName )
```

#### where:

paramName

is a string containing 15 or fewer characters that uniquely describes the parameter; for example, use a company abbreviation as a prefix for the parameter name

**Note:** Only null-terminated strings of less than 32 K bytes can be used for user-defined parameter-value pairs.

Once the parameters have been added, you can set and get them just like the regular Movie Library parameters. To set the values for the parameters, pass a list of the parameters and values to **mvSetParams()**.

You must add the parameters before calling any of the routines for setting and getting their values. If you call **mvSetParams()** before calling **mvAddUserParam()**, the Movie Library will not recognize the parameters and they will be ignored.

If you open a movie that contains user-defined parameters, you can read those parameters because the Movie Library automatically calls **mvAddUserParam()** when loading a movie with user-defined parameters.

Example 28-2 is a code fragment that adds a user-defined global movie parameter named *Newparam* to a Movie Library application, then creates a movie that has access to the user-defined parameter.

**Example 28-2** Adding a User-Defined Global Movie Parameter

```
* Adding a global (movie) user param named Newparam with a value of
* "Movie type code 1113afq1" to a new movie file.
  DMparams* movieParams;
  if ( dmParamsCreate( &movieParams ) != DM_SUCCESS ) {
       fprintf( stderr, "Unable to create default params.\n");
       exit( EXIT_FAILURE );
  if ( mvSetMovieDefaults( movieParams, MV_FORMAT_SGI_3 )
       != DM_SUCCESS ) {
       fprintf( stderr, "Unable to set default params.\n");
       dmParamsDestroy( movieParams );
       exit( EXIT_FAILURE );
  if ( mvAddUserParam( "Newparam" ) != DM_SUCCESS ) {
       fprintf( stderr, "Unable to add user param.\n");
       dmParamsDestroy( movieParams );
       exit( EXIT_FAILURE );
  if ( dmParamsSetString( movieParams, "Newparam",
                          "Movie type code 1113afq1" )
                          != DM_SUCCESS ) {
        fprintf( stderr, "Unable to set user param.\n");
       dmParamsDestroy( movieParams );
        exit( EXIT_FAILURE );
  if ( mvCreateFile( "mymovie", movieParams,
                     NULL, theMovie ) == DM_FAILURE ) {
       fprintf( stderr,
               "Unable to create movie file %s: error = %s.\n",
               getOutMovieName(),mvGetErrorStr( mvGetErrno() ) );
       dmParamsDestroy( movieParams );
       exit( EXIT_FAILURE );
  }
```

Example 28-3 is a code fragment that adds a user-defined image track parameter named *NewImageParam* to a Movie Library application, and then adds an image track that has access to the user-defined parameter.

**Example 28-3** Adding a User-Defined Image Track Parameter for a New Track

```
* Adding a usr param named NewImageParam with a value of
* "Source code a32bg" to an image track.
  DMparams *imageTrackParams;
  if ( dmParamsCreate( &imageTrackParams ) != DM_SUCCESS ) {
       fprintf( stderr, "Unable to create image track params.\n");
       exit( EXIT_FAILURE );
   if ( dmSetImageDefaults( imageTrackParams, imgWidth, imgHeight,
                           DM_PACKING_RGBX ) != DM_SUCCESS ) {
       fprintf( stderr, "Unable to set image defaults.\n");
       dmParamsDestroy( imageTrackParams );
       exit( EXIT_FAILURE );
   if ( mvAddUserParam( "NewImageParam" ) != DM_SUCCESS ) {
       fprintf( stderr, "Unable to add user param.\n");
       dmParamsDestroy( imageTrackParams );
      exit( EXIT_FAILURE );
  if ( dmParamsSetString( imageTrackParams, "NewImageParam",
                          "Source code a32bg" ) != DM_SUCCESS ) {
       fprintf( stderr, "Unable to set user param.\n");
       dmParamsDestroy( imageTrackParams );
       exit( EXIT_FAILURE );
   }
   * Add the image track to the movie.
   * /
  MVid imageTrack;
  if ( mvAddTrack( movie, DM_IMAGE, imageTrackParams,
                   NULL, &imageTrack ) == DM_FAILURE) {
       fprintf( stderr, "Unable to add image track to movie.\n");
       dmParamsDestroy( imageTrackParams );
       exit( EXIT_FAILURE );
  dmParamsDestroy( imageTrackParams );
```

## **Setting and Getting Track Properties**

The Movie Library lets you work with the individual image and audio tracks inside movies. A track is an evenly spaced (in time) sequence of frames, where each of the frames is the same size (although the amount of data per frame may vary after compression).

This section explains how to use the Movie Library routines for setting and getting both general track properties, which apply to all tracks, and specific track properties, which depend upon the track medium.

You can set this general track property for all types of tracks:

SMPTE time code string

The read-only general track properties are:

- track length (number of frames)
- track medium (image or audio)

## **Setting and Getting General Track Properties**

This section explains how to set and get the general track properties using the Movie Library convenience routines, which are built on top of the universal set and get routines **mvSetParams()** and **mvGetParams()**. You can also set and get general track properties by calling **mvSetParams()** and **mvGetParams()**, respectively.

#### Setting and Getting SMPTE Time Code Strings Stored in Tracks

You can store SMPTE time codes as strings in a track parameter. The time code string does not have any relationship to a timer inside the Movie Library, so you have to obtain the time code from an external source before storing it as a parameter. See "Using the SMPTE Time Code Sample Application" in Chapter 32 for sample programs that implement time code capability.

To store a SMPTE time code string in a movie track, call **mvSetSMPTEStart()**. Its function prototype is:

DMstatus mvSetSMPTEStart(MVid movie, const char\* start\_time)

where:

start\_time

is a string representing a SMPTE time code, with two digits each for hour, minute, second, and frame, as in "00:59:30:00"

mvSetSMPTEStart() returns either DM\_SUCCESS or DM\_FAILURE.

To retrieve a SMPTE time code string that is stored in a movie track, call mvGetSMPTEStart(), which returns the starting time code as set by **mvSetSMPTEStart()**. Its function prototype is:

```
const char* mvGetSMPTEStart( MVid track )
```

where:

track

is the track for which you want to obtain the SMPTE time

code parameter

#### **Getting the Track Length**

To retrieve the number of frames stored in a track, call **mvGetTrackLength()**. Its function prototype is:

```
MVframe mvGetTrackLength ( MVid track )
```

#### **Getting the Track Medium**

To determine the track medium, call **mvGetTrackMedium()**. Its function prototype is:

```
DMmedium mvGetTrackMedium( MVid track )
```

mvGetTrackMedium() returns either DM\_IMAGE or DM\_AUDIO.

## **Setting and Getting Audio Track Properties**

The audio track of a movie contains digitized audio samples.

When creating an audio track, you can set and get audio properties individually, using the Movie Library mvSetParams() and mvGetParams() routines or the DM Library dmSetParams() and dmGetParams() routines.

You can also create default audio parameters for an audio track using the DM Library dmSetAudioDefaults() routine. If you set the audio defaults by calling dmSetAudioDefaults(), then other properties, such as number of channels, require setting only if they differ from the defaults.

You can set these properties when creating an audio track:

- audio sample format
- audio sample rate
- audio sample width (bytes per sample)
- number of audio channels

The only parameter that you can change for an existing audio track is the default volume (this is not functional in the current version of the Movie Library). The other parameters cannot be changed after a track has been created because they would invalidate the data stored in the track.

Table 28-2 lists the audio defaults as set by **dmSetAudioDefaults()**.

**Table 28-2** Audio Defaults

Parameter	Value Type	Default
DM_MEDIUM	DMmedium	DM_AUDIO
DM_AUDIO_WIDTH	Integer	width
DM_AUDIO_FORMAT	DMaudioformat	DM_AUDIO_TWOS_COMPLEMENT
DM_AUDIO_RATE	Double	rate
DM_AUDIO_CHANNELS	Integer	channels
DM_AUDIO_COMPRESSION	String	DM_AUDIO_UNCOMPRESSED

**Note:** Currently, only DM\_AUDIO\_UNCOMPRESSED is supported for audio compression.

Since the Movie Library guarantees that all audio tracks have all of these parameters defined, the "get" routines in this section do not return errors.

#### Setting and Getting the Default Volume of an Audio Track

**mvSetDefaultVol()** stores a new value, ranging from 0.0 to 1.0, for the default volume parameter. It returns DM\_SUCCESS or DM\_FAILURE. Its function prototype is:

```
int mvSetDefaultVol( MVid audioTrack double volume)
```

**mvGetDefaultVol()** returns the default volume setting to use when playing this movie. Its function prototype is:

```
DMstatus mvGetDefaultVol( MVid audioTrack )
```

#### where:

audioTrack is the audio track whose volume you want to set or get

**Note:** The Movie Library currently ignores the default volume setting.

#### **Getting the Audio Sample Width of an Audio Track**

The Movie Library supports 8-bit and 16-bit audio samples.

To determine the number of bits used to store each audio sample in an audio track, call **mvGetAudioWidth()**. Its function prototype is:

```
int mvGetAudioWidth( MVid audioTrack )
```

#### where:

audioTrack

is the audio track whose sample width you want to know

#### **Getting the Audio Sample Rate of an Audio Track**

To determine the audio sample rate (in samples per second) for an audio track, call **mvGetAudioRate()**. Its function prototype is:

```
double mvGetAudioRate( MVid audioTrack )
```

where:

audioTrack is the audio track whose sample rate you want to know

### **Getting the Number of Audio Channels in an Audio Track**

To determine the number of audio channels in an audio track, call **mvGetAudioChannels()**. Its function prototype is:

```
int mvGetAudioChannels( MVid audioTrack )
```

**mvGetAudioChannels()** returns the number of audio channels: 1 for mono; 2 for stereo (the Movie Library does not currently support 4-channel audio tracks).

where:

audioTrack is the audio track for which you want to know the number

of channels

### **Getting the Audio Format of an Audio Track**

The Movie Library supports both unsigned and twos complement audio formats that are directly understood by the Audio Library (AL), in particular the **ALwritesamps()** call, for Silicon Graphics native movie formats.

Audio for QuickTime movies is stored using sampling rates and formats understood by QuickTime on the Apple Macintosh computer. Some of these formats are not directly supported by the AL. See Chapter 31, "Using the Movie Library with QuickTime Movies," for more information on QuickTime audio.

To determine the format used to store the audio samples in an audio track, call **mvGetAudioFormat()**. Its function prototype is:

```
DMaudioformat mvGetAudioFormat( MVid audioTrack )
```

where:

audioTrack is the audio track whose format you want to know

**mvGetAudioFormat()** returns either DM\_AUDIO\_TWOS\_COMPLEMENT or DM\_AUDIO\_UNSIGNED.

#### **Getting the Audio Compression Scheme of an Audio Track**

To determine the audio compression scheme for an audio track, call **mvGetAudioCompression()**. Its function prototype is:

```
const char* mvGetAudioCompression( MVid audioTrack )
```

where:

audioTrack is the audio track whose compression you want to know

Since movies do not currently support audio compression, the only valid value is DM\_AUDIO\_UNCOMPRESSED.

## **Setting and Getting Image Track Properties**

The image track of a movie contains a sequence of image frames. All the images in an image track must have the same frame size (width and height).

When creating an image track, you can set and get image properties individually, using the Movie Library mvSetParams() and mvGetParams() routines or the DM Library dmSetParams() and dmGetParams() routines.

You can also create default image parameters for an image track using the DM Library dmSetImageDefaults() routine. If you set the image defaults by calling dmSetImageDefaults(), then other properties, such as compression, require setting only if they differ from the defaults.

Table 28-3 lists the image defaults as set by **dmSetImageDefaults()**.

Table 28-3Image Defaults

Parameter	Value Type	Value
DM_MEDIUM	DMmedium	DM_IMAGE
DM_IMAGE_WIDTH	Integer	width
DM_IMAGE_HEIGHT	Integer	height
DM_IMAGE_RATE	Double	15.0
DM_IMAGE_COMPRESSION	String	DM_IMAGE_UNCOMPRESSED
DM_IMAGE_INTERLACING	DMinterlacing	DM_IMAGE_NONINTERLACED
DM_IMAGE_PACKING	DMpacking	packing
DM_IMAGE_ORIENTATION	DMorientation	DM_BOTTOM_TO_TOP

The only parameter that you can change for an existing image track is the image frame rate. The other parameters cannot be changed after a track has been created because they would invalidate the data stored in the track.

Since the Movie Library guarantees that all image tracks have all image parameters defined, the "get" routines in this section do not return errors.

## **Setting and Getting the Image Frame Rate**

To change a movie's frame rate by storing a new value for the image rate parameter, call **mvSetImageRate()**. Its function prototype is:

#### where:

is the image track for which you want to set the rate

framesPerSecond is the image rate in frames per second

**mvSetImageRate()** returns DM\_SUCCESS if the image rate was successfully changed; otherwise, it returns DM\_FAILURE.

To determine the image rate in frames per second for a movie, call **mvGetImageRate()**. Its function prototype is:

```
double mvGetImageRate( MVid imageTrack )
```

#### where:

*imageTrack* is the image

is the image track whose rate you want to know

## **Getting the Image Frame Size**

To get the width (*x* dimension) in pixels of the images stored in an image track, call **mvGetImageWidth()**. Its function prototype is:

```
int mvGetImageWidth( MVid imageTrack )
```

#### where:

imageTrack

is the image track whose dimensions you want to know

To get the height (*y* dimension) in pixels of the images stored in an image track, call **mvGetImageHeight()**. Its function prototype is:

```
int mvGetImageHeight( MVid imageTrack )
```

#### where:

imageTrack

is the image track whose dimensions you want to know

#### **Getting the Image Orientation**

The ordering of scan lines within an image frame depends upon the source of the data. Bottom-to-top is the default data orientation for Movie Library routines. To find out how pixels are ordered, call **mvGetImageOrientation()**. Its function prototype is:

```
DMorientation mvGetImageOrientation ( MVid imageTrack)
```

#### where:

imageTrack

is the image track whose format you want to know

**mvGetImageOrientation()** returns the order in which scan lines are stored in the image: DM\_BOTTOM\_TO\_TOP or DM\_TOP\_TO\_BOTTOM.

#### **Getting the Image Format**

The Movie Library can read images from and write images to a buffer. You can use the IRIS ImageVision Library to read any of these image formats into a buffer:

FIT Silicon Graphics ImageVision Library tiled image format
GIF Compuserve<sup>™</sup> Graphics Image File, a popular 8-bit format

PCD Kodak Photo CD<sup>™</sup> file format for digital images

SGI Silicon Graphics image data formats: .bw, .rgb, .rgba, .sgi, and

.screen

TIFF extended version of Tag Image File Format, Revision 6.0

#### **Getting the Image Packing Format**

The image *packing format* describes how pixels are packed within images; for example, the word length used to represent the pixel, and the bit order.

Images in a movie file can be stored on disk in any of the image packing formats listed in Table 28-4. When you play a movie whose image track is in a nondefault format, the Movie Library converts the image data to DM\_PACKING\_RGBX on the fly as it plays the movie.

To get the image packing format for a particular image track, call **mvGetImagePacking()**. Its function prototype is:

DMpacking mvGetImagePacking( MVid imageTrack )

#### where:

imageTrack is the ID of the image track whose format you want to know

Table 28-4 lists the image packing formats supported by the Movie Library.

**Table 28-4** Image Packing Formats

Packing Format	Description
DM_PACKING_RGBX	R, G, B, and X (don't care) data are packed into a 32-bit word as 0xXXBBGGRR. This is the default format for the Movie Library.
DM_PACKING_GRAYSCALE	Each pixel is an 8-bit luminance value.
DM_PACKING_APPLE_16	16-bit images are stored with 5 bits each of red, green, and blue. The bit layout is: XRRRRGGGGGBBBBB
	Apple stores images from top to bottom, while SGI goes from bottom to top.
DM_PACKING_APPLE_32	32-bit images are stored with 8 bits each of red, green, and blue. The bit layout is: XXXXXXXXXRRRRRRRRRGGGGGGGBBBBBBBB

## **Getting the Image Compression Scheme**

Compression settings can result in significant differences in image quality and in playback performance. The most commonly used compression schemes for movies to be played on Silicon Graphics computers are MVC1 and MVC2; QT\_VIDEO is frequently used for QuickTime movies to be played on a Macintosh computer.

The Movie Library supports image compression through the Compression Library. The compression methods that have been thoroughly tested with the Movie Library are UNCOMPRESSED, MVC1, MVC2, RLE24, and JPEG. You can also use the QuickTime compression schemes, QT\_VIDEO and QT\_ANIM, in QuickTime and Silicon Graphics movies if you have purchased and installed the Silicon Graphics QuickTime Compressor Library.

To get the image compression scheme, call **mvGetImageCompression()**. Its function prototype is:

const char\* mvGetImageCompression( MVid imageTrack )

where:

in in image Track is the ID of the image track whose format you want to know

mvGetImageCompression() returns the name of the compression scheme that is used to compress the images.

The most commonly used schemes and the reasons for choosing them are:

UNCOMPRESSED For the best quality in a final movie, all image manipulation and storage should be with

uncompressed images until the final movie is produced, at which time the images can be compressed. Repeatedly compressing, decompressing, and then recompressing images

reduces the image quality.

MVC1 This is a good general-purpose compression scheme. It is a color-cell compression

technique that works well for video, but can cause fuzzy edges in high-contrast animation.

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.

RLE24 This is a lossless compression method that uses run-length encoding (RLE). Run-length

encoding compresses images by storing a color and its run-length (the number of pixels of that color) every time the color changes. It is a good technique for animations where there

are large areas that have identical colors.

JPEG is designed for still images and is usable, but slow, for video. 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.

QT\_VIDEO This is the compression used for QuickTime movies that contain video. Like MVC1,

QT\_VIDEO is a color-cell compression technique. It includes temporal compression and is

good for video and reasonable for animation.

QT\_ANIM This is a lossy run-length encoding scheme used for QuickTime movies. It also includes

temporal compression. It has excellent compression ratios for animation that has large

areas of similar colors.

## Chapter 29

# File I/O and Editing Movies with the Movie Library

This chapter explains how to perform file I/O and editing with the Movie Library, including adding and deleteing audio and images, and copying and pasting movie frames.

# File I/O and Editing Movies with the Movie Library

This chapter describes how to set up a Movie Library application and how to use the Movie Library routines for handling file input/output (I/O), editing, compression, and other basic tasks. Playback is discussed in the next chapter.

The Movie Library provides these basic file I/O capabilities:

- creating new movie files
- opening an existing movie from a file descriptor, filename, or memory
- reading data from and writing data to selected tracks and frames

## **Initializing a Movie Library Application**

This section explains the basic program setup you will use. To set up a Movie Library application, you need to:

- open a file descriptor for the movie (if you are using a file descriptor)
- create a new movie or open an existing movie
- test a file to determine whether it is a movie file
- set and get movie properties
- add, remove, and find tracks
- allocate buffers for data passed to and from the Movie Library

Each of these tasks is described in detail in the subsections that follow. It's a good idea to familiarize yourself with the setup procedures described in this section because you'll be performing most of these tasks in every application that works with movies.

Basic routines for creating, verifying, and opening movies are available for each type of interface: filename, file descriptor, and memory-mapped files. Each of these routines features a similar API but has the appropriate arguments for the specified method.

Figure 29-1 shows a diagram of the file I/O routines in the Movie Library.

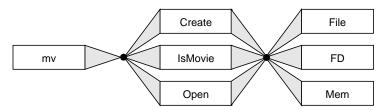


Figure 29-1 Movie Library File I/O Routines

mvCreateFile(), mvCreateFD(), and mvCreateMem() create a new empty
movie, initialized with the given parameters, as set by
mvSetMovieDefaults() or by the DM Library routines, and return an
identifier for the new movie. Any movie that was already present in the file
or memory location will be destroyed.

mvIsMovieFile(), mvIsMovieFD(), and mvIsMovieMem() test whether a movie is present in a file or in memory. Only movies in supported formats (Silicon Graphics and QuickTime) are recognized.

mvOpenFile(), mvOpenFD(), and mvOpenMem() read an existing movie, create a movie instance in memory that holds information about it, and then return an identifier for the new movie.

The following two calls are used when the file I/O or editing is completed:

mvWrite()	flushes all changes that have been made to a movie and
	makes sure that they are written to the file, but does not
	close the file

mvClose() flushes all changes that have been made to the movie and makes sure that they are written to the file, and then destroys the movie instance

Each of these calls is described in detail in the sections that follow.

## **Using File Descriptors with Movies**

File descriptors can be used to work with movie files on disk, CD-ROM, or DAT, or with embedded or previously opened movie files.

**Note:** The Movie Library does not support embedded QuickTime files.

Use the IRIX **open()** system call (see the *open*(2) man page) to open a file descriptor.

The movie instance inherits one of three file access modes, which are associated with the file descriptor:

O\_RDONLY opens a read-only movie file
O\_WRONLY opens a write-only movie file
O\_RDWR opens a read-write movie file

The file access mode determines which operations are possible for a particular movie. You can't write data to a movie opened as a read-only file. Similarly, you can't play a movie that has been opened for writing.

Before opening the movie, the file pointer associated with the file descriptor should be positioned at the beginning of the movie file. If the movie data is embedded within a file, such as an application-specific file format containing a movie as a data chunk, use the IRIX **lseek()** system call (see the *lseek(2)* man page) to seek the file descriptor to the beginning of the movie data. For example, a movie file might be embedded in the file of a word-processing program. When the word processing program is ready to access the movie, it seeks to the position of the movie within its file and passes the file descriptor (*fd*) to the Movie Library.

## **Creating a New Movie**

You can create a new movie and associate it with a file descriptor, file name, or memory location. There is a function call corresponding to each type.

mvCreateFD() creates a movie using a file descriptor returned by open()

mvCreateFile() creates a movie using a file namemvCreateMem() creates a memory-mapped movie

## The function prototypes are:

#### where:

is a file descriptor returned by the *open*(2) system call that is

assigned to the new movie

filename is a filename assigned to the new movie

pointer is a pointer to a memory location that is assigned to the new

movie

size is the size of the block of memory to use—the Movie Library

will not read or write beyond this block size

params is a pointer to parameters that describe the movie attributes

returnParamsSetOrNull

is a pointer to a parameter-value list into which the Movie

Library loads those parameters and values that it

recognized and was able to set; if returnParamsSetOrNull is

NULL, the Movie Library will not return such a list

returnMovie is a pointer into which the movie identifier is returned

Example 29-1 is a code fragment that shows how to create a movie.

## **Example 29-1** Creating a Movie

```
#include <movie.h>
void CreateMovie()
   DMparams* params;
   MVid
             movie;
   if ( dmParamsCreate( &params ) != DM_SUCCESS ) {
       /* handle error */
    }
    if ( mvSetMovieDefaults( params, MV_FORMAT_SGI_3 )
        != DM_SUCCESS ) {
        /* handle error */
   }
    if ( mvCreateFile( "new-movie", params, NULL, &movie )
        != DM_SUCCESS ) {
        /* handle error */
    }
    /* Add tracks, insert frames, etc. */
}
```

## **Checking for the Presence of a Movie**

Before attempting to open an existing movie file from a file descriptor, file name, or memory, your application should check to see whether a movie is present. There is a separate function call to verify the existence of a movie for each source type.

mvIsMovieFD()	checks for a movie identified by the given file descriptor
mvIsMovieFile()	checks for a movie identified by the given file name
mvIsMovieMem()	checks for a movie stored in the given memory location

#### Their function prototypes are:

```
DMboolean mvIsMovieFD ( int fd )
DMboolean mvIsMovieFile ( const char* filename )
DMboolean mvIsMovieMem ( void* pointer, size_t size )
```

#### where:

fd is the file descriptor of the movie file you are checking

filename is the name of the movie file you are checking

pointer is a pointer to a memory location where the movie file you

are checking is stored

size is the size of the block of memory to use—the Movie Library

will not read or write beyond this block size

DM\_TRUE is returned if the given file is a movie file; otherwise, DM\_FALSE is returned.

The following code fragment determines whether a given file is a movie file and prints an error message if it is not:

```
if ( !mvIsMovieFile( filename ) ) {
    PrintError ("Not a movie file.");
}
```

## **Opening an Existing Movie**

Movies can be opened from file descriptors, filenames, or memory. There is a function call corresponding to each source type:

**mvOpenFD()** opens a movie using a file descriptor that has already

been obtained from another source, such as *open*(2)

mvOpenFile() opens a movie from a file namemvOpenMem() opens a memory-mapped movie

#### Opening a Movie from a File Descriptor

To open a movie file from a file descriptor, call **mvOpenFD()**. Its function prototype is:

```
DMstatus mvOpenFD (int fd, MVid* returnMovie)
```

where:

is a file descriptor that references the movie file you want to

open

returnMovie is a pointer into which the movie identifier is returned

## Opening a Movie from a Filename

To open a movie from a filename, call **mvOpenFile()**. Its function prototype is:

where:

filename is the filename of the movie file you want to open

oflag is the file access mode, either O\_RDONLY or O\_RDWR

returnMovie is a pointer into which the movie identifier is returned

#### **Opening Memory-mapped Movies**

To open a movie that resides entirely in memory, beginning at the location pointed to by *pointer*, call **mvOpenMem()**. Its function prototype is:

where:

pointer is a pointer to the starting memory location of a moviesize is the size of the memory buffer—the Movie Library will not

read or write beyond this block size

returnMovie is a pointer into which the movie identifier is returned

Your application must allocate and free the memory buffer used by **mvOpenMem()**.

## Adding, Locating, and Deleting Audio and Image Tracks

The operations that can be performed on a track include:

- adding a new track
- removing an existing track
- finding a track
- mapping frames between tracks in the same movie or another movie
- reading and writing data in a track
- editing operations (copying from one movie to another)

## Adding an Audio or Image Track to a Movie

To add a track to a movie, call **mvAddTrack()**. Its function prototype is:

DMstatus mvAddTrack ( MVid movie, DMmedium medium, DMparams\* params,

DMparams\* returnParamsSetOrNull, MVid\* returnTrack )

where:

medium is the type of track, either DM\_AUDIO or DM\_IMAGE

params is a pointer to a parameter-value list for configuring the new

track, which should be filled in using either

dmSetImageDefaults() or dmSetAudioDefaults(),

depending on the medium type

returnParamsSetOrNull

is a pointer to a parameter-value list into which the Movie

Library loads those parameters and values that it

recognized and was able to set—if returnParamsSetOrNull is

NULL, the Movie Library will not return such a list

returnTrack is a pointer into which the ID of the newly added track is returned

DM\_SUCCESS is returned if the new track was created successfully; otherwise DM\_FAILURE is returned.

The DM\_IMAGE\_COMPRESSION parameter that is given when creating an image track is important to get right, because different settings can result in large differences in image quality and in playback performance. See "Getting the Image Compression Scheme" in Chapter 28 for information on choosing a compression setting when configuring an image track.

Example 29-2 shows how to add an audio track to a movie.

## **Example 29-2** Adding an Audio Track to a Movie

```
#include <movie.h>
void AddAudioTrack( MVid movie )
    DMparams* audioParams;
    MVid
              audioTrack;
    if ( dmParamsCreate( &audioParams ) != DM_SUCCESS ) {
        /* handle error */
    if ( dmSetAudioDefaults( audioParams,
                             8, /* bits per sample */
                             22050, /* sample rate */
                                  /* number of channels */
                             ) != DM_SUCCESS ) {
        /* handle error */
    if ( mvAddTrack( movie,
                     DM_AUDIO,
                     audioParams,
                     NULL,
                     &audioTrack ) != DM_SUCCESS ) {
         /* handle error */
    /* Write audio data to track */
    /* with mvInsertFrames(3mv) */
}
```

## Removing an Audio or Image Track from a Movie

To remove a track from a movie, which deletes all the data from the track, call **mvRemoveTrack()**. Its function prototype is:

```
DMstatus mvRemoveTrack(MVid movie, MVid track)
```

#### where:

is the track you want to remove

DM\_SUCCESS is returned if the track was removed successfully; otherwise, DM\_FAILURE is returned.

## **Locating an Existing Track**

To get a handle for a track that already exists, call **mvFindTrackByMedium()**. Its function prototype is:

#### where:

medium is the medium, either DM\_AUDIO or DM\_IMAGEreturnTrack is a pointer into which the track identifier is returned

**mvFindTrackByMedium()** returns DM\_SUCCESS if a track of the given *medium* exists in the movie instance identified by *movie*; otherwise, it returns DM\_FAILURE.

## Mapping Frames from One Track to Another Track

Because the image and audio tracks are separate, your application must manage the synchronization between tracks when editing movies; for example, when deleting a portion of the image track, the corresponding portion of the audio track must be located and also deleted. Figure 29-2 shows an image track and a corresponding audio track. (The audio track would actually have many more samples per frame, but for clarity, only a few samples are shown.)

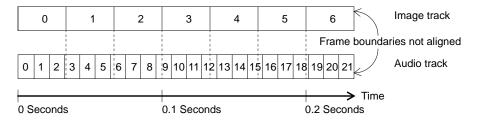


Figure 29-2 Mapping Frames from One Track to Another

As shown in Figure 29-2, the frame numbers in one track do not have a one-to-one time correspondence with the frame numbers in another track. When mapping frame numbers from one track to another, the Movie Library chooses the frame that matches the starting time of the given frame. The frame boundaries are not necessarily aligned, so the mapping can differ by one frame, depending on which track you are mapping from. For example, in Figure 29-2, frame 3 in the image track maps to frame 9 of the audio track, but frame 9 of the audio track maps to frame 2 of the image track.

Before performing any operation on a frame in one track that affects its corresponding frame in the second track, you must locate the frame number in the second track that corresponds to the frame number in the first track.

To locate the frames in one track that correspond (in time) to frames from another track, call **mvMapBetweenTracks()**. **mvMapBetweenTracks()** determines which frame number in *toTrack* corresponds to the frame numbered *fromFrameIndex* in *fromTrack* and writes the result into the location pointed to by *toFrameIndex*. Its function prototype is:

```
DMstatus mvMapBetweenTracks ( MVid fromTrack, MVid toTrack, MVframe fromFrameIndex, MVframe* toFrameIndex )
```

where:

fromTrack

is the track for which you want to locate a corresponding frame in *toTrack* 

*toTrack* is the track in which to locate the frame number

corresponding to the frame numbered from Frame Index in

fromTrack

fromFrameIndex

is the frame number in *fromTrack* for which you want to locate the corresponding frame number in *toTrack* 

toFrameIndex is a pointer into which the frame number in toTrack that

corresponds to the frame numbered fromFrameIndex in

fromTrack is written

DM\_SUCCESS is returned if a corresponding frame was located; otherwise, DM\_FAILURE is returned.

**Tip:** You can also use **mvMapBetweenTracks()** to find corresponding frame numbers in tracks from two different movies.

## **Editing Movies**

The Movie Library provides these editing operations:

- optimizing a movie
- inserting raw image or audio frames from a buffer into a track
- reading frames from a track into a buffer
- deleting frames from a track
- · reading and inserting compressed images directly
- copying frames from one movie to another

**Note:** Movies should not be edited during playback.

When you edit a movie, the Movie Library changes pointers to frames rather than operating on the actual frames themselves. After a series of editing calls, the movie frames might not be arranged in the order in which they play, and there are probably empty spaces in the movie. Such a movie does not provide optimum playback, but you can optimize the movie as described in "Optimizing a Movie File" on page 627 before closing it.

## Optimizing a Movie File

To get the best playback performance from an edited movie, you should optimize the movie file after an editing session. Optimization streamlines the movie file by coalescing the empty space and by flattening the data structure into the most linear structure possible. This is especially helpful for minimizing the excessive seeks that occur during playback that are caused by editing a movie file repeatedly. Optimization does not occur in place; instead, the Movie Library makes a copy of the movie to optimize.

To optimize a movie, call **mvOptimize()**. Its function prototype is:

```
DMstatus mvOptimize ( MVid fromMovie, MVid toMovie )
```

#### where:

fromMovie is the movie you want to optimize toMovie is a name for the optimized movie

## Using a Buffer for Editing

This section explains how to use a buffer for editing. The routines described in this section work on uncompressed data; there is a similar group of routines described in "Reading and Inserting Compressed Images" on page 631 for working with compressed data.

#### **Allocating Buffers**

Memory must be allocated to hold audio or image data that is passed to or obtained from the Movie Library. The buffer that is passed to these routines points to a block of memory that holds an array of frames. Your application is responsible for allocating a buffer large enough to hold the desired number of frames.

**Note:** A playback-only application that does not perform any file I/O operations need not allocate separate memory.

Use dmImageFrameSize() to determine the number of bytes needed to hold one frame of raw image data; similarly, use dmAudioFrameSize() to determine the number of bytes needed to hold one frame of raw audio data.

Before allocating memory, determine the required buffer size as demonstrated in Example 29-3. Allocate the appropriate amount of memory by using one of the IRIX system calls for memory allocation, such as **malloc()**, and then check the **malloc()** return to make sure there was enough memory. See the *malloc(3X, 3C)* man page for information about memory allocation. See the *IRIX Programming Guide* for information about using shared memory.

To determine the buffer size needed to store the uncompressed frames, multiply the number of frames by the frame size, as shown in Example 29-3.

**Example 29-3** Determining What Size Buffer to Allocate

```
static void insertFrames( MVid theEditMovie,
                         MVid theEditTrack,
                          MVid theSourceMovie,
                          MVid theSourceTrack )
{
   MVframe editStartFrame;
   MVframe sourceStartFrame;
   MVframe numFrames = getNumEditFrames();
   size_t insBuffSize;
   void *insBuff;
   if ( getEditTrackType() == DM_IMAGE ) {
        insBuffSize = numFrames *
                    dmImageFrameSize( mvGetParams( theSourceTrack ) );
   }
   else if ( getEditTrackType() == DM_AUDIO ) {
        insBuffSize = numFrames *
                    dmAudioFrameSize( mvGetParams( theSourceTrack ) );
   }
   insBuff = malloc( ( int ) insBuffSize );
   if ( insBuff == NULL ) {
       fprintf( stderr, "%s: Unable to allocate insert buffer.\n",
                getProgramName() );
       exit( EXIT_FAILURE );
    /* insert frames using mvInsertFrames(3mv) */
```

#### Inserting Raw Images and Audio from a Buffer into an Existing Track

You can insert raw image or audio data into an existing movie track—the Movie Library compresses the data as it is inserted into the track. To insert frames from a buffer into a track, call **mvInsertFrames()**. Its function prototype is:

#### where:

is the track into which you want to insert data

frameIndex is the frame in front of which you want to insert data

frameCount is the number of frames to insert

bufferSize is the size of the buffer

buffer is a pointer to a buffer that contains the data you want to

insert into the track

When you insert frames into an existing track, the new frames are inserted in front of *frameIndex*. The existing frames immediately following the insertion point, including frame *frameIndex*, are shifted to the right to make room for the new frames.

Figure 29-3 shows two frames (N1 and N2) inserted at *frameIndex* 5 into a movie with 7 frames. Frames 5 and 6 move to make room for the new frames.

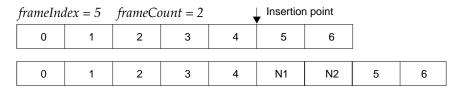


Figure 29-3 Inserting Frames into a Track

To achieve an effect similar to overwriting the existing data, you must first delete the unwanted frames by calling **mvDeleteFrames()** before inserting the new frames.

#### Reading Frames from a Movie into a Buffer for Uncompressed Data

To read a specified number of frames from an existing movie into a buffer that you have allocated for storing movie data, call **mvReadFrames()**. Its function prototype is:

 $\label{eq:def:def:DMstatus} \mbox{ mvReadFrames ( MVid track, MVframe frameIndex, MVframe frameCount, }$ 

size\_t bufferSize, void\* buffer)

where:

track is the track from which you want to read dataframeIndex is the first frame in the sequence that is to be read

frameCount is the number of frames of data to read

bufferSize is the size of the buffer, obtained by multiplying the number

of frames by the value returned from dmAudioFrameSize()

for the audio track or by the value returned from

**dmImageFrameSize()** for the image track

buffer is a pointer to a buffer that you have allocated for storing the

data

**Note:** The data is decompressed as it is read into the buffer. Use **mvReadCompressedImage()**, as described in "Reading and Inserting Compressed Images" on page 631 to read compressed image frames directly into a buffer.

## **Deleting Frames from a Movie Track**

To delete frames from a movie track, call **mvDeleteFrames()**:

DMstatus mvDeleteFrames ( MVid track, MVframe frameIndex, MVframe frameCount )

where:

track is the track from which you want to delete frames frameIndex is the first frame in the sequence that is to be deleted

frameCount is the number of frames of data to delete

## Reading and Inserting Compressed Images

The Movie Library has a special group of routines for working with compressed images. Performing editing operations on compressed images takes less disk space and less time than editing full resolution images. These routines operate on one frame of compressed data at a time because the size of compressed data can vary from frame to frame. Use these routines if you want to read or write compressed image data frame-by-frame, such as reading a frame at a time from a Cosmo Compress<sup>TM</sup> board into a movie.

## Reading a Compressed Image from a Movie into a Buffer

To read a compressed image from an existing movie into a buffer that you have allocated for storing compressed data, call **mvReadCompressedImage()**. Its function prototype is:

#### where:

is the movie track from which you want to read a

compressed image frame

frameIndex is the frame number of the image you want to read

bufferSize is the size of the buffer

buffer is a pointer to a buffer that you have allocated for storing a

compressed image frame

To determine the buffer size (in bytes) needed to hold a compressed image frame, call **mvGetCompressedImageSize()**. Its function prototype is:

#### where:

track is the movie track from which you want to read a

compressed image frame

frameIndex is the frame whose image size you want to know

**mvGetCompressedImageSize()** returns the number of bytes that image number *frameIndex* requires.

Example 29-4 reads a compressed image from *track* into *buffer*.

#### **Example 29-4** Reading a Compressed Image from a Movie into a Buffer

## Inserting a Compressed Image from a Buffer into an Existing Track

To insert a compressed image from a buffer into an existing image track, call **mvInsertCompressedImage()**. Its function prototype is:

```
DMstatus mvInsertCompressedImage ( MVid track,
                                         MVframe frameIndex,
                                         size_t bufferSize,
                                         void* buffer )
where:
track
                is the track into which you want to insert a compressed
                image
                is the frame number in front of which the compressed frame
frameIndex
                is to be inserted
bufferSize
                is the size of the buffer
buffer
                is a pointer to buffer that contains the compressed image
                you want to write to the track
```

When you insert a compressed frame into an existing track, the new frame is inserted in front of frame *frameIndex*. The existing frames immediately following the insertion point, including frame *frameIndex*, are shifted to the right to make room for the new frame (see Figure 29-3 on page 629). To achieve an effect similar to overwriting the existing data, you must first delete the unwanted frames by calling **mvDeleteFrames()** before inserting the new frame.

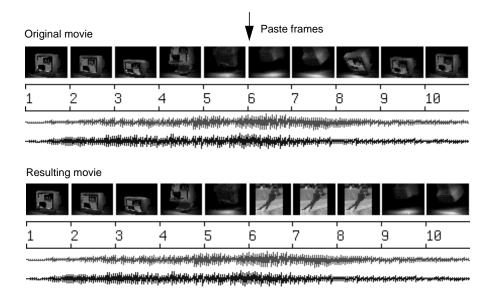
## **Copying and Pasting Frames from One Movie into Another**

The Movie Library has routines that let you copy frames from one movie track to another movie track without using a buffer. The two movies must have the same image frame rate and frame size—if they do not, an error is generated.

These routines also let you work directly with compressed data, without decompressing and recompressing the data. You can copy compressed frames from one movie into another, even if the two movies use different compression schemes.

Figure 29-4 shows image frames being pasted from one movie into another.





**Figure 29-4** Pasting Image Frames from One Movie into Another Movie

**Note:** Because a movie's audio and image tracks are independent, the audio samples associated with the original images do not shift with the image frames (see Figure 29-4). Call **mvMapBetweenTracks()** to locate the audio associated with the displaced frames, then paste the audio frames in the proper location.

To copy a range of frames from one movie and paste them into another movie without overwriting existing data, call mvPasteFrames(). Its function prototype is:

DMstatus mvPasteFrames( MVid fromTrack,

MVframe fromFrameIndex, MVframe frameCount, MVid toTrack,

MVframe toFrameIndex )

where:

fromTrack is the source movie track from which frames are copied

fromFrameIndex

is the first frame in the sequence of frames to be copied from

the source movie

frameCount is the number of frames to copy and paste

is the destination movie track into which the copied frames toTrack

is the frame in the destination movie in front of which the *toFrameIndex* 

new frames will be pasted

When frames are pasted into a non-empty movie, the new frames are inserted in front of frame frameIndex. The existing frames immediately following the insertion point, including frame frameIndex, are shifted to the right to make room for the new frames.

To overwrite the existing data, first call **mvDeleteFrames()** to delete the unwanted frames before calling mvPasteFrames().

## **Finalizing Changes and Closing Movies**

During an editing session, you can flush the changes to the file and make sure they are written into the file by calling **mvWrite()**. Data in tracks is always written immediately; **mvWrite()** flushes the header information. Its function prototype is:

```
DMstatus mvWrite( MVid movie )
```

**mvWrite()** returns DM\_SUCCESS if it was able to write the file; otherwise, DM\_FAILURE is returned.

When you have finished editing a movie, you write and close the file. You may choose to optimize the movie by calling **mvOptimize()** before closing it.

To close a movie file, call **mvClose()**, which flushes all changes that have been made to the movie and makes sure that they are written to the file, and then destroys the movie instance. Its function prototype is:

```
DMstatus mvClose( MVid movie )
```

**mvClose()** returns DM\_SUCCESS if it was able to write and close the file; otherwise, DM\_FAILURE is returned.

## Chapter 30

# Playing Movies with the Movie Library

This chapter explains the Movie Library, playback facility, including:

- handling movie events
- combining movies and graphics

# Playing Movies with the Movie Library

This chapter describes the Movie Library playback facility. It explains how to create and configure a window for playing movies and how to control and synchronize movie playback. The Movie Library can use either OpenGL or IRIS GL rendering to show movies, and you can combine the display of graphics and movies using the techniques described in this chapter.

This chapter also explains how to handle movie events in order to respond to user input or to monitor movie playback for system or I/O errors.

You can play one or more movies at a time, all in one window or in separate windows. Follow these steps to play movies:

- 1. If playing more than one movie at a time, set a hint for multiple movie playback by calling **mvSetNumMoviesHint()**.
- 2. Open the movie(s) that you want to play, using **mvOpenFile()**, **mvOpenFD()**, or **mvOpenMem()**.
- 3. Prepare a window for displaying the movie, as described in "Creating and Configuring a Playback Window" on page 641.
- 4. Set up event handling, as described in "Handling Events" on page 668.
- 5. Bind the window to the movie, as described in "Binding a Movie to a Window for Playback" on page 648. This step tells the Movie Library to display your movie inside the window you have prepared.
- 6. Show the window.
- 7. Set the movie into motion by calling **mvPlay()**, as described in "Controlling Movie Playback" on page 651.
- 8. Process movie and window events as needed, as described in "Handling Events" on page 668.
- 9. When finished, call **mvClose()** to destroy the movie instance. You must complete this step before destroying the window in which the movie is playing.

## **Opening a Movie for Playback**

If the application opens more than one movie at a time, set a hint for the Movie Library before opening any movies.

The Movie Library uses certain IRIX system facilities such as multiple processes and shared arenas whose size must be configured in advance. The Movie Library chooses defaults for these resources that are suitable for most applications; however, to play several movies at the same time, you must set a hint at initialization time to indicate how many movies are likely to be played so that the proper system resource allocations can be made.

To set a hint for the number of movies an application will play, call **mvSetNumMoviesHint()**. Its function prototype is:

DMstatus mvSetNumMoviesHint (int numMovies)

where:

numMovies is the number of movies likely to be played

To retrieve the number of movies at which the hint is currently set, call **mvGetNumMoviesHint()**. Its function prototype is:

int mvGetNumMoviesHint (void)

To open a movie for playback, call **mvOpenFile()**, **mvOpenMem()**, or **mvOpenFD()**, as described in Chapter 29, "File I/O and Editing Movies with the Movie Library." These calls return an identifier, *movieid*, for a movie instance that is used to reference the movie in subsequent Movie Library calls.

## **Creating and Configuring a Playback Window**

The Movie Library requires a mixed-model GL/X window (an X window configured for either OpenGL or IRIS GL rendering) in single-buffered RGB mode.

## **Creating a Window for OpenGL Playback**

See the OpenGL Programming Guide for information on using the OpenGL window configuration routines, including: glXgetconfig(3g), glXCreateContext(3g), and glXChooseVisual(3g).

## Creating a Window for IRIS GL Playback

There are two methods for creating a GL/X window:

- 1. Create a mixed-model window using Xlib calls. You can paste the sample code from <code>glxhelper.c</code> in <code>/usr/people/4Dgifts/examples/dmedia/movie/common/</code> into your application to create an appropriate window. See <code>Graphics Library Programming Tools and Techniques</code> for details about the calls used in <code>glxhelper.c</code>.
- 2. If you are using IRIS IM, create a GLXMDraw widget. See the *IRIS IM Programming Notes* for details.

Example 30-1 is a code fragment that shows how to create an IRIS GL playback window. (KeyReleaseMask is needed only if the application needs to listen to keystrokes when the mouse pointer is inside the movie window.)

**Example 30-1** Creating an IRIS GL Playback Window

```
* Open a connection to the X server, and create an X window
 suitable for GL rendering here.
*******
static DMstatus createXWindow( Display **dpy, Window *win,int width, int height )
   XSetWindowAttributes childWinAttribs;
     {}^{\star}{}Open a connection to the X server.
    */
    *dpy = XOpenDisplay( 0 );
   if ( *dpy == NULL ) {
        fprintf( stderr, "%s: Cannot open display.\n", programName );
       return DM_FAILURE;
   childWinAttribs.colormap = DefaultColormap( *dpy, DefaultScreen( *dpy ) );
     * Even if we don't use it, childWinAttribs.border_pixel must be something.
   childWinAttribs.border_pixel = 0;
    * Create an X window configured for GL rendering, using
    \star the helper functions defined in glxhelper.c
    *win = GLXCreateWindow( *dpy, RootWindow( *dpy, DefaultScreen( *dpy ) ),
                           100, 100, width, height, 0, CWColormap CWBorderPixel,
                           &childWinAttribs, GLXrgbSingleBuffer );
   XSelectInput( *dpy, *win, ExposureMask | StructureNotifyMask | KeyReleaseMask );
   return DM_SUCCESS;
```

# **Configuring the Playback Display**

You can control these settings for the playback display:

Background what color fills the space between the movie and the

window perimeter

View size how big the movie frame is

View offset where the movie frame is relative to the window origin

You can call the routines that configure the size, offset, and background color of the playback window whether or not a movie is currently playing within that window. Figure 30-1 diagrams the view settings.

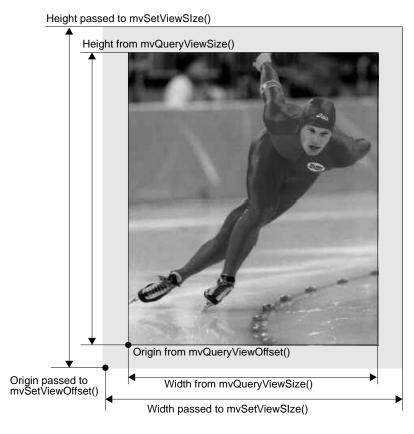


Figure 30-1 Playback View Settings

### Setting and Getting the Background Color

The Movie Library automatically centers the movie in the view you specify and fills the areas within the view that are not part of the movie frame with a background color. The default background color is "SGI light gray," which has the components (170, 170, 170).

To set the background color, call **mvSetViewBackground()**. Its function prototype is:

```
void mvSetViewBackground ( MVid movieid, unsigned short red, unsigned short green, unsigned short blue )

where:

red, green, blue are color components in the range 0-255
```

To get the background color, call **mvGetViewBackground()**. Its function prototype is:

### where:

redreturn, greenreturn, bluereturn

are pointers into which the values of the color components are returned

### **Setting and Getting the Viewing Area Size**

This section explains how to specify the size of the viewing area used for rendering a movie within a window. The default view size is the width and height of the movie frame, as it is specified in the movie file.

The Movie Library zooms the movie frame to fill the specified area as closely as possible, while adhering to the pixel zooming restraints imposed by the IRIS GL or Open GL renderer. Any remaining area between the window and the movie viewing area is filled with the background color that you specified.

**Note:** Currently, only integer zooming  $(2\times, 3\times,$ and so on) is supported, regardless of the graphics hardware platform.

To specify the size of the movie viewing area, call **mvSetViewSize()**. Its function prototype is:

#### where:

newwidth is the requested width, in pixels, of the viewing areanewheight is the requested height, in pixels, of the viewing areakeepaspect controls whether the aspect ratio is preserved when

zooming the movie:

DM\_TRUE preserves the aspect ratio

DM\_FALSE does not preserve the aspect ratio

To determine the viewing area size that the Movie Library will actually create for a desired width and height, call **mvQueryViewSize()**. Its function prototype is:

### where:

width is the requested width, in pixels, of the viewing areaheight is the requested height, in pixels, of the viewing areakeepaspect controls whether the aspect ratio is preserved when

zooming the movie:

DM\_TRUE preserves the aspect ratio

DM\_FALSE does not preserve the aspect ratio

widthreturn is a pointer into which the actual width, in pixels, of the

viewing area is returned

heightreturn is a pointer into which the actual height, in pixels, of the

viewing area is returned

If a movie appears by itself with no other graphics in a GL/X window, you might want to ensure that your code chooses the proper window size by calling **mvQueryViewSize()** in advance, and using the results to resize the GL/X window to be the same size as the movie—or you can simply choose a background color for the unused portions of the display by calling **mvSetViewBackground()**.

If the movie appears in a GL/X window along with other graphic elements, such as a movie that is embedded in a larger display, you might be more concerned about setting an exact view size, so you should set the view size carefully by first calling **mvQueryViewSize()** and then using the results as inputs to **mvSetViewSize()**. This makes sure that the Movie Library draws on the display only over regions that the movie frame actually occupies.

To get the requested view size, call **mvGetViewSize()**. Its function prototype is:

#### where:

widthreturn is a pointer into which the width is returnedheightreturn is a pointer into which the height is returned

mvGetViewSize() always returns the last requested size of the movie view,
regardless of the movie's actual view size. Thus, you can inspect both the
requested size by calling mvGetViewSize(), and the actual frame size by
calling mvQueryViewSize().

# **Setting and Getting the Viewing Location Offset**

This section explains how to set and get the movie viewing location offset, as measured from the window origin.

IRIS GL and the X Window System differ in where they assign the coordinates of the origin (0,0). IRIS GL defines (0,0) as the lower left corner of the screen; the X coordinate system defines (0,0) as the upper left corner of the screen. The default offset is (0,0) in the IRIS GL coordinate system.

To set the viewing location offset, call **mvSetViewOffset()**. Its function prototype is:

where:

offset x is the offset from the origin in the x (horizontal) dimension offset y is the offset from the origin in the y (vertical) dimension

glcoordsystem is a boolean that you use to specify the screen coordinate

system for the *y* offset

If *glcoordsystem* is DM\_TRUE, the Movie Library assumes that you are using the IRIS GL screen coordinate system; if *glcoordsystem* is DM\_FALSE, the Movie Library assumes

that you are using the X coordinate system.

To determine the offset that the Movie Library will actually produce for a given offset, call **mvQueryViewOffset()**. Its function prototype is:

where:

offsetx is the requested offset from the origin in the x (horizontal)

dimension

offsety is the requested offset from the origin in the y (vertical)

dimension

offsetxreturn is a pointer into which the actual offset from the origin in the

*x* (horizontal) dimension is returned

offsetyreturn is a pointer into which the actual offset from the origin in the

y (vertical) dimension is returned

*glcoordsystem* specifies the screen coordinate system for the *y* offset:

DM\_TRUE for the IRIS GL screen coordinate system;

DM\_FALSE for the X coordinate system.

To get the requested offset value, call **mvGetViewOffset()**. Its function prototype is:

where:

offsetxreturn is a pointer into which the requested offset from the origin

in the *x* (horizontal) dimension is returned

offsetyreturn is a pointer into which the requested offset from the origin

in the *y* (vertical) dimension is returned

*glcoordsystem* specifies the screen coordinate system for the *y* offset:

DM\_TRUE for the IRIS GL screen coordinate system;

DM\_FALSE for the X coordinate system.

mvGetViewOffset() always returns the last requested origin of the movie frame, regardless of the movie's actual origin. Thus, you can inspect both the requested origin by calling mvGetViewOffset(), and the actual origin by calling mvQueryViewOffset().

# Binding a Movie to a Window for Playback

In order to play a movie, you need to bind it to a window. Binding a movie to a window performs internal initialization that prepares the movie for playback in that window; therefore, you must bind the movie to the window before calling any playback routines on that movie.

To bind a movie to an IRIS GL window, call **mvBindWindow()**. To bind a movie to an OpenGL window, call **mvBindOpenGLWindow()**.

The function prototypes for **mvBindWindow()** and **mvBindOpenGLWindow()** are:

```
DMstatus mvBindWindow(MVid movieid, Display* dpy, Window win)

DMstatus mvBindOpenGLWindow(MVid movieid, Display* dpy,

Window win, GLXContext ctxt)
```

#### where:

```
dpy is the display on which the movie playswin is the window in which the movie playsctxt is the OpenGL graphics context for the movie
```

For example, to bind the movie named *theMovie* to the GL/X window identified by *win* on the display *dpy*, using IRIS GL:

```
if ( mvBindWindow( theMovie, *dpy, win ) != DM_SUCCESS ) {
   fprintf( stderr, "%s: Could not bind movie to window.\n",
        programName );
}
```

You can bind a movie to only one window at a time. The movie remains bound to that window until you call **mvClose()** to close the movie. You can call **mvBindWindow()** or **mvBindOpenGLWindow()** only once per movie.

# Binding a Window to a Movie with an Audio Track

If the specified movie has an audio track, **mvBindWindow()** or **mvBindOpenGLWindow()** attempts to open an audio port by calling **ALopenport()** from the Audio Library. If the attempt is successful, an audio port with the name **movi***pid:movieid* is created:

#### where:

pid is the system process ID that is using the audio portmovieid is the movie ID containing the soundtrack that is being played on the port

See the *ALopenport*(3A) man page for more information.

If it is not possible to open an audio port, either because there are no free audio ports or because your system does not have audio capability, the Movie Library plays the image track without playing the accompanying audio track.

**Note:** Audio is not part of the X protocol, so the audio track will not follow the DISPLAY variable if the movie is played remotely.

# **Playing Multiple Movies in the Same Window**

To play more than one movie in the same window, bind the different movies to the same window by calling **mvBindWindow()** or **mvBindOpenGLWindow()** with the same *dpy* and *win* values for each *movieid*.

Example 30-2 is a code fragment that binds several movies to a window, and then shifts the position of each movie so they do not overlap.

**Example 30-2** Binding a Window for Playing Multiple Movies

```
static DMstatus bindWinToMovies( MVid* movieList, int numMovies, Display* dpy, Window win )
   int i;
   MVid currentMovie;
   for ( i = 0; i < numMovies; i++ ) {
       currentMovie = movieList[ i ];
         * Bind the GL window to the movie. This will cause several
         * movies to play in the same window.
        if ( mvBindWindow( currentMovie, dpy, win ) != DM_SUCCESS ) {
            fprintf( stderr, "%s: Could not bind movie to window.\n",
                    getProgramName() );
            return DM_FAILURE;
    }
    * Initially, all the movies appear at the same display location.
    * We call a helper function to move them so they do not overlap.
    * The helper function calls mvSetViewOffset(3mv) to accomplish
    * this task.
    * /
   if ( setMovieViewOffsets() != DM_SUCCESS ) {
       return DM_FAILURE;
    }
   return DM SUCCESS;
}
```

# **Controlling Movie Playback**

The Movie Library provides calls for controlling movie playback that let you start and stop playback, control audio playback, control the loop mode and loop limit, and scrub to a random frame.

# Starting and Stopping Playback

To set a movie into motion, call **mvPlay()**. Its function prototype is:

```
void mvPlay ( MVid movieid )
```

To halt playback, call **mvStop()**. Its function prototype is:

```
void mvStop ( MVid movieid )
```

Playing or stopping a movie does not affect any of the other settings such as playback speed or direction; for example, if you stop a movie that is playing backward, then start it again, the movie will continue to play backward.

# **Controlling Audio Playback**

If a movie has an audio track, the Movie Library plays the audio for you automatically if it is able to allocate an audio port.

#### **Enabling and Muting Audio**

You can control audio muting during playback. Muting controls only whether audio is played; it does not alter the audio volume. To control audio muting, call **mvSetEnableAudio()**. Its function prototype is:

```
void mvSetEnableAudio ( MVid movieid, DMboolean onoff )
where:
onoff controls audio muting: DM_TRUE enables audio,
```

DM\_FALSE mutes audio

To retrieve the current mute setting, call **mvGetEnableAudio()**. Its function prototype is:

```
DMboolean mvGetEnableAudio ( MVid movieid )
```

Example 30-3 is an excerpt from *manymovieEvents.c*, in /usr/people/4Dgifts/ examples/dmedia/movie/manymovie, that shows how to use audio muting. See "Playing Multiple Movies" in Chapter 32 for a complete description of manymovie.

### **Example 30-3** Enabling and Muting Audio Playback

```
static void toggleAudioMuting( MVid *theMovies , int
numMoviesInWin )
{
   int i;

   for( i = 0;i < numMoviesInWin; i++ ) {

      printf("%s: %d, toggle mute to ", getProgramName(),
theMovies[i]);

   if ( mvGetEnableAudio( theMovies[i] ) ) {
      printf( "OFF\n" );
      mvSetEnableAudio( theMovies[i], DM_FALSE );
   } else {
      printf( "ON\n" );
      mvSetEnableAudio( theMovies[i], DM_TRUE );
   }
}</pre>
```

### Guaranteeing the Audio Sample Rate When Playing Multiple Movies

If you play several movies at the same time, it is possible that not all of them will have the same audio sample rate. The audio hardware uses the same sample rate for all concurrent audio processes; therefore, if you play two (or more) movies that have different audio sample rates at the same time, audio plays at the wrong speed for movies whose sample rate is different than the current audio hardware rate setting. This happens only when playing more than one movie at a time; the audio sample rate is set to the proper value when playing a movie by itself.

To guarantee the proper playback rate for a certain movie when it is played in conjunction with other movies, call **mvSetPrimaryAudio()**. Its function prototype is:

```
void mvSetPrimaryAudio ( MVid movieid )
```

To determine which movie has the primary audio rate control, call **mvGetPrimaryAudio()**. Its function prototype is:

```
MVid mvGetPrimaryAudio ( void )
```

If you are writing an application that allows users to select and play one movie at a time from among several movies, you want the audio playback to be correct for each movie, even though not all of the movies have the same sample rate. In that case, call **mvSetPrimaryAudio()** on the selected movie before calling **mvPlay()**.

Example 30-4 is an excerpt from *manymovieEvents.c*, in /usr/people/4Dgifts/ examples/dmedia/movie/manymovie, which guarantees the proper audio rate for the first movie that the user enters on the command line. See "Playing Multiple Movies" in Chapter 32 for a complete description of *manymovie*.

### **Example 30-4** Designating a Movie as the Primary Audio Rate Controller

```
static void playTheMovies( MVid *theMovies, int numMoviesInWin )
{
   int i;
   mvSetPrimaryAudio( theMovies[0] );
   for( i = 0;i < numMoviesInWin; i++ ) {
      printf("%s: %d, play\n", getProgramName(), theMovies[i]);
      mvPlay( theMovies[i] );
   }
}</pre>
```

# Looping

Looping is the process of repeatedly playing movie frames. You can define the starting and ending frame for movie playback, thereby allowing looping on an entire movie or only a fragment of the movie. You can define only one such loop per movie. The playback loop mode, which is independent of the default loop mode that is stored in the movie, controls the playback behavior:

None The movie (or fragment) plays once through and then stops.

Continuous The movie (or fragment) plays repeatedly.

Swinging The movie (or fragment) plays repeatedly back and forth

from start to end in a forward direction, then from end to

start in a backward direction.

If the movie was playing backward when looping began, the movie will continue to loop backward.

### **Setting and Getting the Playback Loop Mode**

To set the playback loop mode, call **mvSetPlayLoopMode()**. Its function prototype is:

```
void mvSetPlayLoopMode ( MVid movieid, MVloopmode
newloopmode )
```

### where:

*newloopmode* is the loop mode:

MV\_LOOP\_NONE to play once through (default) MV\_LOOP\_CONTINUOUSLY to play continuously MV\_LOOP\_SWINGING to swing back and forth

You can change the loop mode whether or not the movie is playing. If you change the loop mode from swinging to continuous while the movie is on a backward swing, the movie will play backward continuously.

The Silicon Graphics movie file format lets you store a default loop mode setting within the movie file, which is independent of the playback loop mode setting. When opening a movie file, the Movie Library obtains the default loop mode from the movie if it is set; otherwise, it assumes MV\_LOOP\_NONE is the default loop mode. So, when you first play the movie, it uses the default loop mode until you change the playback loop mode.

To change the default loop mode stored in a movie, call **mvSetLoopMode()**, as described in "Setting and Getting the Default Movie Loop Mode" on page 594.

To retrieve the current loop mode setting for a specified movie, call **mvGetPlayLoopMode()**. Its function prototype is:

```
MVloopmode mvGetPlayLoopMode ( MVid movieid )
```

Example 30-5 is an excerpt from *manymovieEvents.c*, in /usr/people/4Dgifts/ examples/dmedia/movie/manymovie, which is described in "Playing Multiple Movies" in Chapter 32, that toggles the loop mode for a movie that is playing.

### **Example 30-5** Setting and Getting the Loop Mode

```
static void stepToNextLoopState ( MVid *theMovies,
                                   int numMoviesInWin )
    int i;
   MVloopmode loopMode;
    for( i = 0;i < numMoviesInWin; i++ ) {</pre>
       printf("%s: %d, change loop state to ",
               getProgramName(), theMovies[i]);
        switch ( mvGetPlayLoopMode( theMovies[i] ) ) {
        case MV_LOOP_NONE:
            loopMode = MV_LOOP_CONTINUOUSLY;
            printf( "CONTINUOUS\n" );
            break;
        case MV_LOOP_CONTINUOUSLY:
           loopMode = MV_LOOP_SWINGING;
            printf( "SWINGING\n" );
        case MV_LOOP_SWINGING:
            loopMode = MV_LOOP_NONE;
            printf( "NONE\n" );
            break;
       mvSetPlayLoopMode( theMovies[i], loopMode );
}
```

### **Counting and Limiting the Number of Loop Plays**

The Movie Library provides a facility for counting the number of times a movie has played. You can use this facility to query how many times the movie has played and to limit the number of times playback can loop.

Initialize the counter by calling **mvSetPlayLoopCount()**, and provide an upper bound for the number of loops by calling **mvSetPlayLoopLimit()** if you want to limit the number of times the movie is played.

You can query for the current loop count and reset it if desired. For example, if your application lets the user stop and restart playback, you can either reset the loop count when playback resumes or continue counting from the number of loops that were already completed when the movie was stopped. In swinging mode, each play of the movie forward or backward counts as one loop.

To set the loop count, call **mvSetPlayLoopCount()**. Its function prototype is:

```
void mvSetPlayLoopCount( MVid movieid, MVframe newloopcount )
```

#### where:

*newloopcount* is the value to which you want to set the loop count

To retrieve the current loop count for a movie, call **mvGetPlayLoopCount()**. Its function prototype is:

```
MVframe mvGetPlayLoopCount ( MVid movieid )
```

To limit the number of times a movie can play in continuous or swinging mode, call **mvSetPlayLoopLimit()**. Its function prototype is:

```
void mvSetPlayLoopLimit ( MVid movieid, MVframe newlooplimit)
```

#### where:

newlooplimit

is the number of times to loop or swing. This value must be either an integer or MV\_LIMIT\_FOREVER to keep playing

a movie forever.

To retrieve the current loop limit value, call **mvGetPlayLoopLimit()**. Its function prototype is:

```
MVframe mvGetPlayLoopLimit ( MVid movieid )
```

# **Playing or Looping a Movie Fragment**

You can play or loop-play a movie fragment by selecting a start frame and an end frame within the movie. You can define only one such fragment per movie and the fragment must contain at least one frame.

The start frame number must be less than or equal to the end frame number; frame 0 (the first frame of the movie) is the default start frame. If the current frame is outside the range specified, the Movie Library treats the movie as having reached end-of-media.

To define the start frame for a movie (or fragment), call **mvSetStartFrame()**. Its function prototype is:

```
void mvSetStartFrame ( MVid movieid, MVframe startframe )
where:
```

To define the end frame of a movie (or fragment), call **mvSetEndFrame()**. Its function prototype is:

is the frame number where playback begins

```
void mvSetEndFrame ( MVid movieid, MVframe endframe )
where:
endframe is the frame number where playback ends
```

To get the frame number where playback begins, call **mvGetStartFrame()**. Its function prototype is:

```
MVframe mvGetStartFrame ( MVid movieid )
```

startframe

To get the frame number where playback ends, call **mvGetEndFrame()**. Its function prototype is:

```
MVframe mvGetEndFrame ( MVid movieid )
```

# Scrubbing to a Random Frame During Playback

This section explains how to use the Movie Library to respond to an interface such as a scroll bar that lets the user jump to a random location in a movie while it is playing.

The Movie Library has two routines for jumping to a particular frame in a movie: mvSetCurrentFrame() and mvScrubCurrentFrame(). Each routine is designed with a specific use in mind. mvScrubCurrentFrame() provides the quickest response by rapidly locating the nearest frame (keyframe for keyframed formats) to the selected frame; mvSetCurrentFrame() provides the most accurate response, by precisely locating and then showing the frame exactly as selected.

### Their function prototypes are:

```
void mvScrubCurrentFrame( MVid movieid, MVframe newframe )
void mvSetCurrentFrame( MVid movieid, MVframe newframe )
```

where:

*newframe* is the frame to which you want to jump

One way to use these routines is to call **mvScrubCurrentFrame()** to make coarse jumps while the user is dragging the scroll bar, then call **mvSetCurrentFrame()** to go to and display the exact frame when the user releases the mouse button. The triangular pointer in the Movie Player uses this technique.

To retrieve the location of the current frame, call **mvGetCurrentFrame()**. Its function prototype is:

```
MVframe mvGetCurrentFrame ( MVid movieid )
```

# Synchronizing Movie Playback

Movies play in real time, slaved to a time base that is expressed in frames per second (FPS). The time base is either synchronized to the audio track or to a software-based timer that you set with **mvSetImageRate()**.

# **Getting and Setting the Playback Speed**

You can specify a multiplier for modifying a movie's natural frame rate. The natural frame rate is stored in the movie file and can be retrieved with **mvGetImageRate()**. The default playback speed is 1.0 times the natural frame rate.

To alter the normal playing speed, call **mvSetPlaySpeed()**, which expresses the new playback speed as a multiple of the movie's natural image frame rate. Its function prototype is:

```
void mvSetPlaySpeed ( MVid movieid, double newplayspeed )
```

#### where:

newplayspeed is an integer (for speeding up playback) or the reciprocal of an integer (for slowing down playback)

For example, if you specify a *newplayspeed* of 2.0, the movie plays twice as fast as it normally would. If you specify a negative value for *newplayspeed*, the movie plays in reverse.

You can measure the resulting frame rate by calling **mvGetActualFrameRate()**, as described in "Measuring the Current Frame Rate" on page 660.

For movies with audio, the Movie Library attempts to resample the audio to match the requested playback speed. This changes the pitch of the audio. The current implementation uses software techniques to perform audio resampling, so you should be aware that changing the speed of playback for a movie with an audio soundtrack consumes more processor time.

mvSetPlaySpeed() has no effect if the movie has been set to play every frame with mvSetPlayEveryFrame(). To determine the current setting for playback speed, call **mvGetPlaySpeed()**. Its function prototype is:

```
double mvGetPlaySpeed ( MVid movieid )
```

## Measuring the Current Frame Rate

You can obtain the actual playback frame rate for a movie as it is playing; the Movie Library computes the frame rate over the last second of playback. The current frame rate is the movie's natural frame rate, in frames per second (FPS) modified by the speed factor set by **mvSetPlaySpeed()**.

To determine the current frame rate, call **mvGetActualFrameRate()**. Its function prototype is:

```
DMstatus mvGetActualFrameRate ( MVid movieid, double* ratereturn )

where:

ratereturn is a pointer into which the actual frame rate is returned
```

If your application needs more detailed information about the rate of movie playback, you should check the timestamp information associated with movie frame events as described in "Handling Events" on page 668.

# Setting and Getting a Minimum Playback Speed Threshold

If the playback rate lags behind the time base, the Movie Library will drop (not display) image frames to stay in sync.

You can set a threshold value for the slowest acceptable playback speed and have the Movie Library notify you through the movie event queue if the playback speed falls below this minimum accepted speed by requesting MV\_SLOW\_PLAY events using **mvSetSelectEvents()**.

The threshold value must be a number between 0 and 1, which is a fraction of the movie's current speed. The current speed of a movie is its natural frame rate, as specified in the movie file, multiplied by the speed factor set by **mvSetPlaySpeed()**. The threshold setting is global—it applies to all

movies in your application. The default threshold is .5, meaning 50% of the current speed.

To set a minimum playback rate threshold, call **mvSetSlowThreshold()**. Its function prototype is:

```
void mvSetSlowThreshold ( double slowthresh )
```

#### where:

slowthresh is the threshold value for minimum playback speed

To retrieve the current minimum playback rate threshold, call **mvGetSlowThreshold()**. Its function prototype is:

```
double mvGetSlowThreshold ( void )
```

See "Checking and Correcting for Slow Playback" on page 678 for an explanation of how to check for slow playback events and correct for them.

# Forcing Playback of Every Frame

Some applications need to display every frame in a movie; for example, visualization applications where the user is looking for data patterns or trends must show every frame to ensure accuracy. The audio track is ignored (the movie plays silently) when you play every frame. In addition, the slow playback threshold is ignored when you play every frame, and you won't receive MV\_SLOW\_PLAYBACK events either.

To force the Movie Library to show every frame in sequence as fast as it can, call **mvSetPlayEveryFrame()**, passing in a value of DM\_TRUE. Its function prototype is:

```
void mvSetPlayEveryFrame( MVid movieid, DMboolean sync )
```

#### where:

sync

determines whether the Movie Library should play every frame or drop frames to stay in pace with the movie's time base: DM\_TRUE plays every frame; DM\_FALSE (default) drops frames when it becomes necessary in order to maintain a desired frame rate

To find out if playback is set to play every frame, call **mvGetPlayEveryFrame()**. Its function prototype is:

DMboolean mvGetPlayEveryFrame( MVid movieid )

# **Integrating Movies with IRIS GL Graphics**

You can use the Movie Library with the IRIS Graphics Library to combine the display of movies and graphics. You can draw IRIS GL graphics in the same window where a movie is playing, or you can have a movie playing in one window while drawing graphics in another window on the same screen.

The Movie Library has two different methods for combining movies and graphics—which method you use depends on whether the graphics overlap any portion of the movie frame. In one method, the display of movie frames is automatically controlled for you, in the other method, you control the display of movie frames manually from within your application. Frame display is a global setting that applies to all currently open movies.

# **Controlling the Frame Display Automatically**

If the graphics you want to draw using the IRIS GL do not overlap any portion of the movie display, for example, if your application displays movies and graphics side-by-side in one window or in separate windows, you can use the **mvGrabIrisGL()** and **mvReleaseIrisGL()** routines. These routines essentially put movie rendering on hold while the application performs IRIS GL rendering. Their function prototypes are:

```
void mvGrabIrisGL ( void )
void mvReleaseIrisGL ( void )
```

Use **mvGrabIrisGL()** and **mvReleaseIrisGL()** to surround the block of code that performs the IRIS GL drawing. Call **mvGrabIrisGL()** at the beginning of a block of IRIS GL calls, then call **mvReleaseIrisGL()** when the IRIS GL drawing is completed.

Example 30-6 is an excerpt from *moviescreenGl.c*, in /usr/people/4Dgifts/ examples/dmedia/movie/moviescreen, that shows how to use **mvGrabIrisGL()** and **mvReleaseIrisGL()**. It uses the IRIS GL **bgnpolygon()** and

**endpolygon()** routines to draw a box that covers up the distracting visual effects caused by moving the movie around the screen. See "Creating a Movie Screensaver Application" in Chapter 32 for more details about the *moviescreen* program.

## **Example 30-6** Using mvGrabIrisGL() and mvReleaseIrisGL()

```
void undrawSaverPicture()
   int width;
   int height;
   MVid imageTrack;
   MVid theMovie = getMovieID();
    if ( isFullScreen() )
       return;
     * Determine current size of movie on display
   mvFindTrackByMedium( theMovie, DM_IMAGE, &imageTrack );
    width = mvGetImageWidth( imageTrack );
   height = mvGetImageHeight( imageTrack );
   width *= getZoom();
   height *= getZoom();
     * Draw two black boxes to erase trailing garbage as
     * movie dances around the display.
    mvGrabIrisGL();
   bgnpolygon();
        short vctr[2];
        if (dx > 0)
           vctr[0] = offsetx - 1;
           vctr[1] = offsety - 1;
           v2s( vctr );
           vctr[0] = offsetx +dx;
           vctr[1] = offsety - 1;
           v2s( vctr );
```

```
vctr[0] = offsetx + dx;
   vctr[1] = offsety + height + 1;
   v2s( vctr );
   vctr[0] = offsetx - 1;
   vctr[1] = offsety + height + 1;
   v2s( vctr );
} else {
   vctr[0] = offsetx + width;
   vctr[1] = offsety - 1;
   v2s( vctr );
   vctr[0] = offsetx + width + dx + 1;
   vctr[1] = offsety - 1;
   v2s( vctr );
   vctr[0] = offsetx + width + dx + 1;
   vctr[1] = offsety + height + 1;
   v2s( vctr );
   vctr[0] = offsetx + width;
   vctr[1] = offsety + height + 1;
   v2s( vctr );
}
if (dy > 0)
   vctr[0] = offsetx - 1;
   vctr[1] = offsety - 1;
   v2s( vctr );
   vctr[0] = offsetx + width + 1;
   vctr[1] = offsety - 1;
   v2s( vctr );
   vctr[0] = offsetx + width + 1;
   vctr[1] = offsety + dy;
   v2s( vctr );
   vctr[0] = offsetx - 1;
   vctr[1] = offsety + dy;
   v2s( vctr );
```

```
} else {
            vctr[0] = offsetx - 1;
            vctr[1] = offsety + height;
            v2s( vctr );
            vctr[0] = offsetx + width + 1;
            vctr[1] = offsety + height;
            v2s( vctr );
            vctr[0] = offsetx + width + 1;
            vctr[1] = offsety + height + dy + 1;
            v2s( vctr );
            vctr[0] = offsetx - 1;
            vctr[1] = offsety + height + dy + 1;
            v2s( vctr );
    endpolygon();
    mvReleaseIrisGL();
}
```

## **Controlling the Frame Display Manually**

The disadvantage of the automatic **mvGrabIrisGL()** – **mvReleaseIrisGL()** method is that you can't really synchronize the graphics and movie displays, so some things might not display exactly when you want them to.

This section describes a method that lets you disable the Movie Library's automatic display so that your application can control the display of movie frames. When you have explicit control over the display of movie frames, you can tell the Movie Library exactly when to show a frame. This technique lets you draw IRIS GL graphics on top of the movie frame.

To have your application (rather than the Movie Library) control the display of movie frames, call **mvSetFrameDisplay(DM\_FALSE)**. Its function prototype is:

```
void mvSetFrameDisplay ( DMboolean showframes )
```

#### where:

showframes is either DM\_TRUE, to allow automatic frame display, or

DM\_FALSE, to disable automatic frame display for all open

movies

To retrieve the current frame display setting, call **mvGetFrameDisplay()**. Its function prototype is:

```
DMboolean mvGetFrameDisplay ( void )
```

When you disable the frame display, you have to tell the Movie Library when to display a frame by calling **mvShowCurrentFrame()** for every frame that you want to display. Its function prototype is:

```
void mvShowCurrentFrame ( MVid movieid )
```

To play a movie with manual frame display, you should include MV\_EVENT\_FRAME events in your event mask and call **mvShowCurrentFrame()** in response to those events.

Another way to use **mvShowCurrentFrame()** is to repaint a movie frame in response to an expose event from the window manager. See "Handling X Window Events" on page 676 for more details.

You must use one of the two methods described in this section if you want to draw with the IRIS GL while using the Movie Library. The IRIS GL and the Movie Library use separate threads of execution, and the routines discussed in this section supply internal synchronization between the processes. The results of any other method of combining graphics and movies are unpredictable. See Appendix E, "Using Graphics and Share Groups," in the *Graphics Library Programming Guide* for more information.

**Hint:** Advanced programmers might want to turn off the frame display in order to use a double-buffered RGB window. In this case, you have to create a window with the graphics configuration that you want, and you must call **swapbuffers()** from the GL and **mvShowCurrentFrame()** from the Movie Library yourself for every frame. This method is only for special cases; it is not the recommended way to combine graphics and movies.

Example 30-7 is an excerpt from *moviescreenEvents.c*, in /usr/people/4Dgifts/ examples/dmedia/movie/moviescreen, that uses **mvSetFrameDisplay()** to temporarily disable movie display while initialization routines are carried out.

## **Example 30-7** Initializing Movie Playback

```
static void initSaverMovie( )
   MVid
               newMovie;
   char
                *movieName;
   movieName = pickMovieAtRandom( );
    if ( mvOpenFile( movieName, O_RDONLY, &newMovie ) ==
                     DM_FAILURE ) {
        printf("%s: Could not open movie %s\n",
               getProgramName(),movieName );
        exit( EXIT_FAILURE );
    setMovieID( newMovie );
     * Disable movie display temporarily,
     * to avoid visual glitches.
   mvSetFrameDisplay( DM_FALSE );
    setupMovieWindow( newMovie );
    setPlaybackLoopmode( newMovie );
    setVolumeLevel( newMovie );
     * play the movie
   mvSetCurrentFrame( newMovie, 0 );
   mvSetFrameDisplay( DM_TRUE );
   mvPlay( newMovie );
```

# **Handling Events**

If you've used the **select()** or **poll()** system calls, or X event handling, the Movie Library event handling routines will be familiar to you. Even if your application has no user interface, it is recommended that you implement movie event handling because it is the best way to find out about system or I/O errors that occur while a movie is playing.

The Movie Library has an event queue, similar to the X event queue, that stores movie events such as playing a frame, stopping playback, and errors. One queue stores movie events for all currently open movies by storing the identifier of the movie instance responsible for an event as part of the event.

The Movie Library event queue is separate from the X event queue; so if your application handles X events as well as movie events, you'll probably want to create two separate functions, one for handling X events and one for handling movie events, and call them both from **main()**.

To add movie event handling to your application, follow these basic steps:

- 1. Choose which movie events you want to process by creating an event mask, as described in "Preparing an Event Mask" on page 669.
- 2. Get the file descriptor associated with the movie event queue, as described in "Getting a File Descriptor for the Movie Event Queue" on page 670.
- 3. Use one of the following methods to listen for movie and X11 events.
  - If you are using Xlib, prepare a file descriptor set that contains both the Movie Library and X11 file descriptors using the FD\_SET macro, and then code a **select()** loop that listens for movie events and X events, as described in "Preparing a File Descriptor Set" on page 672.
  - If you are using Xt or IRIS IM, pass the Movie Library file descriptor to the X toolkit using **XtAppAddInput**(3Xt). This call associates an Xt callback function that you have written for your application with the Movie Library file descriptor. You can then process movie events inside this Xt callback, which gets called whenever one or more events appear on the Movie Library event queue.
- 4. As long as there are events on the queue, keep processing them.

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

# **Preparing an Event Mask**

An event mask specifies which events the application is interested in processing. The event mask is a bitwise **or** of the events.

Table 30-1 lists and describes the Movie Library events.

**Table 30-1** Movie Library Events

Event	Description
MV_EVENT_FRAME	A frame has been played.
MV_EVENT_STOP	The movie has stopped playing.
MV_EVENT_ERROR	An error has occurred during playback.
MV_EVENT_SLOW_PLAY	The last second of playback measured by the Movie Library was slower than the threshold set in mvSetSlowThreshold().

To set an event mask, call **mvSetSelectEvents()**. Its function prototype is:

```
void mvSetSelectEvents ( MVeventmask eventmask )
```

### where:

eventmask

is a bitwise OR of one or more of the events in Table 30-1

For example, to receive all events, use the following:

To retrieve the current setting of the event mask, call **mvGetSelectEvents()**. Its function prototype is:

```
MVeventmask mvGetSelectEvents ( void )
```

Movie events are contained in an MVevent structure, which is a union of all of the Movie Library event structures. Each event structure contains fields

that store information about the event. Some events have additional fields containing event-specific information.

Table 30-2 lists and describes the fields in the movie events.

**Table 30-2** Event Structure Fields

Туре	Field	Description
MVeventtype	type	Type of event.
MVtime	time	X11-style millisecond timestamp indicating when the event happened.
MVid	id	Movie instance that produced the event.
MVframe	frame	Current frame number, ranging from zero to one less frame than the length of the movie's image track.
int	errcode	Integer value that represents a Movie Library error code that can be retrieved by calling mvGetErrno(). Applies to MV_EVENT_ERROR events only.
int	reason	Integer value that indicates what caused the slow play event. Applies to MV_SLOW_PLAYBACK events only and is currently unused.

See "Handling Movie Events" on page 672 for an explanation of how to handle movie events, and see Example 30-9 on page 675 for a code fragment that branches depending on the event type and then extracts information from the event fields.

# **Getting a File Descriptor for the Movie Event Queue**

The Movie Library provides a file descriptor that becomes active when you can read events from the movie event queue. To get a file descriptor for the movie event queue, call **mvGetEventFD()**. Its function prototype is:

# **Creating the Event Loop**

After you have set up the event mask and obtained a file descriptor for the movie event queue, you can create the event loop. You'll probably want to define separate window and movie event handling functions and call them from main().

Use one of the following methods to listen for movie and X11 events.

- If you are using Xlib, prepare a file descriptor set that contains both the Movie Library and X11 file descriptors using the FD\_SET macro, and then code a **select()** loop that listens for movie events and X events, as described in "Preparing a File Descriptor Set" on page 672.
- If you are using Xt or IRIS IM, pass the Movie Library file descriptor to the X toolkit using **XtAppAddInput**(3Xt). This call associates an Xt callback function that you have written for your application with the Movie Library file descriptor. You can then process movie events inside this Xt callback, which gets called whenever one or more events appear on the Movie Library event queue.

## **Handling Movie Events**

If you are using Xt or IRIS IM, you'll use the Movie Library routines for handling movie events that are described in this section, but you'll get the events from the X11 queue, as described in "Handling X Window Events" on page 676, so skip ahead to "Waiting for Movie Events" on page 673 in this section to learn how to handle movie events.

If you are using Xlib, you need to create a loop that uses **select()** or **poll()** to wait for Movie Library events using a file descriptor set (*fd\_set*). The basic structure of this loop is:

### **Preparing a File Descriptor Set**

File descriptor sets provide a way for an application to wait for available input from several files at once. The Movie Library event queue is separate from the X11 event queue, so you must obtain the file descriptor for each queue and put them both into a file descriptor set.

To get the file descriptor corresponding to the X11 event queue for a specified display, use the Xlib **ConnectionNumber**(3X) macro.

After you have obtained file descriptors for the movie event queue and the X11 event queue, put them into a file descriptor set using the FD\_SET macro.

Example 30-8 shows how to prepare a file descriptor set.

### **Example 30-8** Preparing a File Descriptor Set

The structure of this code outline and the reasoning behind it are explained in the sections that follow.

### **Waiting for Movie Events**

When creating a movie event loop, it is important that you process all the events on the queue to prevent it from stalling.

The Movie Library provides the **mvPendingEvents()** call to check the queue for events, with which you can create a **while** loop. Its function prototype is:

```
int mvPendingEvents ( void )
```

You must use the **while** statement to set up the loop properly; don't use an **if** statement.

When your application wakes up to handle movie events, there may be more than one event in the queue. It is critical that you process all the events in the queue before waiting for more events. As long as events are pending, your application should keep getting events from the queue until it is empty.

#### **Getting Movie Events from the Queue**

To extract the next event from the queue, call **mvNextEvent()**. Its function prototype is:

```
void mvNextEvent ( MVevent* eventreturn )
where:
eventreturn is a pointer into which the next event is returned
```

You can determine which event is waiting next in the queue, without popping it off the queue, by "peeking" at the next event. To peek at the next event, call mvPeekEvent(). Its function prototype is:

```
void mvPeekEvent ( MVevent* eventreturn )
where:
```

eventreturn

is a pointer into which the next event is returned without

being removed from the queue

### **Examining Movie Events**

You can query the fields in the event structure to get specific information about an event, such as the event type. After you know what type of event it is, you can use the structure definitions for the specific event types to extract further information, such as when or why the event occurred.

The first field of every event is a *type* field, which lets you query for the event type without knowing it in advance. After determining the event type, you can examine the fields for each type of event to extract specific information about the event. The typical way to code this is with a **switch** statement.

Example 30-9 is an excerpt from <code>manymovieEvents.c</code>, in <code>/usr/people/4Dgifts/examples/dmedia/movie/manymovie</code>, that uses a movie event loop in which a <code>switch</code> statement is used to examine the event type, then extracts event-specific information such as the frame number that caused the event <code>(event.mvframe.id)</code>, from the fields in the event structure. (The print statement under MV\_EVENT\_FRAME is commented out in the sample program.)

**Example 30-9** Handling Movie Frame, Stop, and Error Events

```
static DMboolean handleMovieEvents( )
   MVevent event;
   while ( mvPendingEvents() != 0 ) {
        mvNextEvent( &event );
        switch ( event.type ) {
            case MV_EVENT_FRAME:
                                                /* a frame played */
                /* Uncomment and recompile to see which frames are played.
                 * printf( "%s: Played frame %d of movie %d.\n",
                          getProgramName(), event.mvframe.frame,
                          event.mvframe.id );
                break;
            case MV_EVENT_STOP:
                                                /* end of movie */
                printf( "%s: Playback of movie %d stopped.\n",
                       getProgramName(), event.mvstop.id );
                break;
                                                 /* error */
            case MV_EVENT_ERROR:
                fprintf( stderr, "%s: Error during playback: %s.\n",
                        getProgramName(), mvGetErrorStr( mvGetErrno() ) );
                return DM_FALSE;
                break;
   }
   return DM_TRUE;
```

## Handling X Window Events

The method for handling the X events depends on whether you use Xlib or an X toolkit:

- If you are using Xt or IRIS IM, you need to write Xt callback functions
  for the GLXMDraw widget that get called when it is resized and when
  it needs repainting. The X toolkit invokes the Xt callback procedures in
  your application whenever one or more events appear on the X11 event
  queue. Call XtAppAddInput(3Xt) to add the Movie Library file
  descriptor to the X11 file descriptors that invoke your Xt callback
  function.
- If you are using Xlib, you need to get the file descriptor corresponding to the X11 event queue for a specified display, by calling the Xlib ConnectionNumber(3X) macro. Put this file descriptor into a file descriptor set along with the Movie Library file descriptor and wait on both of them using select(), as described in "Handling Movie Events" on page 672.

The two window events of greatest concern to Movie Library programmers are expose and resize events.

If a window containing a movie gets exposed, you must repaint the window; if it gets resized, you must resize and repaint the window.

To resize the window identified by *win* on the display *dpy*, call **mvResizeWindow()**. Its function prototype is:

```
void mvResizeWindow ( Display* dpy, Window win )
```

To repaint a window, call **mvShowCurrentFrame()**. Its function prototype is:

```
void mvShowCurrentFrame ( MVid movieid )
```

Example 30-10 is an excerpt from *simplemovie.c*, in /usr/people/4Dgifts/ examples/dmedia/movie/misc, that shows how to handle window events from the X11 queue, in particular, expose and resize events. See "Creating a Simple Keyboard Interface for Playing Movies" in Chapter 32 for more information about *simplemovie.c*.

**Example 30-10** Handling X11 Expose and Resize Window Events

```
static DMboolean handleXEvents( MVid theMovie, Display *dpy, Window win )
   XEvent event;
   while ( XPending( dpy ) != 0 ) {
        XNextEvent( dpy, &event );
        switch ( event.type ) {
                                              /* repaint display */
            case Expose:
                mvShowCurrentFrame( theMovie );
                break;
            case ConfigureNotify:
                                              /* window was resized */
                {
                    XWindowAttributes winAttrs;
                    int actual_width, actual_height;
                    mvResizeWindow( dpy, win );
                    XGetWindowAttributes( dpy, win, &winAttrs );
                    mvQueryViewSize( theMovie, winAttrs.width,
                                     winAttrs.height, DM_TRUE,
                                     &actual_width, &actual_height );
                    printf( "%s: actual width = %d, height = %d\n",
                           programName, actual_width, actual_height );
                    mvSetViewSize( theMovie, winAttrs.width,
                                  winAttrs.height, DM_TRUE );
                break;
return DM_TRUE;
```

# **Checking and Correcting for Slow Playback**

The Movie Library checks the playback speed once per second and sends a slow play event to the movie event queue if it finds that the last second of playback was too slow.

To check for slow playback, you must:

- set a minimum playback speed threshold
- set an event mask to request MV\_SLOW\_PLAYBACK events

To check playback speed more frequently, you can choose to receive frame events and compute the difference between the two timestamps, which is the real time in milliseconds that has elapsed between the two events.

Playback may be slow because:

- the I/O bandwidth from a hardware device is not high enough
- the filesystem is fragmented
- the system CPU cannot decompress data fast enough
- the movie is unoptimized (see *mvOptimize*(3mv))

# Chapter 31

# Using the Movie Library with QuickTime Movies

This chapter explains how to use the Movie Library to create, edit, and play QuickTime movies. It also describes the QuickTime Compressor LIbrary, which gives Movie Library applications access to QuickTime compression schemes.

# Using the Movie Library with QuickTime Movies

This chapter describes how to create, edit, and play uncompressed QuickTime movies on a Silicon Graphics computer using the Movie Library. Silicon Graphics also provides a separate option, the QuickTime Compressor Library, under license from Apple Computer, Inc. for use with the Movie Library, that supports Apple Animation and Apple Video compression for QuickTime movies.

**Note:** To use the Apple Animation and Apple Video compression discussed in this chapter, you must purchase and install the Silicon Graphics QuickTime Compressor Library.

**Note:** Apple QuickTime software for the Macintosh that is mentioned in this chapter is available from Apple Computer, Inc. in the Apple QuickTime Starter Kit.

#### **QuickTime Basics**

This section presents basic concepts for using QuickTime movies with the Movie Library.

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 Sound

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.

When playing a QuickTime movie soundtrack on a Silicon Graphics computer, the Movie Library chooses the nearest appropriate audio sample rate during playback and file format conversion of nonnative movies, so no performance penalty is incurred for rate conversion. This may change the pitch of some sounds slightly, but usually not enough to cause an audible difference. The Movie Library supports 8-bit and 16-bit uncompressed audio in either two's complement or unsigned format. The native rates for Silicon Graphics audio hardware are 8000, 11025, 16000, 22050, 32000, 44100, and 48000 Hz.

When playing a QuickTime movie soundtrack on a Macintosh, the Macintosh QuickTime software converts nonnative audio to the native Macintosh 8-bit format and performs the necessary rate conversions for playing the nonnative soundtrack; but if you use the Movie Library to write audio data to a QuickTime movie that is intended for playback only on the Macintosh, use a format and sampling rate suitable for the Macintosh. QuickTime supports the AIFF sound file format and both signed and unsigned audio, at any rate up to 65 KHz. The native format for Macintosh audio hardware is 8-bit, unsigned audio at either 11127 Hz or 22254 Hz.

# **QuickTime Compression**

The Macintosh QuickTime system software extension includes five compressors, two of which are compatible with the Movie Library:

- Apple None (uncompressed)
- Apple Photo (JPEG standard)

and two of which are compatible with the Movie Library only when you purchase and install the Silicon Graphics QuickTime Compressor Library:

- Apple Animation
- Apple Video

You can use a QuickTime movie saved with either Apple Animation or Apple Video compression with the Movie Library if you have purchased and installed the Silicon Graphics QuickTime Compressor Library; otherwise, you must save the movie without compression or use JPEG.

When creating QuickTime movies on a Macintosh for use with the Movie Library, you can select the compression settings using one of the following methods:

- 1. Use the Macintosh QuickTime Movie Recorder, which gives you two options for compressing a movie:
  - accepting the default setting, "Use Simple Compression," which automatically applies Apple Video compression when you save the movie
  - selecting an appropriate type of compression from the Compression Settings dialog box and then choosing "Compress" from the Movie menu before saving the movie
- 2. Use the Macintosh QuickTime Movie Converter to change compression settings for existing movies

Compression settings and the applications for which each setting is best suited are summarized in the sections that follow.

#### **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. The Movie Library supports the JPEG standard through the Compression Library. 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. JPEG works better on high-resolution continuous-tone images, such as photographs, than on crisp-edged, high-contrast images like line drawings.

#### **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 developed by Apple Computer 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:** The Apple Video compressor has 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.

#### QuickTime Frame Differencing (Keyframes)

QuickTime provides a feature called *frame differencing*, which allows you to save only the changes from one movie frame to the next, rather than store each individual frame for the entire movie. When frame differencing is used,

the entire contents of every *n*th frame are compressed and stored in a *keyframe*, while only the differences from frame to frame are stored for the frames between keyframes. (On the Macintosh, you enable frame differencing by clicking on the "Key frame every *n* frames" box in the Motion area of the Compression Settings dialog box.)

Reducing the time between keyframes increases the performance when scrubbing to a particular frame (because of the increased likelihood of landing on a keyframe) but increases the size of the movie.

You can open and play keyframe movies with the Movie Library. You can't use the mvPasteFrames() editing function to paste frames into the middle of a keyframe movie because the Movie Library does not support the recomputing of frame differencing. For the same reason, you cannot delete frames from a keyframe movie. To create a movie using Apple Animation or Apple Video, you must use mvInsertFrames() to add the frames in order from beginning to end.

# Movie Library QuickTime Compatibility Requirements

QuickTime movies must be single-fork and self-contained in order to be compatible between Apple Macintosh and Silicon Graphics computers. This section describes how to make QuickTime movies that meet these requirements.

#### Making a Single-fork Movie

Files on an Apple computer are double-fork, containing both a data fork and a resource fork. Only Apple computers use the double-fork file system; the file system on Silicon Graphics computers does not use double-fork files.

Macintosh computers convert a double-fork file into a single-fork file by writing the resource fork into the data fork. You must perform this conversion on the Macintosh for any movie files that you plan on using with the Movie Library before transferring them to the Silicon Graphics computer.

To make a single-fork movie on the Macintosh:

- 1. Click on the "Playable on non-Apple computers" box in the QuickTime Movie Converter Save dialog box (see the *QuickTime Starter Kit User's Guide* for details).
- Equivalently, if you are writing an application on a Macintosh using the Apple QuickTime developer kit, call the Movie Toolbox PutMovieIntoDataFork function in that application to write the movie into a single-fork file before using it in a Movie Library application (see Inside Macintosh: QuickTime for details).

#### Making a Self-contained Movie

Apple QuickTime movie utilities offer two ways of saving a QuickTime movie file on the Macintosh:

- Save normally
- Make movie self-contained

A QuickTime movie that has been saved on the Macintosh using the "Save normally" feature contains movie synchronization information, such as the location and sequence of frames in the movie, but does not itself contain the actual movie data. The movie data could have been stored in one or more separate files on the disk, on CD-ROM, or on a remote computer accessible from the network. Such a movie is said to have dependencies, because it depends on being able to locate and use other files that make up the movie. You cannot use this type of movie with the Movie Library.

Only self-contained (also called "flattened") QuickTime movie files are compatible with the Movie Library. Self-contained QuickTime movies have both the synchronization information and the movie data in one file.

You can use the Macintosh Movie Info window to determine whether an existing movie is self-contained. If the movie is self-contained, nothing will appear in the box under "Uses data from one other file." If anything does appear in this box, you need to convert the movie to a self-contained file.

You must convert any movie files that you plan on using with the Movie Library to self-contained movies on the Macintosh before transferring them to the Silicon Graphics computer, using one of the following methods:

- Click on "Make movie self-contained" from the Save dialog box that
  appears when you select "Save" or "Save as" from the Macintosh File
  menu of the QuickTime Movie Player or Movie Recorder. Because the
  Macintosh default is "Save normally," you must specify "Make movie
  self-contained" when saving any QuickTime movie that you plan on
  using with the Movie Library.
- 2. To convert an existing QuickTime movie, use the Macintosh QuickTime Movie Converter to open the movie, select "Make movie self-contained," and then save the movie.
- Equivalently, if you are writing a Macintosh application using the Apple QuickTime developer kit, call the Movie Toolbox FlattenMovieData function in that application to make the movie file self-contained before using it in a Movie Library application.

# **Transferring Files Between Macintosh and Silicon Graphics Computers**

You can transfer QuickTime movies between an Apple Macintosh computer and a Silicon Graphics computer by using a floppy disk or by using a network file transfer application.

#### **Transferring Files from Floppy Disk**

If you have an Indigo or an Indy with a floptical drive, you can read files from a floppy disk that was formatted on a Macintosh computer.

#### **Transferring Files Over a Network**

To transfer QuickTime movies between an Apple Macintosh computer and a Silicon Graphics computer over a network, use either *ftp* or the Apple Computer AppleShare network software. When using *ftp*, specify "binary" to ensure complete data transfer.

# Adding QuickTime Capability to Your Movie Library Application

Once you know how to program a Movie Library application, adding QuickTime capability is as easy as adding two lines to your code. Only three steps are needed to create an application that can use QuickTime movies (as long as they meet the compatibility criteria):

- 1. including *qt.h*, the QuickTime header file
- 2. calling **mvInitQuickTime()** to initialize the Movie Library for QuickTime movies
  - This step also installs the QuickTime codecs in the Movie Library if you have purchased the QuickTime Compressor Library.
- linking with the QuickTime Compressor Library by putting -lqt before lmovie on the link line

QuickTime versions of many of the Movie Library sample programs are provided in the *createmovieqt* and *miscqt* directories under /usr/people/4Dgifts/examples/dmedia/movie.

# Using the QuickTime Compressor Library

If you have purchased and installed the Silicon Graphics QuickTime Compressor Library, codecs for Apple Video and Apple Animation are installed into the Movie Library and are available for your application to use on either QuickTime or Silicon Graphics movies.

#### Creating a QuickTime Movie

You can use the Movie Library to create a QuickTime movie that is playable on both Silicon Graphics and Apple computers. Use the same steps as creating a Silicon Graphics movie, but specify QuickTime format, as in the *createmovieqt.c* sample program.

Example 31-1 highlights the changes that were made to the *createmovie* sample program to turn it into *createmovieqt*, which is an identical program that offers QuickTime compatibility. These excerpts are from *createmovieqtArgs.c++* in the /usr/people/4Dgifts/examples/dmedia/movie/ *createmovieqt* directory.

**Example 31-1** Creating QuickTime Movies with the Movie Library

```
#include "createmovieArgs.h"
                                 #include <movie.h>
Include QuickTime header file
                                 #include <qt.h>
                                 #include <audiofile.h>
                                 #include <getopt.h>
                                 #include <string.h>
                                 #include <assert.h>
                                 #include <il/ilImage.h>
                                 #include <il/ilGenericImgFile.h>
                                 typedef enum _compScheme
                                     unknownComp,
                                     none,
                                     mvc1,
                                     mvc2,
                                     rle,
                                      jpeg,
                                     rgb8,
                                     qtvideo,
Add QuickTime compressors -
                                     gtanim
                                 } compScheme;
Use QuickTime Video compressor — static compScheme
                                                      compressionScheme = qtvideo;
Use QuickTime movie format
                             — static MVfileformat movieFormat
                                                                          = MV_FORMAT_QT;
                                                   setMovieFormat( char *formatArg );
Add function to set movie format — static void
Add function to check
                               - static DMboolean badCompressionScheme( void );
compression compatibility
```

```
while(( ch = getopt( argc, argv, "f:c:l:r:s:p:o:" ) ) != -1)
Add movie format to arguments
                                          switch (ch) {
                                              case 'f':
                                                   setMovieFormat( optarg );
                                                   break;
                                      if ( badCompressionScheme() ) {
                                          fprintf( stderr, "%s: Compression %s"
Check whether selected compression
                                                    "unavailable for QuickTime.\n",
                                                     programName, getCompressionScheme() );
scheme is compatible with file format
                                          exit( EXIT_FAILURE );
                                      }
Initialize QuickTime Library
                                      mvInitQuickTime();
                                  static void usage( void )
                                     fprintf( stderr, "[-f format] [-s xsize,ysize]"
                                                       "[-o outMovie] \n" );
                                     fprintf( stderr, "[-p paramType,userParam,userParamVal]"
                                                       " file ...\n" );
Add QuickTime info to usage
                                     fprintf( stderr, "\n");
                                     fprintf( stderr, "\"compression\" = none, mvc1, mvc2,"
                                               " rle, jpeg, " );
                                     fprintf( stderr, "8rgb, qtvideo, or qtanim.\n");
                                     fprintf( stderr, "Default compression scheme qtvideo.\n" );
                                     fprintf( stderr, "\"format\" = sgi or qt,"
                                                " the format of the new movie.");
                                     fprintf( stderr, "The default is qt.\n" );
```

```
Add function to set movie format _____ static void setMovieFormat( char *formatArg )
                                     if ( ( strcmp( formatArg, "sgi" ) == 0 ) ||
                                         ( strcmp( formatArg, "SGI" ) == 0 ) ) {
                                         movieFormat = MV_FORMAT_SGI_3;
                                     else if ( ( strcmp( formatArg, "qt" ) == 0 ) ||
                                              (strcmp(formatArg, "QT") == 0))
                                         movieFormat = MV_FORMAT_QT;
                                     }
                                     else {
                                         fprintf( stderr, "%s: Unknown movie format %s.\n",
                                                 programName, formatArg );
                                         usage();
                                     }
                                 }
Add function to check
                                 static DMboolean badCompressionScheme( void )
compression compatibility
                                     if ( ( getMovieFormat() == MV_FORMAT_QT ) &&
                                         ( ( compressionScheme != none ) &&
                                         ( compressionScheme != jpeg ) &&
                                          ( compressionScheme != qtanim ) &&
                                          ( compressionScheme != qtvideo ) ) ) {
                                         return( DM_TRUE );
                                     return( DM_FALSE );
                                 }
Add function to set compression —
                             — static void setCompressionScheme( char *compressArg )
                                     if ( strcmp( compressArg, "none" ) == 0 ) {
                                         compressionScheme = none;
                                     else if ( strcmp( compressArg, "mvc1" ) == 0 ) {
                                         compressionScheme = mvc1;
                                     else if ( strcmp( compressArg, "mvc2" ) == 0 ) {
                                         compressionScheme = mvc2;
                                     else if ( strcmp( compressArg, "rle" ) == 0 ) {
                                         compressionScheme = rle;
                                     }
```

```
else if ( strcmp( compressArg, "jpeg" ) == 0 ) {
                                        compressionScheme = jpeg;
                                    else if ( strcmp( compressArg, "8rgb" ) == 0 ) {
                                        compressionScheme = rgb8;
                                    else if ( strcmp( compressArg, "qtvideo" ) == 0 ) {
                                        compressionScheme = qtvideo;
                                    else if ( strcmp( compressArg, "qtanim" ) == 0 ) {
                                        compressionScheme = qtanim;
                                    }
                                    else {
                                        fprintf( stderr, "%s: Unknown compress scheme %s.\n",
                                                programName, compressArg );
                                        usage();
                                    }
                                }
Add function to get compression ——
                                schar *getCompressionScheme( void )
                                    switch( compressionScheme ) {
                                        case none:
                                            return( DM_IMAGE_UNCOMPRESSED );
                                        case mvc1:
                                            return( DM_IMAGE_MVC1 );
                                        case mvc2:
                                            return( DM_IMAGE_MVC2 );
                                        case jpeg:
                                            return( DM_IMAGE_JPEG );
                                        case rle:
                                            return( DM_IMAGE_RLE );
                                        case rgb8:
                                            return( DM_IMAGE_UNCOMPRESSED );
                                        case qtvideo:
                                            return( DM_IMAGE_QT_VIDEO );
                                        case qtanim:
                                            return( DM_IMAGE_QT_ANIM );
                                        case unknownComp:
                                            assert( DM_FALSE );
                                            break;
```

# Reading Existing QuickTime Movies

The Movie Library can read uncompressed QuickTime movies in 16- and 32-bit depths, which correspond to the "Thousands" and "Millions" of colors choices in the QuickTime compression settings on the Macintosh.

To open an uncompressed QuickTime movie, call mvOpenFile().

To read the data, call **mvReadFrames()**, which puts the image data into your buffer.

#### Converting QuickTime Image Data to Silicon Graphics Image Format

If you're working with an uncompressed QuickTime movie, you need to convert the QuickTime picture data to a Silicon Graphics image data format such as RGBX.

You can then invert it and transform it into RGBX, as shown in Example 31-2. Example 31-2 contains a listing of *createmovieConvert.c*, which is located in the /usr/people/4Dgifts/examples/dmedia/movie/createmovieqt directory.

**Example 31-2** Converting QuickTime Picture Data to RGBX Format

```
**************
* File:
             createmovieConvert.c
* Description: Part of createmovie. Used only in conjunction with the
             SGI QuickTime Library. Code for converting between
             QuickTime and SGI rgb image data.
#include <sys/types.h>
#include "createmovieConvert.h"
{}^{\star} Forward declarations for functions local to this module.
static void InvertImage32( void* buffer, int width, int height );
static void InvertImage16( void* buffer, int width, int height );
/*****
* Apple16ToRGBX
* 16-bit images are stored with 5 bits each of red, green, and blue.
* The bit layout is:
            XRRRRGG GGGBBBBB
* The bit layout for SGI 32-bit RGBX images is:
            XXXXXXX BBBBBBBB GGGGGGG RRRRRRR
* Apple stores images from top to bottom, while SGI goes from bottom to top.
*******/
```

```
void Apple16ToRGBX( int width, int height, void* from, void* to)
    static unsigned char Apple16Table[32];
    unsigned short* src = ( unsigned short* ) from;
                   dst = ( __uint32_t* )
    uint32 t*
                                               to;
    size t
                    size = ( ( size_t ) width ) * ( ( size_t ) height );
    size t
                    i;
    for (i = 0; i < 32; i++) {
        Apple16Table[i] = ((i << 3) | (i >> 2));
    }
    for ( i = 0; i < size; i++ ) {
        unsigned short bits = src[i];
        unsigned char red = Apple16Table[( ( 0x1F << 10 ) & bits ) >> 10];
        unsigned char green = Apple16Table[( ( 0x1F << 5 ) & bits ) >> 5];
        unsigned char blue = Apple16Table[( ( 0x1F << 0 ) & bits ) >> 0];
        dst[i] = ( blue << 16 ) | ( green << 8 ) | ( red << 0 );
    }
    InvertImage32( to, width, height );
/*****
 * RGBXToApple16
 *******/
void RGBXToApple16( int width, int height, void* from, void* to)
    __uint32_t*
                    src = ( \underline{uint32_t*} )
    unsigned short* dst = ( unsigned short* ) to;
    size_t
                    size = ( ( size_t ) width ) * ( ( size_t ) height );
    size_t
                    i;
    for ( i = 0; i < size; i++ ) {
        __uint32_t bits
                             = src[i];
        unsigned char red = ( 0x1F << 3 ) & bits ) >> 3;
        unsigned char green = ( ( 0x1F << 11 ) & bits ) >> 11; unsigned char blue = ( ( 0x1F << 19 ) & bits ) >> 19;
        dst[i] = ( blue << 0 ) | ( green << 5 ) | ( red << 10 );</pre>
    }
    InvertImage16( to, width, height );
```

```
/*****
 * Apple32ToRGBX
 * 32-bit images are stored with 8 bits each of red, green, and blue.
 * The bit layout is:
          XXXXXXX RRRRRRR GGGGGGG BBBBBBB
 * The bit layout for SGI 32-bit RGBX images is:
          XXXXXXX BBBBBBB GGGGGGG RRRRRRR
 * Apple stores images from top to bottom, while SGI goes from bottom to top.
 *******/
void Apple32ToRGBX( int width, int height, void* from, void* to)
   __uint32_t*
                 src = ( \__uint32_t* ) from;
   __uint32_t*
                   dst = ( __uint32_t* ) to;
   size_t
                   size = ( ( size_t ) width ) * ( ( size_t ) height );
   size_t
                   i;
   for ( i = 0; i < size; i++ ) {
       __uint32_t bits = src[i];
       dst[i] = ( ( bits & ( 0xFF << 16 ) ) >> 16 ) |
                ( ( bits & ( 0xFF << 8 ) ) >> 0 )
                ( ( bits & ( 0xFF << 0 ) ) << 16 );
   }
   InvertImage32( to, width, height );
}
```

```
/*****
 * RGBXToApple32
 * Apple packs the colors in a different order than SGI does:
void RGBXToApple32( int width, int height, void* from, void* to)
   __uint32_t*
                   src = ( \underline{uint32_t*}) from;
    __uint32_t*
                   dst = ( __uint32_t* ) to;
                   size = ( ( size_t ) width ) * ( ( size_t ) height );
   size_t
   size_t
   for ( i = 0; i < size; i++ ) {
        __uint32_t bits = src[i];
       dst[i] = ( (bits & (0xFF << 16)) >> 16) |
                 ( ( bits & ( 0xFF << 8 ) ) >> 0 ) |
                 ( ( bits & ( 0xFF << 0 ) ) << 16 );
   }
   InvertImage32( to, width, height );
/*****
 * InvertImage32
 * Inverts a 32-bit image.
 ******/
static void InvertImage32( void* buffer, int width, int height )
   __uint32_t* buff = ( __uint32_t* ) buffer;
   int x;
   int y1;
   for (x = 0; x < width; x++) {
       for (y1 = 0; y1 < height/2; y1++) {
           int y2 = height - y1 - 1;
           int index1 = x + y1 * width;
           int index2 = x + y2 * width;
            __uint32_t t = buff[index1];
           buff[index1] = buff[index2];
           buff[index2] = t;
   }
}
```

```
/*****
 * InvertImage16
* Inverts a 16-bit image.
*******/
static void InvertImage16( void* buffer, int width, int height )
   unsigned short* buff = ( unsigned short* ) buffer;
   int x;
   int y1;
   for ( x = 0; x < width; x++ ) {
       for ( y1 = 0; y1 < height/2; y1++ ) {
           int y2 = height - y1 - 1;
           int index1 = x + y1 * width;
           int index2 = x + y2 * width;
           unsigned short t = buff[index1];
                         = buff[index2];
           buff[index1]
           buff[index2]
                            = t;
       }
}
```

# Chapter 32

# Using the Movie Library Sample Programs

This chapter describes the Movie Library sample programs.

# Using the Movie Library Sample Programs

A comprehensive set of sample programs is provided with the Movie Library. You can use sample programs to learn about the Movie Library or as skeleton code for building your own application. This chapter describes the Movie Library sample programs.

# **About the Sample Programs**

Sample programs demonstrating how to use the Movie Library for creating, editing, and playing movies and displaying their parameters are included in the /usr/people/4Dgifts/examples/libmovie directory. Sample movies, somersault.mv and sampleQT.mv are also provided.

The sample programs are organized into directories according to purpose; most programs reside in a directory having the same name:

common contains source code to a helper function called *glxhelper*,

which creates an X window suitable for GL drawing—it uses no *libmovie* functions but is used by the *simplemovie*,

manymovie, and moviescreen sample programs

*createmovie* is a movie-making application with a command-line

interface

*createmovieqt* is a movie-making application with a command-line

interface for making QuickTime movies

editmovie is a movie-editing application with a command-line

interface for editing QuickTime movies

manymovie is a movie-playing application that allows several movies to

play at the same time

misc

contains four programs:

- aud-to-movie, which adds (or replaces) a movie audio track
- img-to-movie, which converts a sequenced image file into a movie
- mvinfo, which displays the parameters of a movie file
- *simplemovie*, which provides a keyboard interface for playing a movie

miscqt

contains two programs:

- *mvinfoqt*, which displays the parameters of a movie file
- simplemovieqt, which provides a keyboard interface for playing a QuickTime movie

The Movie Library sample programs have been put through an extensive code review and testing process by the engineers who created the Movie Library. The sample programs demonstrate how to write solid code with the Movie Library and provide many basic movie application features that you may want to incorporate into your own code.

Some of the guidelines followed in creating these sample programs:

- global functions are put in header files to increase modularity
- forward declarations are provided for local functions
- variables are lowercase
- symbols are uppercase
- functions begin with lowercase and have uppercase letters at word breaks
- extensive comments appear both at the beginning of each program or module and in the body of the code

Figure 32-1 shows the format of the introductory comments that appear in all the Movie Library sample programs. Introductory comments document the name of the file, its usage, a brief description of what the program does, and a complete listing of the functions used in the program.

```
File:
                                        createmovie.c++
                        Usage:
                                        createmovie [-c compression] [-l loopMode] [ -r frameRate
Complete usage
                                                     [-s xsize,ysize] [-o outMovie] file . . .
description. A usage
                                        file may include one or more image, audio, and movie file
message is displayed if
                                        Compression schemes: none mvcl rle jpeg 8rgb (default is
the user makes an error.
                                        Loop modes:
                                                               once loop swing
                                                                                         (default is
                                        Command-line program to make a movie file from image, aux
                        Description:
Description of program
                                        and/or other movie files.
                                        The following SGI Movie Library functions are used:
                         Functions:
                                        mvOpenFile()
                                        mvSetMovieDefaults()
                                        mvSetLoopMode()
                                        mvCreateFile()
                                        mvGetErrorStr()
Complete listing of the
                                        mvGetErrno()
Movie Library functions <
                                        mvClose()
used in this program
                                        mvGetParams()
                                        mvAddTrack()
                                        mvFindTrackByMedium()
                                        mvInsertFrames()
                                        mvGetAudioRate()
                                        mvGetImageRate()
                                        mvGetAudioWidth()
                                        mvGetTrackLength()
                                        The following SGI Digital Media Library functions are use
                                        dmParamsCreate()
Complete listing of the
                                        dmParamsDestroy()
Digital Media Library
                                        dmSetImageDefaults()
                                        dmSetAudioDefaults()
functions used in this
                                        dmParamsSetInt()
program
                                        dmParamsSetFloat()
                                        dmParamsSetEnum()
                                        dmParamsSetString()
                                        dmImageFrameSize()
```

**Figure 32-1** Comments in Movie Library Sample Programs: *createmovie.c++* 

Figure 32-2 shows the body of *createmovie.c++*, showing the modularity of this program—main() contains only 5 lines that call other modules.

```
#include "createmovieArgs.h"
Global functions or those
                         #include "createmovieInit.h"
needed by other modules are
                         #include "createmovieFiles.h"
put in header files.
                         #include <movie.h>
Local functions are defined
                           * Forward declarations of functions that appear below.
as static and have forward
declarations.
                         static void makeMovie( MVid *theMovie );
                          * main
                         main( int argc, char **argv )
                              MVid theMovie;
Modularization simplifies
                              processCmdArgs( argc, argv );
the program.
                              makeMovie( &theMovie );
                              mvClose( theMovie );
                              exit( EXIT_SUCCESS );
                           * Make the new movie file.
                          static void makeMovie( MVid *theMovie )
                              initMovie( theMovie );
                              putFilesInMovie( *theMovie );
```

**Figure 32-2** Modularity of Movie Library Sample Programs: *createmovie.c++* 

# **Creating Movies**

This section describes two sample programs that demonstrate how to use the Movie Library to create movies:

createmovie which creates a movie from any combination of image files,

audio files, or other movie files, letting you specify the compression scheme, loop mode, frame rate and size, and

the output movie filename from the command line

*img-to-movie* which creates a movie file from an image file

createmovie [-c compression] [-1 loopMode] [-r frameRate] [-s xsize,ysize] [-p paramtype, userParam, userParamVal] [-o outMovie] file[...]

#### where:

-c compression specifies the image compression scheme, where you enter

one of the following for *compression*:

jpeg DM\_IMAGE\_JPEG

mvc1 (default) DM\_IMAGE\_MVC1

mvc2 DM\_IMAGE\_MVC2

none DM\_IMAGE\_UNCOMPRESSED

rle DM\_IMAGE\_RLE

8rgb DM\_IMAGE\_UNCOMPRESSED

**-1** *loopMode* specifies the loop mode, where you enter one of the

following for loop Mode: once, loop, or swing

**-r** *frameRate* specifies the frame rate in frames per second

If one or more movie files are included in the *file* source material, *outmovie* will have the same frame rate as the first movie file on the command line; otherwise, if the frame rate is not set explicitly, the default is 15.0 frames per

second.

**-s** *xsize,ysize* specifies the frame size, where:

*xsize* is the horizontal dimension *ysize* is the vertical dimension

The default frame size is that of the first file on the command line containing frame size information, which might be either an image or a movie file.

You can set the frame size explicitly, which enlarges or reduces the images as required. Aspect ratios are preserved when scaling images, so that the resulting images might be letter-boxed if the aspect ratio of the source image is different from the aspect ratio specified by the given *xsize* and *ysize*.

-р paramtype, userParam, userParamVal

adds a user-defined parameter of type paramtype, named

userParam, whose value is userParamVal

-o *outmovie* specifies the name of the output movie

file includes one or more image, audio, or movie files

You can also use *createmovie* to convert a movie file from one compression scheme to another, by creating a new movie in the specified compression scheme from the old movie.

*createmovie* contains the following files:

createmovieArgs.c++ contains command line processing functions, access

to user preferences, and creation and access to the

filenames entered on the command line

createmovieArgs.h is the external interface to createmovieArgs.c

createmovieConvert.c is used only in conjunction with the Silicon Graphics

QuickTime Compressor Library to convert between

QuickTime and rgb image data

createmovieConvert.h is the external interface to *createmovieqtConvert.c++* 

createmovieFiles.c++ puts files into a new movie

createmovieFiles.h has external declarations for *createmovieFiles.c++* 

createmovieInit.c++ initializes a new movie file

createmovieInit.h has external declarations for *createmovieInit.c*++ createmovieResize.c++ performs filtering operations on image frames

createmovieResize.h has external declarations for *createmovieResize.c++* 

Figure 32-3 shows a call graph for *createmovie*. The call graph shows the overall structure of the program, indicating which modules call which functions. Some calls are omitted for the sake of clarity.

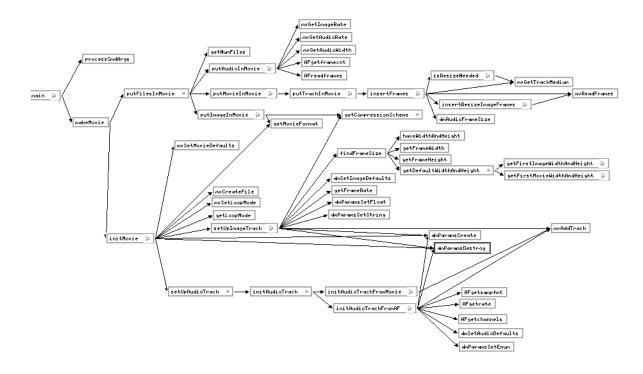


Figure 32-3 Call Graph for *createmovie* 

#### Creating a Movie from a Sequence of Images

The *img-to-movie.c* program in the */usr/people/4Dgifts/examples/libmovie/misc* directory creates a movie file from an image file. The image file can have a single image, in which case the movie file has only a single frame, or a sequence of images. The resulting movie file has a frame rate of 15.0 frames per second and uses MVC1 compression.

To run the program, enter:

img-to-movie imagefile newmoviefile

where:

imagefile is the name of the image file you want to make into a movie

*newmoviefile* is the name of the new movie

### Adding or Replacing a Movie Audio Track

The *aud-to-movie.c* program in the */usr/people/4Dgifts/examples/libmovie/misc* directory adds an audio track to a silent movie or replaces the movie's existing audio track. The audio track can be an AIFF file or any audio file that is readable by *libaudiofile*.

To run the program, enter:

aud-to-movie audiofile moviefile

where:

audiofile is the name of the audio track you want to put in the movie

*moviefile* is the name of the movie

# **Editing Movies**

The /usr/people/4Dgifts/examples/libmovie/editmovie directory contains editmovie.c, a simple command line program for editing movies, and its associated files. The editing operations available are insert, delete, and paste. Only one editing operation can be selected at a time.

To run the program, enter:

editmovie -e editMovie,trackType,firstEditFrame,numFrames
[-d] [-s sourceMovie,firstSrcFrame [-i] [-p] [-m] [-o outMovie]

#### where:

-e	is followed by four arguments:		
	editMovie	the name of the movie to be edited	
	trackType	the type of track to edit, either image or audio	
	firstEditFrame	the first frame to be edited	
	numFrames	the number of frames to be edited	
-d	deletes frames from editMovie		
-s	specifies both tl	ne source movie and first frame to copy	
-i	inserts frames f	rom sourceMovie into editMovie	
-p	pastes frames from sourceMovie into editMovie		
-m	performs editing on a memory-resident copy of the movie		
-0	optimizes the e	dited movie for playback and places it in	

*editmovie* has the following requirements:

- numFrames must be > 0
- firstSrcFrame must be  $\geq 0$
- when inserting (-i) or pasting (-p), the -s option must be supplied in order to specify the movie to copy from and the first frame to copy

- when inserting (-i) or pasting (-p), the sum of *numFrames* and *firstSrcFrame* must not exceed the last frame number in *sourceMovie*.
- when deleting (-d), the sum of *numFrames* and *firstEditFrame* must not exceed the last frame number in *editMovie*

editmovie contains the following files:

editmovie.c a simple command line movie editor

editmovieArgs.c a command-line-parsing module for editmovie

editmovieArgs.h declarations for external functions in editmovieArgs.c

editmovieEdit.c contains editTheMovie(), which is the only external

function, and its supporting functions

editmovieEdit.h external interface to editmovieEdit.c, which actually

performs the editing for editmovie

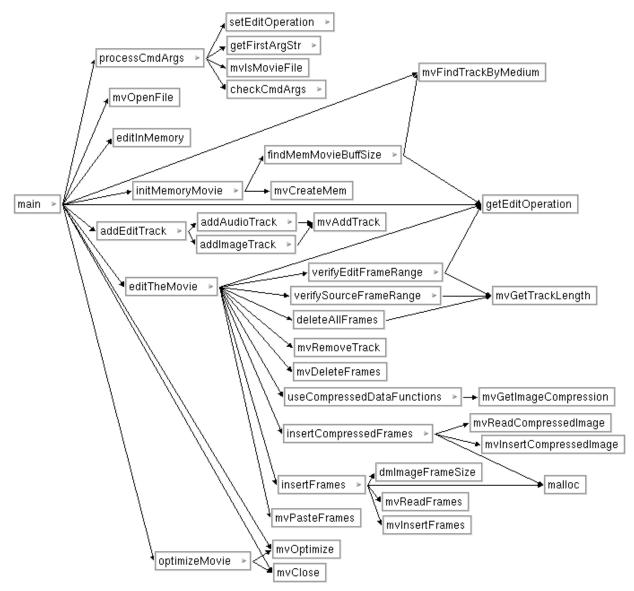


Figure 32-4 shows the call graph for *editmovie.c* 

**Figure 32-4** Call Graph for *editmovie.c* 

# **Displaying Movie Parameters**

*mvinfo* displays information about a movie, including its image track and audio track (if present) parameters. To run the program, enter: mvinfo moviefile

# **Playing Movies**

This section describes three sample programs that play movies:

*simplemovie* implements a keyboard interface for playing movies

manymovie plays up to four movies simultaneously

moviescreen implements a movie-based screen-saver application

# **Creating a Simple Keyboard Interface for Playing Movies**

*simplemovie* has a keyboard interface for playing a movie. To run the program, enter:

simplemovie moviefile

The keyboard commands are:

<3>	loop 3 times
<b> or <b></b></b>	play backward
<e> or <e></e></e>	play every frame
<f> or <f></f></f>	play fast
<h> or <h></h></h>	play slow
<1> or <l></l>	toggle loop state
<m> or <m></m></m>	toggle audio muting
or	play the movie
<q> or <q></q></q>	quit simplemovie
<r> or <r></r></r>	rewind the movie
<s> or <s></s></s>	stop playback

# **Playing Multiple Movies**

*manymovie* is a command line program for playing up to 4 movies simultaneously. The movies can have different frame sizes.

**Note:** The 4 movie limitation is not related to the Movie Library; rather, it is the result of the very simple layout scheme used in *manymovie* for placing all movies in one window.

To run the program, enter:

```
manymovie [-one] moviefile1 [moviefile2...]
```

By default, each of the movies appears in a separate window. When each movie is played in a separate window, the keyboard commands apply only to the current window, except for the *quit* command, which always exits from *manymovie*.

To play all movies in a single window, use the **-one** command line option. If the **-one** option is used, all keyboard commands apply simultaneously to all movies.

*manymovie* uses the following keyboard interface:

<1> or <l></l>	changes the looping state
<m> or <m></m></m>	toggles audio muting
or	plays the movie
<q> or <q></q></q>	quits manymovie
<r> or <r></r></r>	rewinds to the beginning of the movie
<s> or <s></s></s>	stops the movie

manymovie contains the following files:

*manymovie.c* is the main program, which plays several movies at

once.

*manymovieArgs.c* contains the code for processing command-line

arguments, creating and accessing the movie list, and

recording the number of movie

*manymovieArgs.h* contains the external interface to the command-line

argument processing code

manymovieEvents.c contains X and Movie event handling code

manymovieEvents.h is the external interface to manymovieEvents.c and

handles X and Movie events for manymovie

manymovieWin.c contains code for creating X windows suitable for

playing movies

*manymovieWin.h* is the external interface to manymovieWin.c, creates

X windows suitable for GL rendering for all the movies, opens the movies and the windows, provides

access to the X Display

#### Creating a Movie Screensaver Application

*moviescreen* is a screensaver application that plays movies. *moviescreen* does not save screens by itself; rather, it is designed to run under *haven*(1), a wrapper for IRIS GL-based screensavers, which is available on Silicon Graphics computers. Normally, the screen turns black and the movie begins playing silently, slowly drifting around the screen as it plays.

To run the program as a screensaver under *haven*(1), enter:

haven  $[-n \mid -o][< moviescreen [-f][-s][-v volume][-z zoom][-1 loopmode] moviefile1 [moviefile2...]> <math>|-k]$ 

#### where:

**-f** enables fullscreen playback

-s turns on sound for the movie (by default, screensaver

movies play silently)

-v *volume* sets the playback volume if the -s option is used

Enter a value from 0 to 255 for volume.

**-z** *zoom* zooms the movie larger for playback

Enter an integer value for zoom. Zooming takes effect only

if the **-f** option, fullscreen playback, is not used.

**-1** *loopmode* sets the loop mode.

To loop continuously (default), set loopmode to 0. To swing,

set it to 1.

*moviescreen* contains the following files:

moviescreen.c is the main program, which uses a movie as a

screensaver

moviescreenArgs.c contains code for helper functions used by

moviescreen for accessing/manipulating data entered by the user via command-line arguments; maintains this information as static variables that

are restricted to this module

moviescreenArgs.h contains functions for processing the command-

line arguments and accessing variables set

therefrom, including movie names

moviescreenEvents.c contains code to start, perform, and end screen

saving, depending on reception of XEvents

moviescreenEvents.h contains code to wait for an XEvent and initiate or

terminate screen saving as appropriate

moviescreenGl.c contains code used by moviescreen for determining

the movie window position and erasing the movie

window as it moves via GL drawing

moviescreenGl.h contains functions for controlling positioning and

erasing (via GL drawing) of the movie window

moviescreenWin.c contains code for creating and accessing an X

window suitable for GL rendering that is used by

moviescreen

moviescreenWin.h contains functions to create and access a mixed-

model GL window

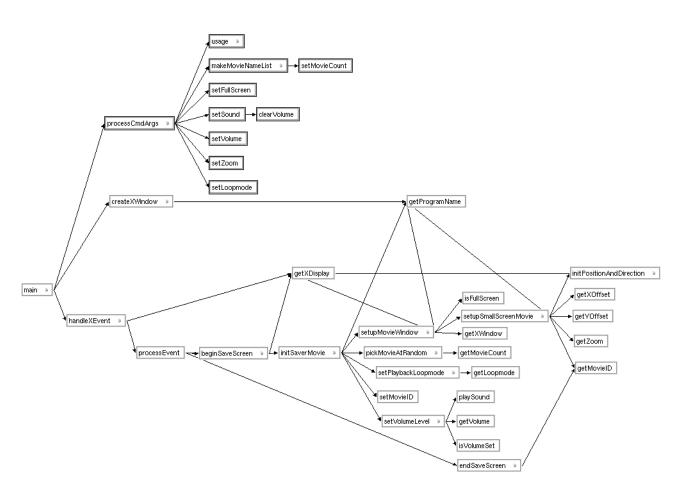


Figure 32-5 shows the call graph for *moviescreen.c* 

**Figure 32-5** Call Graph for *moviescreen.c* 

# **Using the SMPTE Time Code Sample Application**

Included with the Movie Library sample code is a group of utility routines in a sample application for handling SMPTE time codes. These routines work with the movie frame that is closest to the given time code rather than providing exact time code precision. Time code type is indicated by the *timetype* argument, which supports the four basic types of time codes listed in Table 32-1.

**Table 32-1** SMPTE Time Code Types

framereturn

Time Code Type	Meaning	
MV_TIME_SMPTE_24	24 frames/second (motion pictures)	
MV_TIME_SMPTE_25	25 frames/second (PAL video)	
MV_TIME_SMPTE_30	30 frames/second (NTSC)	
MV_TIME_SMPTE_D30	30 frame/second drop (NTSC 29.97)	

**Note:** Currently, only MV\_TIME\_SMPTE\_30 is supported.

# Converting a SMPTE Time Code String to a Frame Number

To access the frame closest to a given SMPTE code, pass the SMPTE code as a string to **mvStringToFrame()**, which converts the given string to a frame number in the specified movie. Its function prototype is:

```
extern DMstatus mvStringToFrame ( MVid movieid, char *timestring, MVtimetype timetype, MVframe *framereturn )

where:

timestring is the SMPTE time code string

timetype is the SMPTE code type
```

is a pointer to the frame number that is to be returned

Specify the time code string in the format "HH:MM:SS:FF" where HH signifies two digits that represent hours, and similarly, MM minutes, SS seconds, and FF the frame count for the current second. The converter assumes that incomplete time strings are specified from the least-significant unit up; that is, a value of "02" means SMPTE frame code 02 of the current second. "03:02" means second 03, frame 02 of the current minute, and so on. The frame number is between 0 and one less than the total number of frames in the movie.

# Converting a Frame Number to a SMPTE Time Code String

To obtain a SMTPE-style time code string for a given frame number, call **mvFrameToString()** with the frame number for which you need the SMPTE code. Its function prototype is:

```
extern DMstatus mvFrameToString ( MVid movieid, MVframe frame, MVtimetype timetype, char *timereturn)
```

#### where:

frame is the frame number for which you want to get the time code

timetype is the SMPTE code type

timereturn is a pointer into which the time code string is returned

The SMPTE code is returned as a string that lists the hour, minute, second, and frame count for the given frame number.

# **Converting a Time Specification to a Frame Number**

To convert a fully formed time code to the nearest corresponding frame in the specified movie, call **mvTimeToFrame()**. Its function prototype is:

extern	DMstatus	mvTimeToFrame	(	MVid	movieid,
				int	hour,
				int	minute,
				int	second,
				MVframe	framecnt,
				MVtimetype	timetype,
				MVframe	*framereturn)

#### where:

hour is the hour field of the time code
 minute is the minute field of the time code
 second is the second field of the time code
 framecnt is the frame field of the time code
 timetype is a time type from Table 32-1

*framereturn* is a pointer into which the frame number is returned

# **Converting a Frame Number to a Time Code**

To obtain a time code corresponding to a particular frame, call **mvFrameToTime()** with the index of the frame for which you want the time code. Its function prototype is:

${\tt DMstatus}$	mvFrameToTime	(	MVid	movieid,
			MVframe	frame,
			MVtimetype	timetype,
			int	*hourreturn,
			int	*minutereturn,
			int	*secondreturn,
			MVframe	*framecntreturn)

#### where:

frame is the frame number for which you want to get the time code

*timetype* is a time type from Table 32-1

*hourreturn* is a pointer into which the time code hour is returned

minutereturn is a pointer into which the time code minutes are returnedsecondreturn is a pointer into which the time code seconds are returnedframecntreturn is a pointer into which the time code frame is returned



# **Audio Specifications**

This appendix describes the audio hardware specifications for the Indigo and Indigo<sup>2</sup> workstations. It also contains some video specifications for IndigoVideo.

# **Indigo Workstation Audio Hardware Specifications**

This section contains the specifications for the Indigo audio hardware.

Unless otherwise stated, all audio hardware parameters are measured under the following conditions:

- analog input signal levels are +7 dB re: 1Vrms
- digital input signal levels are 100% of full scale
- 48 kHz sample rate
- input source impedances are  $600 \Omega$
- output destination impedances are  $5 \text{ k}\Omega$
- measurement bandwidth is 10 to 30 kHz unweighted

All measurement results are typical. All connectors are single-ended 3.5-mm stereo phono plugs.

# Indigo Analog Audio I/O

The following specifications describe the analog audio I/O.

# **Analog Stereo Line-level Inputs**

- Impedance:  $5 k\Omega$  nominal
- Amplitude at full scale: 1 Vpp to 10 Vpp
- Level control: two independent, digitally-controlled analog attenuators
- Frequency response: 20 Hz–20 kHz ±0.25 dB
- Total Harmonic Distortion + Noise (THD+N): < 0.003% @ 1 kHz, < 0.005% 20 Hz–20 kHz
- Residual noise: –90 dB unweighted, –93 dB A-weighted (re: full scale)
- Interchannel isolation: -76 dB @ 1 kHz, -68 dB @ 10 kHz, -77 dB @ 20 kHz
- Analog-to-Digital Converter resolution: 16-bit Delta-Sigma

# **Analog Mono Microphone Input**

- Impedance:  $2 k\Omega$
- Amplitude at full scale: 0.25 Vpp to 2.5 Vpp
- Power supply: +3 Vdc @ 1 mA

#### Microphone

- Type: omnidirectional electret condenser (powered by system)
- Output level: 65 dB 4 dB @ 1 kHz (+0 dB = 1 V/0.1 pa)
- Frequency response: 40 Hz–18 kHz 2 dB

## **Analog Stereo Line-level Outputs**

- Impedance:  $600 \Omega$  nominal
- Amplitude at full scale: 6.0 Vpp
- Frequency response: 20 Hz–20 kHz +0 dB, –0.8 dB

- Total Harmonic Distortion + Noise (THD+N): <0.005% @ 1 kHz,</li>
   <0.02% 20 Hz-20 kHz</li>
- Residual noise: –85 dB unweighted, –92 dB A-weighted (re: full scale)
- Interchannel isolation: -76 dB @ 1 kHz, -66 dB @ 10 kHz,
   -61 dB @ 20 kHz
- Digital output filter data resolution: 16-bit input, 18-bit output, 8× oversampling
- Digital-to-Analog Converter resolution: 18 bits
- Digital-to-Analog Converter sample rate: 8× oversampling

# Analog Stereo Headphone Outputs/Mono Internal Speaker

- Headphone output impedance: 16  $\Omega$
- Headphone level: 200 mW into 32  $\Omega$  load
- Headphone and speaker level control: two independent, digitally-controlled analog attenuators
- Speaker: 2.6" diameter dynamic
- Speaker sound pressure level: 88 dB/W
- Speaker output level: 3 Watts max, 1.5 Watts nominal

# Indigo Digital Audio I/O

The following specifications describe the digital audio serial I/O.

## **Digital Coaxial Serial Input**

- Impedance: 75  $\Omega$ , transformer-coupled
- Level: 1 Vpp
- Sample rates: 30 kHz to 50 kHz
- Resolution: supports up to 24 bits per sample
- Coding: AES-3, IEC-958

# **Digital Coaxial Serial Output**

- Impedance: 75 Ω, transformer-coupled
- Level: 1 Vpp into 75  $\Omega$  load
- Sample rates: 32, 44.1, 48 kHz, and divisors
- Resolution: supports up to 24 bits per sample
- Coding: AES-3, IEC-958

# **Indigo Dedicated Real-time Processor**

The dedicated real-time processor in the Indigo workstation has the following characteristics:

- Processor: 20 MHz Motorola DSP56001
- Native word length: 24 bits, fixed point
- DSP RAM: 32K word SRAM

# Indigo<sup>2</sup> Workstation Audio Hardware Specifications

This section contains the specifications for the Indigo<sup>2</sup> audio hardware.

# Indigo<sup>2</sup> Analog Stereo Line-level Inputs

- Impedance:  $20 \text{ k}\Omega$  nominal
- Amplitude at full scale: 0.63Vpp to 8.4Vpp
- Level control: analog gain control internal to CODECs
- Frequency response: 20Hz-20 kHz 0.81 dB
- Total Harmonic Distortion + Noise (THD+N): < 0.006% @ 1 kHz,</li>
   < 0.007% 20 to 20 kHz</li>
- Residual noise: –86 dB unweighted, –88 dB A-weighted (re: Full Scale)
- Interchannel isolation: -82 dB@1 kHz, -72 dB@10 kHz, -67 dB@20 kHz
- Analog-to-Digital Converter resolution: 16-bit

# Indigo<sup>2</sup> Stereo Microphone Input

- Impedance:  $1.5 \text{ k}\Omega$
- Amplitude at full scale: 0.063Vpp to 0.84Vpp
- Power supply: +3Vdc @ 1mA
- In 4-channel mode, the microphone input connector can be configured as a line-level input. When configured this way, it has the characteristics described in "Indigo2 Analog Stereo Line-level Inputs"

# Indigo<sup>2</sup> Analog Stereo Line-level Outputs

- Impedance:  $600 \Omega$  nominal
- Amplitude at full scale: 4.7 Vpp
- Frequency response: 20 Hz–20 kHz 1.2 dB
- Total Harmonic Distortion + Noise (THD+N): <0.02% 20 to 20 kHz
- Residual noise: –81 dB unweighted, –85 dB A-weighted (re: full scale)
- Interchannel isolation:-80 dB@1 kHz, -75 dB@10 kHz, -71 dB@20 kHz
- Digital-to-Analog converter resolution: 16-bit

# Indigo<sup>2</sup> Analog Stereo Headphone Output/Mono Internal Speaker

- Headphone output impedance:  $10 \Omega$
- Headphone level: 57mW into  $32 \Omega$  load
- Headphone and speaker level control: two independent, digitally-controlled analog attenuators
- Speaker: 70mm by 40mm
- Speaker sound pressure level: 80 dB @1W, 1meter
- Speaker output level: 5W max, 2W nominal
- In 4-channel mode, the headphone output connector is configured as a line-level output. When configured this way, it has the characteristics described in the "Indigo2 Analog Stereo Line-level Outputs"

# Indigo<sup>2</sup> Digital Audio I/O

The following specifications describe the digital audio serial I/O.

# **Digital Coaxial Serial Input**

Impedance: 75 Ω, transformer-coupled

Level: 1 Vpp

• Sample rates: 30 kHz to 50 kHz

• Resolution: supports up to 24 bits per sample

• Coding: AES-3, IEC-958

# **Digital Coaxial Serial Output**

• Impedance: 75  $\Omega$ , transformer-coupled

• Level: 1 Vpp into 75  $\Omega$  load

• Sample rates: 32, 44.1, 48 kHz, and divisors

• Resolution: supports up to 24 bits per sample

• Coding: AES-3, IEC-958

# Aware Scalable Audio Compression Software

This appendix describes built-in licensable compression software from Aware, Inc. Developers are encouraged to integrate Aware's audio compression products to add value to their applications that incorporate audio processing. Doing so requires that you add Network License System (NetLS<sup>TM</sup>) licensing support to your application, which is described in "Installing a NetLS Nodelocked License" on page 743.

Aware's compression software can be accessed through the following Silicon Graphics digital media development libraries:

- Audio File Library (AF)
- Compression Library (CL)

The AF and CL parameters to access Aware compression are provided in this appendix.

# Introduction to Aware Audio Compression Software

Aware offers audio compression software engines that enable users of Silicon Graphics workstations to reduce the storage size and transmission bandwidth required for audio data. These products are compatible with third-party applications that incorporate audio in the Silicon Graphics digital media computing environment. Scalable operation provides the ability to control processor loading for audio playback concurrently with other computational tasks.

Aware's software compression engines include:

- Aware AudioPublisher<sup>™</sup> MPEG and MultiRate audio codec Provides advanced psychoacoustic processing for compression of CD-quality sound in authoring, publishing, and large audio database applications. Compatible with MPEG-audio standard.
- Aware AudioProducer MultiRate audio codec

Provides a compression solution for studio and audio production applications requiring lossless and near-lossless coding and using low processor loading. Includes AudioProducer and AudioPlayback engines.

Aware AudioPlayback<sup>™</sup> audio decoder

Provides a decoding solution for applications requiring playback-only of compressed audio in multi-tasking processing environments.

Aware markets its audio compression software to end users. Once an end user licenses and activates Aware's audio compression engines in his or her workstation, the engines are available as a system resource and can work with any application that calls Aware's compression engines.

Portions of this software are

© Copyright 1993, Aware, Inc.—All Rights Reserved.

The Aware Software is proprietary and confidential. You may interface to the Aware Software by using the Audio File (AF) Library and the Compression Library (CL) supplied on this software release. You or any other user may gain authorization to execute the functions of the Aware Software (which includes a keylock mechanism) only by purchasing a usage license from Aware, Inc. Unauthorized use of the Aware Software is expressly forbidden. To obtain a usage license from Aware, contact:

Aware, Inc.
One Memorial Drive
Cambridge, MA 02142
phone: (617) 577-1700
fax: (617) 577-1710
email: sales@aware.com
Ask for audio products.

# **Aware Software Products Features and Applications**

This section describes the applications and features of Silicon Graphics digital media-compatible products from Aware.

# Aware Products Available in IRIS Digital Media Libraries

The IRIS digital media libraries contain three software products that end users can license by contacting Aware (see "Introduction to Aware Audio Compression Software" on page 729) to obtain a license password:

#### Aware AudioPublisher

The Aware AudioPublisher provides advanced psychoacoustic processing for compression of high-quality audio (up to 48 kHz sampling rate) in studio, authoring, and archiving applications.

AudioPublisher supports the MPEG (Moving Pictures Experts Group) audio standard format. The encoding process invokes advanced psychoacoustic modeling to achieve high compression ratios.

Encoding bit-rates range from 32 Kbits/second to 448 Kbits per second (corresponding to compression ratios as high as 48:1). AudioPublisher also includes Aware's AudioProducer and AudioPlayback software.

#### Aware AudioProducer

The Aware AudioProducer provides a compression solution for audio production applications requiring lossless (perfectly invertible) and near-lossless coding at low computational complexity.

Lossless mode reduces storage requirements by 2:1 to 3:1. Near-lossless operation achieves even greater storage savings, while retaining a 90 dB signal-to-noise ratio. Real-time encoding of audio at 8, 11.025, 16, 22.05, and 44.1 kHz sampling rates is supported. Scalable encoding and decoding operation provides control of processor loading during playback. AudioProducer includes Aware's AudioPlayback software.

#### Aware AudioPlayback

The Aware AudioPlayback engine provides a decoding solution for applications requiring decoding of compressed audio in multi-tasking processing environments. The decoder runs in real time, at sampling rates of up to 48 kHz. Installation of the decoder enables real-time

playback of compressed audio that was encoded with the Aware AudioPublisher or with the Aware AudioProducer. Scalable decoding operation provides the ability to control processor loading used for playback, and it enables special functions such as fast playback to let you speed-search through compressed audio libraries.

# Other Digital Media Compatible Aware Audio Products

These audio productivity software products are compatible with Aware's audio compression engines in the Silicon Graphics digital media environment. Contact Aware for more information on these products.

#### **Aware AudioSuite Tools**

The Aware AudioSuite <sup>™</sup> provides these graphical user interface (GUI) software applications for the end user that work with Aware's licensable software audio compression engines on Silicon Graphics platforms:

Audition<sup>™</sup> provides compressed file playback and extraction

Archiver<sup>™</sup> provides file-to-file compression batch processing

Psycoder provides optimal compression parameter selection

## Aware Speed-of-Sound Library, Volume I, Sound Effects

The Aware Speed-of-Sound  $^{\text{TM}}$  Library is a single CD-ROM that puts more than 1,000 digitally recorded sound effects at your fingertips. Cars, trains, planes, birds, bees, laughing, crying, water, and wind—all the effects you need to make your creative vision come alive. Each sound on this disk was professionally recorded at the 16-bit, 44.1 kHz CD-audio format. But unlike standard CDs that contain a maximum of 72 minutes of sound, the Aware Speed-of-Sound Library offers over seven hours of effects, along with the interactive  $BrowsFX^{\text{TM}}$  librarian tool that makes finding and auditioning any sound a snap.

#### Aware AudioSuite Package

This comprehensive package includes the Speed-of-Sound Library, Volume I: SFX, plus all audio compression engines and GUIs.

# **Accessing Aware Audio Compression from the Audio File Library**

This section describes the data formats, defaults, and parameters for accessing the Aware audio compression software engines from the Audio File Library.

# **Valid Audio Input Data**

Audio input data can be accepted in any of these formats:

- 16-bit two's complement samples
- single or dual channel
- sampling rates for MPEG must be 32 kHz, 44.1 kHz, or 48 kHz.
- sampling rates for MultiRate must be 8 kHz, 11.025 kHz, 16 kHz, 22.05 kHz, 32 kHz, 44.1 kHz, or 48 kHz.

# **Compression Defaults**

The simplest method of invoking Aware compression is to use this command to pick from one of four default settings:

```
AFinitcompression ( AFfilesetup setup, long track, long compression_scheme)
```

The defaults for the Aware compression parameters are:

AF\_COMPRESSION\_AWARE\_DEFAULT\_MPEG\_I
MPEG layer 1, joint-stereo, fixed rate at 192 Kbps/channel

AF\_COMPRESSION\_AWARE\_DEFAULT\_MPEG\_II MPEG layer 2, joint-stereo, fixed rate at 128 Kbps/channel

AF\_COMPRESSION\_AWARE\_DEFAULT\_MULTIRATE MultiRate, near-lossless

AF\_COMPRESSION\_AWARE\_DEFAULT\_LOSSLESS MultiRate, lossless

# **Compression Custom Configuration**

To access individual compression parameters, use the AF library call:

```
AFinitcompressionparams ( AFfilesetup setup, long track, long scheme, AUpvlist pybuffer, long buffersize)
```

The compression *scheme* should be passed as one of:

- AF\_COMPRESSION\_AWARE\_MPEG
- AF\_COMPRESSION\_AWARE\_MULTIRATE
- AF\_COMPRESSION\_AWARE\_DEFAULT\_MPEG\_I
- AF\_COMPRESSION\_AWARE\_DEFAULT\_MPEG\_II
- AF\_COMPRESSION\_AWARE\_DEFAULT\_MULTIRATE
- AF\_COMPRESSION\_AWARE\_DEFAULT\_LOSSLESS

The parameters and values passed in the *AUpvlist* structure include:

# AF\_AWARE\_PARAM\_LAYER

(valid only for algorithm AF\_AWARE\_MPEG) selects which MPEG layer:

- AF\_AWARE\_LAYER\_I
- AF\_AWARE\_LAYER\_II (default)

**Note:** AF\_AWARE\_PARAM\_CHANNEL\_POLICY chooses how multiple channels should be treated (these settings are equivalent for single channel input).

- AF\_AWARE\_STEREO
  indicates that the channels are part of a single
  multichannel signal, such as quadraphonic, and so on.
- AF\_AWARE\_JOINT\_STEREO (default for MPEG) indicates that the algorithm may attempt to exploit redundancy between channels for greater coding gain. Not valid for AF\_AWARE\_MULTIRATE.
- AF\_AWARE\_INDEPENDENT (default for MultiRate) indicates that the separate channels are unrelated and should be processed separately, such as multilingual sound tracks.

#### AF\_AWARE\_PARAM\_BITRATE\_TARGET

specifies the desired bitrate for all channels of compressed data, in bits per second. Note that for some schemes such as MPEG's *maxrate* (not yet implemented) this is treated as an upper limit, whereas for MPEG's *fixrate*, this is strictly achieved as a constant rate. This parameter is not used for Aware's *MultiRate* algorithm.

The following is a list of valid bitrates for MPEG:

Layer 1: 32000, 64000, 96000, 128000, 160000, 192000, 224000, 256000, 288000, 320000, 352000, 384000, 416000, and 448000.

Layer 2: 32000, 48000, 56000, 64000, 80000, 96000, 112000, 128000, 160000, 192000, 224000, 256000, 320000, and 384000.

Default value is 192 Kbps/channel for layer 1 and 128 Kbps/channel for layer 2.

#### AF\_AWARE\_PARAM\_BITRATE\_POLICY

selects variants for interpreting AF\_AWARE\_PARAM\_BITRATE\_TARGET. The valid values depend on the compression type:

• Aware MPEG (AF\_COMPRESSION\_AWARE\_MPEG) uses either:

AF\_AWARE\_FIXED\_RATE (default) fixed bitrate per second, where the compression ratio is set by AF\_AWARE\_PARAM\_BITRATE\_TARGET or

AF\_AWARE\_CONST\_QUAL

lets the bitrate be driven by the psychoacoustic model. Enough bits are assigned so that a constant noise-to-mask ratio is attained. See AF\_AWARE\_CONST\_QUAL\_NMR.

Aware MultiRate
 (AF\_COMPRESSION\_AWARE\_MULTIRATE) uses
 either:

AF\_AWARE\_LOSSLESS supplies enough bits to provide for perfect reconstruction. Compression ratios are typically between 2:1 and 3:1.

or

AF\_AWARE\_CONST\_QUAL enough bits are assigned so that the signal is 90+ dB above the quantization noise. Compression ratios are typically between 2.5:1 and 4:1.

# AF\_AWARE\_CONST\_QUAL\_NMR

not used in the Aware *MultiRate* algorithm. For AF\_COMPRESSION\_AWARE\_MPEG with AF\_AWARE\_CONST\_QUAL, it sets the constant quality mode noise-to-mask ratio in dB. Zero yields a theoretical psychoacoustically imperceptible compression. Positive values provide more compression and noise becomes audible. Negative values cause less compression and less perceptible noise. The type is AU\_PVTYPE\_DOUBLE.

# **Accessing Aware Audio Compression from the Compression Library**

The Aware audio compression engines are already installed in the Silicon Graphics Compression Library uniform interface to video and audio compression routines. You can incorporate these routines into your applications and set up license querying to enable end users who choose to do so to license the routines. This section describes how to use the Compression Library interface to the Aware engines.

# Compression Schemes

Aware currently provides two distinct compression schemes under the CL:

CL\_AWARE\_MPEG\_AUDIO

ISO/MPEG-audio standard algorithm with layers I and II.

CL\_AWARE\_MULTIRATE

Aware MultiRate I proprietary lossless or low-distortion algorithm.

Both scheme identifiers are defined in *cl.h*, as are all the identifiers in this section with the prefix CL\_. Identifiers prefixed AWCMP\_ are defined in *awareAudio.h*.

# **Using Compression Library Parameters**

The CL is controlled by a wide range of parameters. Some parameters have multiple uses—others are appropriate only in certain circumstances. This list explains the use of all the parameters of relevance to the Aware schemes:

### CL ORIGINAL FORMAT

indicates the format of the components of the original uncompressed audio. Default is CL\_STEREO\_INTERLEAVED, legal alternative is CL\_MONO.

#### **CL COMPONENTS**

indicates the number of channels in the original uncompressed audio. Only single and dual channel signals are currently supported.

## CL\_BITS\_PER\_COMPONENT

informs the scheme of the format of the audio samples. They are assumed to be two's complement linear ints with the specified number of bits. The default is 16, which is also currently the only legal value.

# CL\_FRAME\_RATE

tells the scheme the sampling rate of the uncompressed audio. Default is 44100.0 Hz. Legal values for scheme CL\_AWARE\_MPEG are 32000.0, 44100.0, and 48000.0. Legal values for scheme CL\_AWARE\_MULTIRATE are 8000.0, 11025.0, 16000.0, 22050.0, 32000.0, 44100.0, and 48000.0 Hz.

## CL\_SPEED

during decompression, this parameter controls the computational complexity by controlling several factors: sample rate decimation, combining of stereo channels, filter shape (CL\_AWARE\_MPEG only), and lossy decode (CL\_AWARE\_MULTIRATE only). Acceptable values range from 1.0 to 100.0. CL\_SPEED has no effect during compression.

**Note:** Changing this parameter may change the sampling rate, the total number of sample frames available, and the sample frames per compressed block. After changing this parameter, you must call **clGetParams()** to find out the new CL\_FRAME\_RATE and CL\_BLOCK\_SIZE.

#### CL\_CHANNEL\_POLICY

selects the treatment of stereo signals:

- AWCMP\_STEREO stereo channels are coded separately.
- AWCMP\_JOINT\_STEREO stereo redundancy is exploited by coding high frequency sub-bands together.
- AWCMP\_INDEPENDENT channels are unrelated and are coded separately, such as multilingual sound tracks.

**Note:** Scheme CL\_AWARE\_MPEG\_AUDIO supports all three modes; CL\_AWARE\_MULTIRATE supports only AWCMP\_STEREO and AWCMP\_INDEPENDENT.

#### CL\_BITRATE\_POLICY

Provides for different modes of bit assignment:

- AWCMP\_FIXED\_RATE
   fixed bitrate for CL\_AWARE\_MPEG\_AUDIO only. See
   CL\_BITRATE\_TARGET. This is the default for
   CL\_AWARE\_MPEG\_AUDIO.
- AWCMP\_CONST\_QUAL
   bitrate is allowed to vary to satisfy MPEG
   psychoacoustic model or *MultiRate* data requirements
   (90+ dB). This is the default for
   CL\_AWARE\_MULTIRATE. See CL\_NOISE\_MARGIN.
- AWCMP\_LOSSLESS valid for CL\_AWARE\_MULTIRATE only. Data is coded losslessly.

## CL\_NOISE\_MARGIN

used in conjunction with CL\_BITRATE\_POLICY's AWCMP\_CONST\_QUAL. Provides for specification of the noise-to-mask ratio in MPEG psychoacoustics. Zero yields a theoretical psychoacoustically imperceptible compression. Positive values provide more compression and noise becomes audible. Negative values cause less compression

and less perceptible noise. For the *MultiRate* algorithm, this parameter sets the noise-floor at –90 dB below the signal level. This is the only valid *MultiRate* setting at this time.

CL\_LAYER

(MPEG only) selects the MPEG layer. Default is AWCMP\_MPEG\_LAYER\_II for layer II, other legal value is AWCMP\_MPEG\_LAYER\_I for layer I.

## CL\_BITRATE\_MODE

provides for different modes of operation such as fixed rate or constant quality; however, the only legal values at present are AWCMP\_FIXED\_RATE for MPEG and AWCMP\_CONST\_QUAL or AWCMP\_LOSSLESS for *MultiRate*.

#### CL\_BITRATE\_TARGET

used in conjunction with CL\_BITRATE\_POLICY's AWCMP\_FIXED\_RATE (MPEG). Determines the output data rate in bits per second. The allowed values are:

- MPEG layer I: 32000, 64000, 96000, 128000, 160000, 192000, 224000256000, 288000, 320000, 352000, 384000, 416000, 448000.
- MPEG layer II: 32000, 48000, 56000, 64000, 80000, 96000, 112000,128000, 160000, 192000, 224000, 256000, 320000, 384000.

# CL\_COMPRESSED\_BUFFER\_SIZE

When a scheme is configured, this field can be queried to find the maximum compressed size of a block of CL\_BLOCK\_SIZE samples. This is useful for allocating buffers to hold this compressed data. If the application sets this parameter (to inform the CL of the size of an externally allocated buffer), the compression code does not change that value, even if its configuration is modified.

The following parameters are read-only:

# CL\_BLOCK\_SIZE

The scheme writes the number of uncompressed sample frames to compress at a time by the scheme. It is strongly advised that data be passed to **clCompress()** in blocks that

are exact multiples of this size; otherwise, the algorithm will defer to compression of partial frames, which may cause unexpected problems.

# CL\_COMPRESSION\_RATIO

The scheme writes a compression ratio into this parameter. Parameter CL\_EXACT\_COMPRESSION\_RATIO will be true if this is an exact compression ratio and false if it is a worst-case estimate.

# CL\_EXACT\_COMPRESSION\_RATIO

Indicates whether the compression ration in CL\_EXACT\_COMPRESSION\_RATIO is exact (TRUE), or a worst-case estimate (FALSE). This parameter is always true for pure MPEG, because it is a fixed-bitrate scheme, and false otherwise; therefore, this parameter is currently of interest only for the MPEG scheme.

#### CL\_FRAME\_TYPE

indicates the status of the next frame to be processed:

- CL\_KEYFRAME, meaning that the frame is the first frame in a compressed block, which is what it *should* be if all calls are in BLOCK SIZE blocks.
- CL\_PREDICTED is the frame type for all other alignments.

# **Usage Hints**

You'll get the best compression results if you keep these hints in mind:

- Always compress blocks of CL\_BLOCKSIZE sample frames.
- The first block that comes out of an instance of a compression scheme
  has extra configuration information attached in front of it. On
  decompression, this should be fetched first (you will have to query its
  maximum size), then fed to clReadHeader(). Thereafter, the
  CL\_COMPRESSED\_BUFFER\_SIZE can be read and used to allocate a
  proper data buffer, if so desired.
- No provision has been made for alignment of delay as a result of filter latency through the Compression Library (which can be a few hundred samples depending on scheme), or for zero-padding the final frame to

ensure that the filter is flushed. These must be handled by the application, if required.

# **Aware Audio Compression Software Specifications**

This section lists the specifications for the Aware audio compression software.

- Compression Algorithms
  - ISO/MPEG Layer I and II low bitrate psychoacoustic compression
  - Aware MultiRate I high resolution audio compression
- Library Support
  - Silicon Graphics Audio File Library (AF)
  - Silicon Graphics Compression Library (CL)
- Encoder Input

Format: 16-bit linear audio samples, mono or stereo

Sample Rates:

MPEG Encoder 48, 44.1, 32 kHz

MultiRate Encoder 48, 44.1, 32, 22.05, 16, 11.025 kHz

Decoder Output

Format: 16 bit linear audio samples, mono or stereo

Sample Rates: 48, 44.1, 32, 24, 22.05, 16, 11.025 (MPEG, MultiRate

Decoders)

Compression Ratios:

2:1 to 3:1 typical *MultiRate* lossless (perfectly invertible) mode

2.5:1 to 4:1 typical *MultiRate* near-lossless (90+ dB signal to noise)

mode

2.2:1 to 48:1 selectable, MPEG Layer I and II

Bitrates (Kbits/sec):

MPEG Layer I 32, 64, 96, 128, 160, 192, 224, 256, 288, 320, 352, 384, 416, 448

MPEG Layer II 32, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320, 384

Channel Processing

MPEG Layer I and II mono, stereo, joint stereo, dual channel

MultiRate mono, stereo

Scalable Processing

Controlled reduction of CPU usage by parametric control of frequency response, signal to noise ratio, and mono decoding of stereo.

Table B-1 lists the compression algorithms that are built in to each Aware software compression engine product.

 Table B-1
 Built-in Algorithms for Aware Audio Software Compression Engines

Product	Compression Algorithms Installed
AudioPublisher	ISO/MPEG Layer I and II Encoder
	Aware MultiRate I Encoder
	ISO/MPEG Layer I and II Decoder
	Aware MultiRate Decoder
AudioProducer	Aware MultiRate I Encoder
	ISO/MPEG Layer I and II Decoder
	Aware MultiRate Decoder
AudioPlayback	ISO/MPEG Layer I and II Decoder
	Aware MultiRate Decoder

# Installing a NetLS Nodelocked License

A product that is licensed with NetLS can use either a nodelocked license or a concurrent access license. On the *Silicon Graphics Vendor and License Information* sheet, in the *licensetype* field, it lists either "Concurrent Access" or "NodeLocked" specifying the type of license you have been issued.

This section documents nodelocked licenses. With a nodelocked license, the product can be run only on the machine that has the nodelocked license installed. To enable a nodelocked license, the "vendor ID" and "product password" from the *Silicon Graphics Vendor and License Information* sheet are entered into the file /usr/netls/nodelock. After that, a product should have permission to run on the machine with the nodelock file.

#### To install a nodelocked license:

- 1. Verify that the product you want to license is already installed, by entering:
  - % /usr/sbin/versions
- 2. For IRIX release 5.0 or earlier, netls\_eoe and nck subsystems do not need to be installed. For IRIX release 5.0.1 or later, it is recommended that netls\_eoe be installed, in order to create the link from /usr/netls/nodelock to /var/netls/nodelock. This link is needed by some NetLS licensed products that are not Silicon Graphics products.
- 3. The *llb*, *glb*, and *netls* flags, which you can set with the */etc/chkconfig* command, do not affect whether a nodelocked license will work, but it is a good idea to set these flags correctly anyway. If this machine is not being used as a NetLS or glb database server, *chkconfig* these flags off. To change the flags, enter this sequence of commands:
  - # su
  - # /etc/chkconfig glb off
  - # /etc/chkconfig llb off
  - # /etc/chkconfig netls off
- 4. Edit the *nodelock* file. For IRIX release 5.0 or earlier, the *nodelock* file is /usr/netls/nodelock. For IRIX release 5.0.1 or later, the *nodelock* file is /var/netls/nodelock. A license in the *nodelock* file takes up one line and looks something like this:

549db468491e.02.c0.1a.3d.52.00.00.00 4cb3cwxxy29awcv9998xa

where the first string is a vendor ID and the second string is the product password. Comment lines are ignored. A comment line is a line whose first character is a "#".

If a temporary nodelocked license is already installed for the product for which you are adding a longer term license, comment out that temporary license line by putting a "#" in front of the line. If no other products are nodelocked and other licenses exist in this file, comment all of the licenses out by putting a "#"character in front of them.

5. Continue editing the *nodelock* file to add the vendor ID and the product password for the product. Put them on the same line in the *nodelock* file with the vendor ID first, followed by a space, then the product password.

It is *strongly* recommended to add a comment line before the license that describes what product the license is for and when it expires.

See the online *NetLS Administration Guide* for more information and troubleshooting guidelines.

# Glossary

#### active video

The portion of the video signal containing the chrominance or luminance information; all video lines not occurring in the vertical blanking signal containing the chrominance or luminance information. See also *chrominance*, *composite video*, *horizontal blanking*, *luminance*, and *video waveform*.

# aliasing

One of several types of digital video artifact appearing as jagged edges. Aliasing results when an image is sampled that contains frequency components above the Nyquist limit for the sampling rate. See also *Nyquist limit*.

## alpha

See alpha value.

#### alpha blending

Overlaying one image on another so that some of the underlying image may or may not be visible. See also *key*.

# alpha plane

A bank of memory that stores alpha values; the values are 8 bits per pixel.

# alpha register

Registers that stores an alpha value.

## alpha value

The component of a pixel that specifies the pixel's opacity, translucency, or transparency. The alpha component is typically output as a separate component signal.

#### antialiasing

Filtering or blending lines of video to smooth the appearance of jagged edges in order to reduce the visibility of aliasing.

#### **APL**

Average Picture Level, with respect to blanking, during active picture time, expressed as a percentage of the difference between the blanking and reference white levels. See also *blanking level*.

#### artifact

In video systems, an unnatural or artificial effect that occurs when the system reproduces an image; examples are aliasing, pixellation, and contouring.

#### aspect ratio

The ratio of the width to the height of an electronic image. For example, the standard aspect ratio for television is 4:3.

# back porch

The portion of the horizontal pedestal that follows the horizontal synchronizing pulse. In a composite signal, the color burst is located on the back porch, but is absent on a YUV or GBR signal. See also *blanking level*, *video waveform*.

#### **Betacam**

A component videotape format developed by Sony<sup>®</sup> that uses a Y/R-Y/B-Y video signal and 1/2-inch tape.

# **Betacam format**

Advanced form (Superior Performance) of Betacam using special metal tape and offering longer recording time (90 minutes instead of 30 minutes) and superior performance.

#### bit map

A region of memory that contains the pixels representing an image. The pixels are arranged in the sequence in which they are normally scanned to display the image.

# bitplane

One of a group of memory arrays for storing an image in bitmap format on a workstation. The workstation reads the bitplanes in parallel to re-create the image in real time.

#### black burst

Active video signal that has only black in it. The black portion of the video signal, containing color burst. See also *color burst*.

#### black level

In the active video portion of the video waveform, the voltage level that defines black. See also *horizontal blanking* and *video waveform*.

## blanking level

The signal level at the beginning and end of the horizontal and vertical blanking intervals, typically representing zero output (0 IRE). See also *video waveform* and *IRE units*.

#### blend

To combine proportional amounts of a 3D graphic over a clip frame by frame, pixel by pixel, with the alpha determining how they are combined. See also *key, frame*, and *alpha*.

#### breezeway

In the horizontal blanking part of the video signal, the portion between the end of the horizontal sync pulse and the beginning of the color burst. See also *horizontal blanking* and *video waveform*.

# broad pulses

Vertical synchronizing pulses in the center of the vertical interval. These pulses are long enough to be distinguished from other pulses in the signal; they are the part of the signal actually detected by vertical sync separators.

#### **Bruch blanking**

In PAL signals, a four-field burst blanking sequence used to ensure that burst phase is the same at the end of each vertical interval.

#### burst, burst flag

See color burst.

#### burst lock

The ability of the output subcarrier to be locked to input subcarrier, or of output to be genlocked to an input burst.

#### burst phase

In the RS-170A standard, burst phase is at field 1, line 10; in the European PAL standards, it is at field 1, line 1. Both define a continuous burst waveform to be in phase with the leading edge of sync at these points in the video timing. See also *vertical blanking interval* and *video waveform*.

# B-Y (B minus Y) signal

One of the color difference signals used on the NTSC and PAL systems, obtained by subtracting luminance (Y) from the blue camera signal (B). This signal drives the horizontal axis of a vectorscope. Color mixture is close to blue; phase is 180 degrees opposite of color sync burst; bandwidth is 0.0 to 0.5MHz. See also *luminance*, *R-Y signal*, *Y signal*, and *Y/R-Y/B-Y*.

#### C signal

Chrominance; the color portion of the signal. For example, the Y/C video format used for S-VHS has separate Y (luminance) and C (chrominance) signals. See also *chrominance*.

## CAV

Component Analog Video; a generic term for all analog component video formats, which keep luminance and chrominance information separate. D1 is a digital version of this signal. See also *component video*.

#### C format

Type C, or one-inch reel-to-reel videotape machine; an analog composite recording format still used in some broadcast and postproduction applications.

#### **CCIR 601**

The digital interface standard developed by the CCIR (Comite' Consultatif International de Radiodiffusion, International Radio Consultative Committee) based on component color encoding, in which the luminance and chrominance (color difference) sampling frequencies are related in the ratio 4:2:2: four samples of luminance (spread across four pixels), two samples of  $C_R$  color difference, and two samples of  $C_R$  color difference. The

standard, which is also referred to as 4:2:2, sets parameters for both 525-line and 625-line systems.

#### chroma

See chrominance.

# chroma keying

Overlaying one video source on another by choosing a key color. For example, if chroma keying is on blue, video source A might show through video source B everywhere the color blue appears in video source B. A common example is the TV weather reporter standing in front of the satellite weather map. The weather reporter, wearing any color but blue, stands in front of a blue background; keying on blue shows the satellite picture everywhere blue appears. Because there is no blue on the weatherperson, he or she appears to be standing in front of the weather map.

## chroma signal

A 3.58MHz (NTSC) or 4.43MHz (PAL) subcarrier signal for color in television. SECAM uses two frequency-modulated color subcarriers transmitted on alternate horizontal lines;  $SC_R$  is 4.406MHz and  $SC_B$  is 4.250MHz.

# chrominance

In an image reproduction system, a separate signal that contains the color information. Black, white, and all shades of gray have no chrominance and contain only the luminance (brightness) portion of the signal. However, all colors have both chrominance and luminance.

Chrominance is derived from the I and Q signals in the NTSC television system and the U and V signals in the PAL television system. See also *luminance*.

## chrominance signal

Also called the chroma, or C, signal. The high-frequency portion of the video signal (3.58MHz for NTSC, 4.43MHz for PAL) color subcarrier with quadrature modulation by I (R-Y) and Q (B-Y) color video signals. The amplitude of the C signal is saturation; the phase angle is hue. See also *color subcarrier*, *hue*, and *saturation*.

#### client

In the context of the Video Library, an application that has connected to the video daemon to perform video requests.

#### clip

Segment of video, audio, or both. An image is a clip that is one frame long.

#### color bars

A test pattern used by video engineers to determine the quality of a video signal, developed by the Society of Television and Motion Picture Engineers (SMPTE). The test pattern consists of equal-width bars representing black, white, red, green, blue, and combinations of two of the three RGB values: yellow, cyan, and magenta. These colors are usually shown at 75% of their pure values. Figure Gl-1 diagrams the color bars.

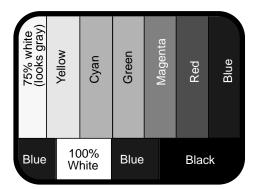
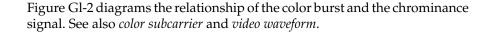


Figure GI-1 SMPTE Color Bars (75%)

#### color burst

Also called burst and burst flag. The segment of the horizontal blanking portion of the video signal that is used as a reference for decoding color information in the active video part of the signal. The color burst is required for synchronizing the phase of 3.58MHz oscillator in the television receiver for correct hues in the chrominance signal.

In composite video, the image color is determined by the phase relationship of the color subcarrier to the color burst. The color burst sync is 8 to 11 cycles of 3.58MHz color subcarrier transmitted on the back porch of every horizontal pulse; The hue of the color sync phase is yellow-green.



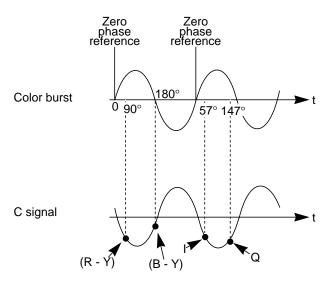


Figure GI-2 Color Burst and Chrominance Signal

# color difference signals

Signals used by color television systems to convey color information so that the signals go to zero when the picture contains no color; for example, unmodulated R-Y and B-Y, I and Q, U, and V.

# color-frame sequence

In NTSC and S-Video, a two-frame sequence that must elapse before the same relationship between line pairs of video and frame sync repeats itself. In PAL, the color-frame sequence consists of four frames.

# color space

A space defined by three color components, such as R, G, and B.

### color subcarrier

A portion of the active portion of a composite video signal that carries color information, referenced to the color burst. The color subcarrier's amplitude determines saturation; its phase angle determines hue. Hue and saturation

are derived with respect to the color burst. Its frequency is defined as 3.58MHz in NTSC and 4.43MHz in PAL. See also *color burst*.

### complementary color

Opposite hue and phase angle from a primary color. Cyan, magenta, and yellow are complementary colors for red, green, and blue, respectively.

# comb filtering

Process that improves the accuracy of extracting color and brightness portions of the signal from a composite video source.

# component video

A color encoding method for the three color signals—R, G, and B; Y, I, and Q; or Y, U, and V—that make up a color image. See also *RGB*, *YIQ*, and *YUV*.

# component video signals

A video signal in which luminance and chrominance are send as separate components, for example:

- RGB (basic signals generated from a camera)
- YIQ (used by the NTSC broadcasting standard)
- Y/R-Y/B-Y (used by Betacam and M-II recording formats and SECAM broadcasting standard)
- YUV (subset of Y/R-Y/B-Y used by the PAL broadcasting standard)

Separating these components yields a signal with a higher color bandwidth than that of composite video.

Figure Gl-3 depicts video signals for one horizontal scan of a color-bar test pattern. The RGB signals change in relation to the individual colors in the test pattern. When a secondary color is generated, a combination of the RGB signals occurs. Since only the primary and secondary colors are being displayed at 100% saturation, the R, G, and B waveforms are simply on or off. For more complex patterns of color, the individual R, G, and B signals would be varying amplitudes in the percentages needed to express that particular color.

See also *composite video*, *RGB*, *YUV*, *Y/R-Y/B-Y*, and *YIQ*.

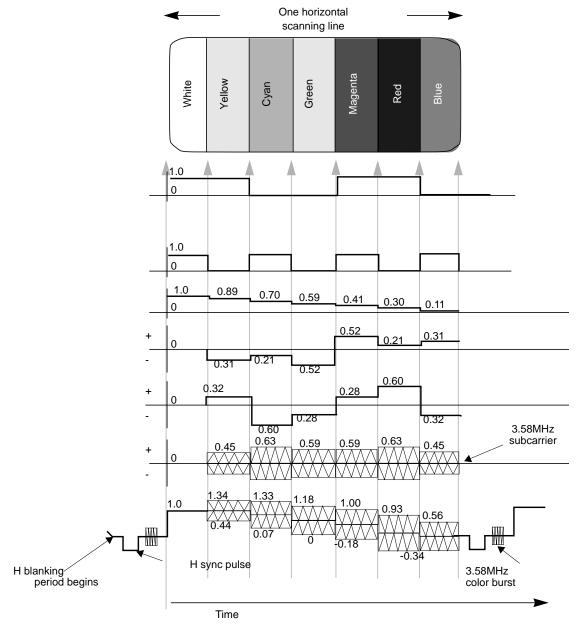


Figure GI-3 Component Video Signals

# compositing

Combining graphics with another image.

# composite video

A color encoding method or a video signal that contains all of the color, brightness, and synchronizing information in one signal. The chief composite television standard signals are NTSC, PAL, and SECAM. See also NTSC, PAL, and SECAM.

# cross-chrominance, cross-luminance

Also known as cross-color, hanging dots, dot crawl; moving colors on stationary objects. This undesirable artifact is caused by high bandwidth luminance information being misinterpreted as color information. Hanging dots are a byproduct of the comb filters (used to help separate the color and brightness information) found in most modern television receivers. This artifact can be reduced or eliminated by using S-Video or a component video format.

### cross-fade

A type of transition in which one video clip is faded down while another is faded up.

# **D1**

Digital recording technique for component video; also known as CCIR 601, 4:2:2. D1 is the best choice for high-end production work where many generations of video are needed. D1 can be an 8-bit or 10-bit signal. See also *CCIR* 601.

# D2

Digital recording technique for composite video. As with analog composite, the luminance and chrominance information is sent as one signal. A D2 VTR offers higher resolution and can make multiple generation copies without noticeable quality loss, because it samples an analog composite video signal at four times the subcarrier (using linear quantization), representing the samples as 8-bit digital words. D2 is not compatible with D1.

### D3, DX

Developed by Panasonic, a 1/2-inch tape version of D2. More often called DX.

### decoder

Hardware or software that converts, or decodes, a composite video signal into the various components of the signal. For example, to grab a frame of composite video from a VHS tapedeck and store it as an RGB file, it would have to be decoded first. Several Silicon Graphics video options have on-board decoders.

# dithering

Approximating a signal value on a chroma-limited display device by producing a matrix of color values that fool human perception into believing that the signal value is being reproduced accurately. For example, dithering is used to display a true-color image on a display capable of rendering only 256 unique colors, such as IndigoVideo images on a Starter Graphics display.

#### drain

In the context of the Video Library, a target or consumer of video signals.

# editing

The process in which data is examined, created, and modified. In video, the part of the postproduction process in which the finished videotape is derived from raw video footage. Animation is a subset of editing.

# encoder

Device that combines the R, G, and B primary color video signals into hue and saturation for the C portion of a composite signal. Several Silicon Graphics video options have on-board encoders.

# equalizing pulse

Pulse of one half the width of the horizontal sync pulse, transmitted at twice the rate of the horizontal sync pulse, during the portions of the vertical blanking interval immediately before and after the vertical sync pulse. The equalizing pulse makes the vertical deflection start at the same time in each interval, and also keeps the horizontal sweep circuits in step during the portions of the vertical blanking interval immediately before and after the vertical sync pulse.

#### event

Exceptional or noteworthy condition produced during video processing, such as loss of sync, dropping of frames or fields, and synchronization with other applications.

### exclusive use

A term applied to usage of the video data stream and controls on a pathway. A pathway in exclusive-use mode is available for writing of controls only to the client that requested the exclusive use, yet any application may read the controls on that pathway.

#### fade

To modify the opacity and/or volume of a clip. A faded-up clip is unaffected, a clip faded down to 50% has 50% less opacity or volume, and a faded-down clip is completely transparent of turned off.

### field

One of two (or more) equal parts of information in which a frame is divided in interlace scanning. A vertical scan of a frame carrying only its odd-numbered or its even-numbered lines. The odd field and even field make up the complete frame. See also *frame* and *interlace*.

### field averaging

A filter that corrects flicker by averaging pixel values across successive fields. See also *flicker*.

# field blanking

The blanking signals at the end of each field, used to make the vertical retrace invisible. Also called vertical blanking; see *vertical blanking* and *vertical blanking interval*.

### filter

To process a clip with spatial or frequency domain methods. Each pixel is modified by using information from neighboring (or all) pixels of the image. Filter functions include blur (low-pass) and crisp (high-pass).

### flicker

The effect caused by a one-pixel-deep line in a high-resolution graphics frame that is output to a low-resolution monitor, because the line is in only

one of the alternating fields that make up the frame. This effect can be filtered out by field averaging. See also *field* and *frame*.

### frame

The result of a complete scanning of one image. In television, the odd field (all the odd lines of the frame) and the even field (all the even lines of the frame) make up the frame. In motion video, the image is scanned repeatedly, making a series of frames.

# freeze, freeze-frame

A condition on the digitized video signal where the digitizing is stopped and the contents of the signal appear frozen on the display or in the buffer. Sometimes used to capture the video data for processing or storage.

# frequency

Signal cycles per second.

# frequency interlace

Placing of harmonic frequencies of C signal midway between harmonics of horizontal scanning frequency Fh. Accomplished by making color subcarrier frequency exactly 3.579545 MHz. This frequency is an odd multiple of H/2.

# front porch

The portion of the video signal between the end of active video and the falling edge of sync. See also *back porch*, *horizontal blanking*, and *video waveform*.

# **G-Y signal**

Color mixture close to green, with a bandwidth 0.0MHz to 0.5MHz. Usually formed by combining B-Y and R-Y video signals.

### gamma correction

Correction of gray-scale inconsistency. The brightness characteristic of a CRT is not linear with respect to voltage; the voltage-to-intensity characteristic is usually close to a power of 2.2. If left uncorrected, the resulting display has too much contrast and detail in black regions is not reproduced.

To correct this inconsistency, a correction factor using the 2.2 root of the input signal is included, so that equal steps of brightness or intensity at the input are reproduced with equal steps of intensity at the display.

# genlocking

Synchronizing with another video signal serving as a master timing source. The master timing source can be a composite video signal, a video signal with no active video (only sync information), or, for video studio, a device called house sync. When there is no master sync available, VideoFramer, for example, can be set to "free run" (or "stand-alone") mode, so that it becomes the master timing device to which other devices sync. See also *line lock*.

# gray-scale

Monochrome or black-and-white, as in a monitor that does not display color.

### H rate

Number of complete horizontal lines, including trace and retrace, scanned per second.

### **HDTV**

High-definition television. Though there is more than one proposal for a broadcast standard for HDTV, most currently available equipment is based on the 1125/60 standard, that is, 1125 lines of video, with a refresh rate of 60Hz, 2:1 interlacing (same as NTSC and PAL), and aspect ratio of 16:9 ( $1920 \times 1035$  viewable resolution), trilevel sync, and 30MHz RGB and luminance bandwidth.

### Hi-8mm

An 8mm recording format developed by Sony; accepts composite and S-Video signals.

# horizontal blanking

The period when the electron beam is turned off, beginning when each scan line finishes its horizontal path (scan) across the screen (see Figure Gl-4).

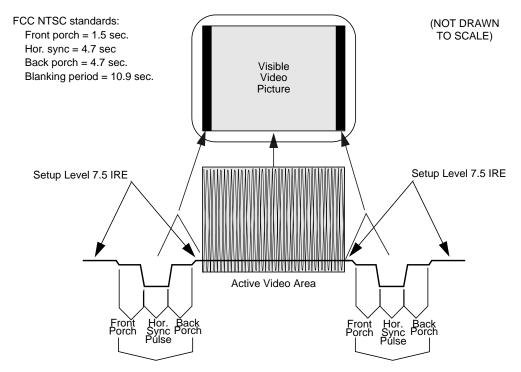


Figure GI-4 Horizontal Blanking

# horizontal blanking interval

Also known as the horizontal retrace interval, the period when a scanning process is moving from the end of one horizontal line to the start of the next line. This portion of the signal is used to carry information other than video information. See also *video waveform*.

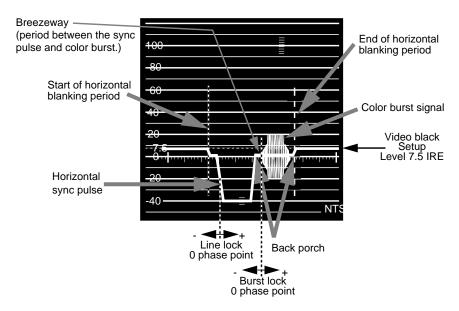


Figure GI-5 Horizontal Blanking Interval

# horizontal drive

The portion of the horizontal blanking part of the video signal composed of the sync pulse together with the front porch and breezeway; that is, horizontal blanking minus the color burst. See also *video waveform*.

# horizontal sync

The lowest portion of the horizontal blanking part of the video signal, it provides a pulse for synchronizing video input with output. Also known as h sync. See also *horizontal blanking* and *video waveform*.

#### HSI

See hue-saturation-intensity.

### **HSV**

Hue-saturation-value; see hue-saturation-intensity.

#### hue

The designation of a color in the spectrum, such as cyan, blue, magenta. Sometimes called tint on NTSC television receivers. The varying phase angles in the 3.58MHz (NTSC) or 4.43MHz (PAL) C signal indicate the different hues in the picture information.

# hue-saturation-intensity

A tri-stimulus color system based on the parameters of hue, saturation, and intensity (luminance). Also referred to as HSI or HSV.

# I signal

Color video signal transmitted as amplitude modulation of the 3.58 MHz C signal (NTSC). The hue axis is orange and cyan. This signal is the only color video signal with a bandwidth of 0 to 1.3 MHz.

# image plane

See bitplane.

# image processing

Manipulating an image by changing its color, brightness, shape, or size.

### interlace

A technique that uses more than one vertical scan to reproduce a complete image. In television, the 2:1 interlace used yields two vertical scans (fields) per frame: the first field consists of the odd lines of the frame, the other of the even lines. See also *field* and *frame*.

### **IRE** units

A scale for measuring analog video signal levels, normally starting at the bottom of the horizontal sync pulse and extending to the top of peak white. Blanking level is 0 IRE units and peak white level is 100 IRE units (700mv). An IRE unit equals 7.14mv (+100 IRE to -40 IRE = 1v). IRE stands for Institute of Radio Engineers, a forerunner of the IEEE.

### keying

Combining proportional amounts of two frames, pixel by pixel, with optional opacity. This process resembles taking two panes of glass with images on them and placing one pane on top of the other. The opacity of the top pane determines the parts of the bottom pane that show. Usually, keying is a real-time continuous process, as in the "over the shoulder" graphics in TV news programs. The alpha component of each pixel, which defines its opacity, determines how the images are combined. Combining images based on the alpha component is often called alpha keying or luma keying. See also *compositing* and *mixing*.

# leading edge of sync

The portion of the video waveform after active video, between the sync threshold and the sync pulse. See also *video waveform*.

### level

Signal amplitude.

#### line

The result of a single pass of the sensor from left to right across the image.

### line blanking

The blanking signal at the end of each horizontal scanning line, used to make the horizontal retrace invisible. Also called horizontal blanking.

# line frequency

The number of horizontal scans per second, normally 15,734.26 times per second for NTSC color systems. The line frequency for the PAL 625/50H. system is 15,625 times per second.

# line lock

Input timing that is derived from the horizontal sync signal, also implying that the system clock (the clock being used to sample the incoming video) is an integer multiple of the horizontal frequency and that it is locked in phase to the horizontal sync signal. See also at *video waveform*.

### linear matrix transformation

The process of combining a group of signals through addition or subtraction; for example, RGB signals into luminance and chrominance signals.

#### live video

Video being delivered at a nominal frame rate appropriate to the format.

#### luma

See luminance.

#### **luminance**

The video signal that describes the amount of light in each pixel. Luminance is a weighted sum of the R, G, and B signals. See also *chrominance* and *Y signal*.

### map

Numerical lookup of pixel data that modifies each pixel without using neighboring pixels. This large category of video editing functions includes clip/gain, solarization, and histogram equalization.

### MII (M2)

A second-generation recording format based on a version of the Y/R-Y/B-Y video signal. Developed by Panasonic, MII is also marketed by other video manufacturers. Though similar to Betacam, it is nonetheless incompatible.

# matrix transformation

The process of converting analog color signals from one tristimulus format to another, for example, RGB to YUV. See also *tristimulus color system*.

### mixing

In video editing, combining two clips frame by frame, pixel by pixel. Usually, a linear interpolation between the pixels in each clip is used, with which one can, for example, perform a cross-fade. Other operations include averaging, adding, differencing, maximum (non-additive mix), minimum, and equivalence (white where equal, else black). See also *compositing* and *keying*.

### multiburst

A test pattern consisting of sets of vertical lines with closer and closer spacing; used for testing horizontal resolution of a video system.

#### NTSC

A color television standard or timing format encoding all of the color, brightness, and synchronizing information in one signal. Used in North America, most of South America, and most of the Far East, this standard is named after the National Television Systems Committee, the standardizing body that created this system in the U.S. in 1953. NTSC employs a total of 525 horizontal lines per frame, with two fields per frame of 262.5 lines each. Each field refreshes at 60Hz (actually 59.94Hz).

# **Nyquist limit**

The highest frequency of input signal that can be correctly sampled without aliasing. The Nyquist limit is equal to half of the sampling frequency.

### offset

In the context of a video signal, the relative coordinates from the upper left corner of the video image where signal sampling begins.

### overscan

To scan a little beyond the display raster area of the monitor so that the edges of the raster are not visible. Television is overscanned; computer displays are underscanned.

# **PAL**

A color television standard or timing format developed in West Germany and used by most other countries in Europe, including the United Kingdom but excluding France, as well as Australia and parts of the Far East. PAL employs a total of 625 horizontal lines per frame, with two fields per frame of 312.5 lines per frame. Each field refreshes at 50Hz. PAL encodes color differently from NTSC. PAL stands for Phase Alternation Line or Phase Alternated by Line, by which this system attempts to correct some of the color inaccuracies in NTSC. See also *NTSC* and *SECAM*.

# pathway

In the Video Library, a connection of sources and drains that provide useful processing of video signals. Pathways have controls and video streams. Pathways can be locked for exclusive use, and are the target of events generated during video processing. See also *exclusive use* and *event*.

### pedestal

See setup; see also video waveform.

# pixel

Picture element; the smallest addressable spatial element of the computer graphics screen. A digital image address, or the smallest reproducible element in analog video. A pixel can be monochrome, gray-scale, or color, and can have an alpha component to determine opacity or transparency. Pixels are referred to as having a color component and an alpha component, even if the color component is gray-scale or monochrome.

# pixel map

A two-dimensional piece of memory, any number of bits deep. See also *bitmap*.

# postproduction

The processes that occur before release of the finished video product, including editing, painting (2D graphics), production, and 3D graphics production.

### primary colors

Red, green, and blue. Opposite voltage polarities are the complementary colors cyan, magenta, and yellow.

# Q signal

The color video signal that modulates 3.58MHz C signal in quadrature with the I signal. Hues are green and magenta. Bandwidth is 0.0MHz to 0.5MHz. See also *C signal*, *I signal*, *YC*, and *YIQ*.

### quantization error

The magnitude of the error introduced in a signal when the actual signal is between levels, resulting from subdividing a video signal into distinct increments, such as levels from 0 to 255.

### raster

The scanning pattern for television display; a series of horizontal lines, usually left to right, top to bottom. In NTSC and PAL systems, the first and last lines are half lines.

# raster operation, raster op

A logical or arithmetic operation on a pixel value.

# registration

The process of causing two frames to coincide exactly. In component video cameras or displays, the process of causing the three color images to coincide exactly, so that no color fringes are visible.

### resolution

Number of horizontal lines in a television display standard; the higher the number, the greater a system's ability to reproduce fine detail.

### **RGB**

Red, green, blue; the basic component set used by graphics systems and some video cameras, in which a separate signal is used for each primary color.

#### **RGB** format

The technical specification for NTSC color television. Often (incorrectly) used to refer to an RGB signal that is being sent at NTSC composite timings, for example, a Silicon Graphics computer set to output 640 x 480. The timing would be correct to display on a television, but the signal would still be split into red, green and blue components. This component signal would have to go through an encoder to yield a composite signal (RS-170A format) suitable for display on a television receiver.

# R-Y (R minus Y) signal

A color difference signal obtained by subtracting the luminance signal from the red camera signal. It is plotted on the 90 to 270 degree axis of a vector diagram. The R-Y signal drives the vertical axis of a vectorscope. The color mixture is close to red. Phase is in quadrature with B-Y; bandwidth is 0.0MHz to 0.5MHz. See also *luminance*, *B-Y* (*B minus Y*) *signal*, *Y/R-Y/B-Y*, and *vectorscope*.

### sample

To read the value of a signal at evenly spaced points in time; to convert representational data to sampled data (that is, synthesizing and rendering).

### sampling rate, sample rate

Number of samples per second.

#### saturation

Color intensity; zero saturation is white (no color) and maximum saturation is the deepest or most intense color possible for that hue. Different saturation values are varying peak-to-peak amplitudes in the 3.58MHz modulated C signal. In signal terms, saturation is determined by the ratio between luminance level and chrominance amplitude. See also *hue*.

# scaling

To change the size of an image.

#### scan

To convert an image to an electrical signal by moving a sensing point across the image, usually left to right, top to bottom.

#### **SECAM**

Sequential Couleur avec Memoire, the color television system developed in France and used there as well as in eastern Europe, the Near East and Mideast, and parts of Africa and the Caribbean.

# setup

The difference between the blackest level displayed on the receiver and the blanking level (see Figure Gl-6). A black level that is elevated to 7.5 IRE instead of being left at 0.0 IRE, the same as the lowest level for active video. Because the video level is known, this part of the signal is used for black-level clamping circuit operation. Setup is typically used in the NTSC video format and is typically not used in the PAL video format; it was originally introduced to simplify the design of early television receivers, which had trouble distinguishing between video black levels and horizontal blanking. Also called pedestal.

Figure Gl-6 shows waveform displays of a signal with and without setup. See also *video waveform*.

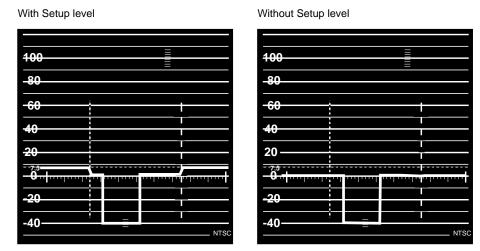


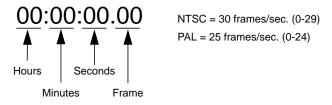
Figure GI-6 Waveform Monitor Readings with and without Setup

### smear

An artifact usually caused by mid-frequency distortions in an analog system that results in the vertical edges of the picture spreading horizontally.

# **SMPTE** time code

A signal specified by the Society of Motion Picture and Television Engineers for facilitating videotape editing; this signal uniquely identifies each frame of the video signal. Program originators use vertical blanking interval lines 12 through 14 to store a code identifying program material, time, frame number, and other production information (see Figure Gl-7).



**Figure GI-7** SMPTE Time Code

#### source

In the context of the Video Library, a provider of video input signals.

### subcarrier

A portion of a video signal that carries a specific signal, such as color. See *color subcarrier*.

# subpixel

A unit derived from a pixel by using a filter for sizing and positioning.

# S-VHS, S-Video

Video format in which the Y (luminance) and C (chrominance) portions of the signal are kept separate. Also known as YC.

# sync information

The part of the television video signal that ensures that the display scanning is synchronized with the broadcast scanning. See also *video waveform*.

# sync pulse

A vertical or horizontal pulse (or both) that determines the display timing of a video signal. Composite sync has both horizontal and vertical sync pulses, as well as equalization pulses. The equalization pulses are part of the interlacing process.

# sync tip

The lowest part of the horizontal blanking interval, used for synchronization. See also *video waveform*.

### synchronize

To perform time shifting so that things line up.

# texturing

Applying images to three-dimensional objects to give additional realism to displayed renderings.

#### termination

To send a signal through a transmission line accurately, there must be an impedance at the end which matches the impedance of the source and of the line itself. Amplitude errors, frequency response, and pulse distortions and reflections (ghosting) occur on a line without proper termination. Video is a 750hm system; therefore a 750hm terminator of .5% to .25% accuracy must be installed at the end of the signal path.

### threshold

In a digital circuit, the signal level that is specified as the division point between levels used to represent different digital values; for example, the sync threshold is the level at which the leading edge of sync begins. See also *video waveform*.

### time-base errors

Analog artifacts caused by nonuniform motion of videotape or of the tape head drum. Time-base errors usually cause horizontal display problems, such as horizontal jitter.

### time code

See SMPTE time code.

# time-delay equalization

Frame-by-frame alignment of all video inputs to one sync pulse, so that all frames start at the same time. This alignment is necessary because cable length differences cause unequal delays. See *time-base errors*.

### transcoder

A device that converts a component video signal to a different component video signal, for example, RGB to Y/R-Y/B-Y, or D1 to RGB.

### transducer

A microphone, video camera, or other device that can convert sounds or images to electrical signals.

### transform

The geometric perspective transformation of 3-D graphics models and planar images.

# tristimulus color system

A system of transmitting and reproducing images that uses three color signals, for example, RGB, YIQ, and YUV.

### U signal

One of the chrominance signals of the PAL color television system, along with V. Sometimes referred to as B-Y, but U becomes B-Y only after a weighting factor of 0.493 is applied. The weighting is required to reduce peak modulation in the composite signal.

# **U-Matic**

Sony trademark of its 3/4-inch composite videotape format. SP U-Matic is an improved version using metal tape.

#### underscan

To scan a television screen so that the edges of the raster are visible. See also *overscan*.

# V signal

One of the chrominance signals of the PAL color television system, along with U. Sometimes referred to as R-Y, but V becomes R-Y only after a weighting factor of 0.877 is applied. The weighting is required to reduce peak modulation in the composite signal.

# vectorscope

A specialized oscilloscope that demodulates the video signal and presents a display of R-Y versus B-Y for NTSC (V and U for PAL). Video engineers use vectorscopes to measure the amplitude (gain) and phase angle (vector) of the primary (red, green, and blue) and the secondary (yellow, cyan, and magenta) color components of a television signal.

# vertical blanking

The portion of the video signal that is blanked so that the vertical retrace of the beam is not visible.

# vertical blanking interval

The blanking portion at the beginning of each field. It contains the equalizing pulses, the vertical sync pulses, and vertical interval test signals (VITS). Also the period when a scanning process is moving from the lowest horizontal line back to the top horizontal line.

# video level

Video signal amplitude.

# video output

See drain.

# video signal

The electrical signal produced by a scanning image sensor.

# videotape formats

Table Gl-1 lists major videotape formats.

 Table GI-1
 Videotape Formats

Electronics	Consumer	Professional	Broadcast	Postproduction
Analog	VHS cassette	U-Matic (SP) cassette, 3/4-inch	Type C reel-to-reel, 1-inch composite	
	S-VHS		Type B (Europe), composite	
	S-Video (YC-358)	S-Video (YC-358)		
	Beta		Betacam (component)	
	8mm		Type MII (component)	
	Hi-8mm (YC)	Hi-8mm (YC)		
Digital				D1 525/625 (YUV)
				D2 525 (NTSC)
				D2 625 (PAL)

# video waveform

The main components of the video waveform are the active video portion and the horizontal blanking portion. Certain video waveforms carry information during the horizontal blanking interval.

Figure Gl-8 and Figure Gl-9 diagram a typical red or blue signal, which carries no information during the horizontal blanking interval, and a typical Y or green-plus-sync signal, which carries a sync pulse.

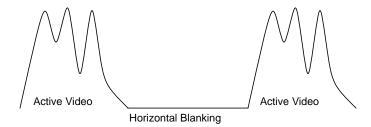


Figure GI-8 Red or Blue Signal

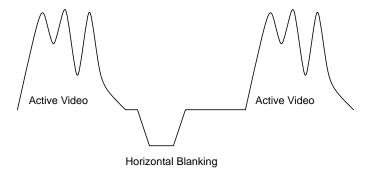
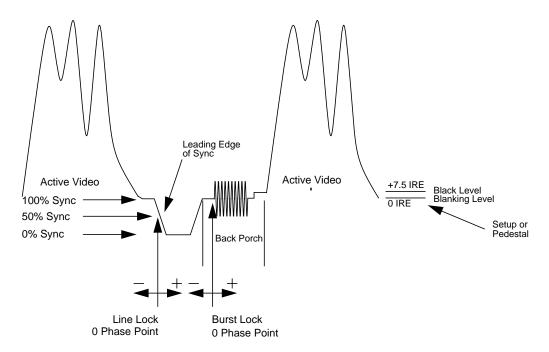


Figure GI-9 Y or Green Plus Sync Signal

Figure Gl-10 and Figure Gl-11 show the video waveform and its components for composite video in more detail. The figures show the composite video waveform with and without setup, respectively.

Figure Gl-10 shows a composite video signal with setup.



**Figure GI-10** Video Waveform: Composite Video Signal With Setup (Typical NTSC)

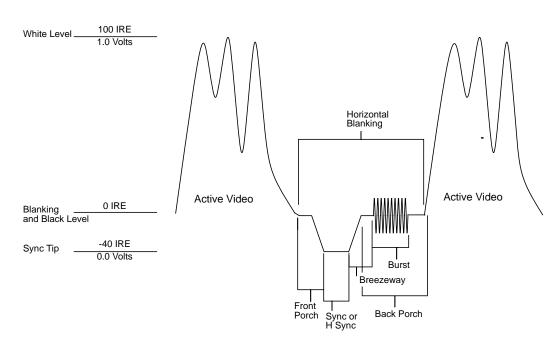


Figure Gl-11 shows a composite video signal without setup.

Figure Gl-11 Video Waveform: Composite Video Signal (Typical PAL)

# white level

In the active video portion of the video waveform, the 1.0-volt (100 IRE) level. See also *video waveform*.

# Y signal

Luminance, corresponding to the brightness of an image. See also luminance and Y/R-Y/B-Y.

#### YC

A color space (color component encoding format) based on YIQ or YUV. Y is luminance, but the two chroma signals (I and Q or U and V) are combined into a composite chroma called C, resulting in a two-wire signal. C is derived from I and Q as follows:

 $C - I \cos(2 \operatorname{fsct}) + Q \sin(2 \operatorname{fsct})$ 

where fsc is the subcarrier frequency. YC-358 is the NTSC version of this luminance/chrominance format; YC-443 is the PAL version. Both are referred to as S-Video formats.

### YIQ

A color space (color component encoding format) used in decoding, in which Y is the luminance signal and I and Q are the chrominance signals. The two chrominance signals I and Q (in-phase and quadrature, respectively) are two-phase amplitude-modulated; the I component modulates the subcarrier at an angle of 0 degrees and the Q component modulates it at 90 degrees. The color burst is at 33 degrees relative to the Q signal.

The amplitude of the color subcarrier represents the saturation values of the image; the phase of the color subcarrier represents the hue value of the image.

Y = 0.299R + 0.587G + 0.114B I = 0.596R - 0.275G - 0.321BQ = 0.212R - 0.523G + 0.311B

### Y/R-Y/B-Y

A name for the YUV color component encoding format that summarizes how the chrominance components are derived. Y is the luminance signal and R-Y and B-Y are the chrominance signals. R-Y (red minus Y) and B-Y (blue minus Y) are the color differences or chrominance components. The color difference signals R-Y and B-Y are derived as follows:

Y = 0.299R + 0.587 + 0.114B

Y/R-Y/B-Y has many variations, just as NTSC and PAL do. All component and composite color encoding formats are derived from RGB without scan standards being changed. The matrix (amount of red, green, and blue) values and scale (amplitude) factors can differ from one component format to another (YUV, Y/R-Y, B-Y, SMPTE Y/R-Y, B-Y).

### YUV

A color space (color component encoding format) used by the PAL video standard, in which Y is the luminance signal and U and V are the chrominance signals. The two chrominance signals U and V are two-phase amplitude-modulated. The U component modulates the subcarrier at an angle of 0 degree, but the V component modulates it at 90 degrees or 180 degrees on alternate lines. The color burst is also line-alternated at +135 and -135 degrees relative to the U signal. The YUV matrix multiplier derives colors from RGB via the following formula:

$$Y = .299R + .587 G + .114 B$$
  
 $C_R = R - Y$   
 $C_B = B - Y$ 

In this formula, Y represents luminance; red and blue are derived from it:  $C_R$  denotes red and (V),  $C_B$  denotes blue. V corresponds to  $C_R$ ; U corresponds to  $C_B$  c. The U and V signals are carried on the same bandwidth. This system is sometimes referred to as Y/R-Y/B-Y.

The name for this color encoding method is YUV, despite the fact that the order of the signals according to the formula is YVU.

# Index

Numbers	AES3-1985 (ANSI S4.40-1985), 136
	AF_FILE_AIFF, 134
4-channel audio	AF_FILE_AIFFC, 134
cabling, 49-50	AFclosefile(), 149
configuring hardware, 101	AFfilehandle, 132
frames illustrated, 70	AFfilesetup, 132
Indigo <sup>2</sup> , 46	creating, 132
Indy, 46	defaults, 132
input, 94	freeing, 133
output, 96	parameters, 132
querying, 111	setting, 134
4Dgifts, 61	AFfreefilesetup(), 133
8mm video, 258	AFgetaeschanneldata(), 155
	AFgetchannels(), 155
	AFgetcompression, 156
A	AFgetcompressionname(), 158
adding	AFgetcompressionparams(), 156
movie	AFgetfd(), 148
parameters, 598	AFgetfilefmt(), 150
example, 599, 600	AFgetframecnt(), 158
tracks, 622-623	AFgetinstids(), 163
movie tracks	AFgetinstparamlong(), 164
example, 623	AFgetloopend(), 167
adding algorithms to the Compression Library, 567	AFgetloopmode(), 165
ADPCM, 138	AFgetloopstart(), 166
AES	AFgetmarkids(), 159
channel status bytes, 136	AFgetmarkname(), 159
jacks, 136	AFgetmarkpos(), 160
resolutions, 71	AFgetmiscids(), 168
standard, 136	111 Semiiscius(), 100

AFgetmiscsize(), 169	AFsetaeschanneldata(), 136
AFgetmisctype(), 168	AFseterrorhandler(), 128
AFgetrate(), 152	AFsetinstparamlong(), 165
AFgetsampfmt(), 152	AFsetloopend(), 167
AFidentifyfd(), 150	AFsetloopmode(), 166
AFinitaeschanneldata(), 136	AFsetloopstart(), 167
AFinitchannels(), 135	AFsetmarkpos(), 160
AFinitcompression(), 136	AFsyncfile(), 149
AFinitcompressionparams(), 136	AFwriteframes(), 162
AFinitfilefmt(), 134	AFwritemisc(), 170
AFinitinstids(), 140	AIFF, 131
AFinitloopids(), 140	AIFF-C, 129-??, 131, ??-131
AFinitmarkids(), 139	chunks, 131
AFinitmarkname, 139	ALcloseport(), 88
AFinitmiscids(), 141	ALconfigs, 74-87
AFinitmiscsize(), 142	cloning, 85
AFinitmisctype(), 141	creating, 76
AFinitrate, 134	default, 74 defined, 69
AFinitsampfmt(), 135	freeing, 85
AF Library, 127-180	ALerrfunc, 73
AFfilehandle, 132	ALfreeconfig(), 85
chunks, 129, 131	ALgetchannels(), 78
compiling, 62	ALgetconfig(), 85
error handling, 128 example, 177, ??-180	ALgetdefault(), 104
file formats, 127, 134	ALgetfd(), 120
tasks, 127	ALgetfillable(), 92
AFLibrary	ALgetfilled(), 92, 93
AFfilesetup, 132	ALgetfillpoint(), 121
AFnewfilesetup(), 132	ALgetfloatmax, 84
AFopenfd(), 148	ALgetminmax(), 104
AFopenfile(), 147	ALgetname(), 105
AFreadframes(), 161	ALgetparams(), 106
AFreadmisc(), 170	ALgetqueuesize(), 79
AFseekframe(), 161	ALgetsampfmt(), 81
AFseekmisc(), 170	ALgetstatus(), 93, 97

ALgetwidth(), 83	animation, 476
algorithm-independent compression, 478	ANSI C, 63
algorithms	apanel, 57, 58, 59, 60
adding to the Compression Library, 563	API
Compression Library, 537	Compression Library, 491
allocating, 142	Apple Computer, Inc.
buffers	AIFF format, 131
audio, 21	applications
image, 24	Movie Library, 580
parameter-value lists, 17	arenas
ALnewconfig(), 76	shared, 123
ALopenport(), 88	example, 124-??
ALports, 74-89	assertions
allocating and initializing, 88	DM Library, 16
closing and deallocating, 88	Movie Library, 590
configuring, 74-87	attenuation
example, 76	audio, 99, 101
counting, 100	audio
defined, 68	4-channel
features, 74	cabling, 49-50
opening and closing, 87-89	adding a movie soundtrack, 623
example, 89 static settings, 75	attenuation, 99, 101
<u> </u>	breaks
AL and described in the AL and	cause, 78, 91
ALreadsamps(), 94	troubleshooting, 97
conversions, 95	buffer size, 21
fill points, 121	CD Audio Library
ALsetchannels(), 77	changing
errors and returns, 77	global state, 107 channels for movies, 605
ALseterrorhandler(), 73	clipping, 57
ALsetfillpoint(), 120	compression
<b>ALsetparams()</b> , 104, 106	Aware, Inc., 61
ALsetqueuesize(), 79	compression for movies, 606
ALsetsampfmt(), 80-81	concurrent, 57, 59
ALsetwidth(), 82-83	confidence tests, 60
ALwritesamps(), 96	configurations, 74-87
fill points, 121	cloning, 85
analog-to-digital (A/D) converters, 71	connections, 68,74

conversions, 72	interleaving, 70
core global parameters, 99	I/O
defaults, 20	errors, 97-98
floating point range, 84	mutiplexing, 119
I/O rate, 59	libraries, 6
movie, 603	limiting, 57
port, 74	memory requirements, 50
devices, 68	monitoring, 100
digitizing, 69	movie
discontinuities, 97	defaults
disk space, 50	volume, 604
distortion, 107	native formats, 605
error handlers, 73	properties, 603
examples, 61	track properties, 603
file descriptors, 119	native formats, 71
fill points, 120	nonblocking I/O, 92
illustrated, 121	Nyquist Theorem, 69
formats, 71	output, 95-96
formats for movies, 605	conversions, 96
frames, 70	overflow, 93
illustrated, 70	overflow and underflow, 98
getting file descriptors, 120	illustrated, 91
global state, 59, 98	parameters, 19
hardware specifications, 723, 726	current value, 106
icons, 44	getting and setting, 102-107
Indigo	names, 105
features, 43-44	setting, 104, 106
hardware, 43-45	special features, 101
jacks, 44	parameter-value buffer, 102
Indigo <sup>2</sup>	performancetuning, 97
features, 46	ports, 74-89
hardware, 46-47	allocating and initializing, 88
jacks, 46, 47	channels, 77
Indy	closing and deallocating, 88
features, 46-50	configuring, 74-87
hardware, 46-50	counting, 100
jacks, 48	default, 74
volume buttons, 48	defined, 68
input, 94-95	example, 76
4-channel, 94	formats, 80
conversions, 95	movies, 649

names, 88	troubleshooting, 97
opening and closing, 87-89	system software, 55
example, 89	third-party software, 61-62
queue size, 79	threshold, 120
static settings, 75	time required for output, 95
precision, 81	tools, 60
prioritizing, 122	underflow, 93
priority, 57	writing samples, 95-96
process control, 122	Audio Engineering Society. See AES
programming	Audio File (AF) Library. See AF Library
guidelines, 59	Audio File Library
quality, 69	purpose, 7
quantization steps, 84	
querying	Audio File Library, libaudiofile
4-channel, 111	accessing Aware compression from, 733
concurrent processes, 107-108	audio files
I/O rates, 108-110	AES, 155
supported features, 99	AES channel status bytes, 136
queues, 90-93	AIFF-C, 129-131
illustrated, 91	channels, 135, 136
size, 78-80, 92	closing, 149
size limits, 79	compression, 131, 136, 156
status, 92	name, 158
thresholds, 120	configuring tracks, 134-135
reading and writing data, 93-96	creating and configuring, ??-133
real-time programming, 119-??	defaults, 132
references, 40	editing
resolutions, 71, 82	soundeditor, 60
sample rate for movies, 604	emphasis, 135
samplers, 130	file descriptors, 148
sample width for movies, 604	file formats, 150
sample widths	formats, 129, 135, 152
getting and setting, 81	frame counts, 158
sampling, 69	frames, 130, 161
sampling from CD	initializing
example, 196	format, 134
sampling rates, 44	initializing tracks, 134
scheduling control, 122	instrument configurations, 130, 140, 163-167
selecting inputs and outputs, 99	defined, 129
silence	IDs, 163
cause, 91	parameters, 164

writing, 165	initializing, 74-89
loops, 130, 140	programming
ending, 167	model, 68
getting and setting, 165	outline, 73
mode, 166	purpose, 7
start, 166	Audio Utility Library
markers, 129, 159	purpose, 7
IDs, 139	Aware
names, 159	audio compression
position, 160	accessing from the AL, 733
track, 130	accessing from the CL, 736
miscellaneous chunks, 142, 168-171	software engines, 729
IDs, 141	audio compression for multimedia applications,
parameters, 168	538
reading, 170	audio libraries, 51
seeking, 170	AudioPlayback, 731
size, 169	AudioProducer, 731
types, 141, 142, 168	AudioPublisher, 731
writing, 170	AudioSuite, 732
opening, 147	compression software, 475
previewing, 60	compression software specifications, 741
reading and writing, 150-160	MPEG noise-to-mask ratio, 738
sampling rate, 152	MultiRate, 733, 734, 735, 736, 737, 738
seeking, 161	MultiRate specifications, 741
setup, 128	obtaining software licenses, 730
tracks, 130	Speed-of-Sound library, 732
defined, 129	Aware, Inc.
updating, 149	audio compression software, 61
writing, 149, 162	contacting, 61
Audio Interchange File Format with Compression. See	θ,
AIFF-C	
audio I/O, 93-96	
Audio Library, 67-??	В
ALconfigs, 69	
ALports, 68	Betacam, 255, 258
compiling, 62	Betacam SP, 258
data types, 82	binding
error handling, 72	movies to windows, 648-650
examples, 107-111, 123-??	audio considerations, 649
features, 68	multiple movies, 650
	1110111710 1110 1100, 000

blocking	callbacks
audio	CD, 191, 192-194
preventing, 92	adding, 193
blocks	removing, 194
CD, 189-190	DAT parser, 215-216
seeking, 190	capacity
brightness. See luminance	CDs, 186
broadcast video	catalog numbers
formats, 258	DAT, 204
standards, 253	CCIR 601 video standard, 255
buffered interface of the Compression Library, 492	CCITT /TSB G.711 A-law, 482
buffering interface of the Compression LIbrary, 506	CCITT/TSB G.711 mu-law, 482
buffers	<b>-cckr</b> , 63
allocating for movies	CDallowremoval(), 187
example, 628	CDatomsf(), 188, 189
audio	CD Audio Library, 183-199
size, 21	compiling, 63
image	example, 196
size, 25	features, 184
internal versus external, 507	purpose, 7
managing when adding algorithms, 568	CDbestreadsize(), 191
movies allocating, 627	CDclose(), 187
non-blocking playback, 515	CDcreateparser(), 192
non-blocking recording application, 517	CDDA_DATASIZE, 185
playback application, 514	
record application, 516	CDdeleteparser(), 194
ring, 506	CDeject(), 187
.bw images in movies, 609	CDFRAME, 184
byte ordering	CDframetomsf(), 189
DATs, 204	CDgetstatus(), 189, 195
B-Y video signal, 255	CDgettrackinfo(), 195
0 /	cdman, 60
	CDmsftoframe(), 188
	CDparseframe(), 194
С	<b>CDplay()</b> , 190
C	CDplayabs(), 191
C++ references, 40	CDplaytrack(), 190
references, 40	CDplaytrackabs(), 191

CDpreventremoval(), 187	playing
CDreadda(), 191	cdman, 60
CDremovecallback(), 194	preemphasis in AES bytes, 135
CDresetparser(), 192	preventing ejection, 187
CD-ROM	reading, 191
movies, 617	recording to DAT, 136
testing movie I/O, 591	sample rate, 185
CD-ROM drives	samples, 184
	illustrated, 185
confidence tests, 60 controlling, 187	seeking, 187-190
playing audio CDs, 190-199	blocks, 190
status, 194	tracks, 189
	status, 189
CDs	subcodes, 184
audio libraries, 51	modes, 185
blocks, 189-190	subcodeQ, 184, 185
byte ordering, 185	table of contents, 185
callbacks, 191, 192-194	time codes, 186 tracks, 186
capacity, 186 conversions	
ISRC to ASCII, 195	playing, 191
locations, 188-190	cdsample.c, 196
time codes, 195	CDsbtoa(), 195
ejecting, 187	<b>CDseek()</b> , 189
frame counts, 188	CDseekblock(), 189
frames, 184	CDseektrack, 189
ISRC, 185	<b>CDstop()</b> , 190
lead-in track, 185	CDtctoframe(), 188, 189
locations, 187-190	CDtimetoa(), 195
conversions, 188-190	
current, 189	CDtogglepause(), 190
formats, 188	CHALLENGE, 262
navigating, 187-190	changing
parser, 192-194	audio
basics, 186	global state, 107
callbacks, 193	channels
freeing, 194	audio
initializing, 192	configuring, 77
resetting, 192	defaults, 74
parsing, 192-194	audio files, 135
pausing, 190	DAT, 204

checking	clicks
audio	audio, 78
parameter ranges, 104	clients
parameters, 31	Video Library (VL), 264
chunks	clipping
AF Library, 129, 131 AIFF-C, 131	audio, 57
miscellaneous, 142	cloning
CL_BEST_FIT, 484	ALconfigs, 85
CL_GRAYSCALE, 485	clOpenCompressor(), 497
CL_MVC1	clOpenDecompressor(), 503
in an example, 499, 505	close(), 206
CL_ORIENTATION, 484	closing
CL_RGB, 485	CD-ROM device, 187
CL_RGB332, 485	movies, 636
CL_RGBA, 485	clQuery(), 515, 517
CL_RGBX, 485	clQueryAlgorithms(), 541
CL_Y, 485	clQueryBufferHdl(), 508
CL_YCbCr, 486	clQueryFree(), 509, 511
CL_YUV, 486	clQueryHandle(), 508
CL_YUV422, 486	clQueryLicense(), 543
CL_YUV422DC, 486	clQueryMaxHeaderSize(), 501
	clQueryScheme(), 501
clAddAlgorithm(), 564	clQuerySchemeFromHandle(), 542
clAddParam(), 572	clQuerySchemeFromName(), 542
clCloseCompressor(), 499	clQueryValid(), 509, 511, 515, 517
clCloseDecompressor(), 505	clReadData(), 568
clCompress(), 498, 499, 516, 518	clReadHeader(), 501
clCompressImage(), 494	clSetMax(), 567
clCreateBuf(), 501, 507	clSetMin(), 567
clDecompress(), 503, 504, 514	clSetUnique(), 566
clDecompressImage(), 496	clUpdateHead(), 509, 514, 516
clDestroyBuf, 507	clUpdateTail(, 517
clDone(, 570	clUpdateTail(), 509, 515
clDoneUpdatingHead(), 509, 515	codecs
clGetAlgorithmName(), 542	Aware, Inc., 61
clGetUnique(), 566	

codes	definition, 475
CD, 185	getting
color	movie image setting, 611
encoding	hardware acceleration, 493
illustrated, 256	image, 492, 494
sync burst, 257	JPEG
color encoding methods, ??-256	movies, 611
color space conversion, 484	movie images, 610-611
, • •	multiprocessing example, 519
formats not requiring, 487	multithreading, 506
combining	MVC1, 481, 611
movies and graphics	MVC2, 611
view size, 646	performance, 540
comments	QT_ANIM, 611
movie, 594	QT_VIDEO, 611
communicating	RLE24
DAT status, 217	RLE24 compression, 611
communications	server-client environment, 477
interprocess, 123, 124-??	Compression Library, 475
compensating	accessing Aware audio compression, 736
for CD preemphasis, 135	adding algorithms, 563
compiling	adding parameters, 572
AF Library, 62	algorithm independence, 478, 540
•	algorithm performance statistics, 540
Audio Library, 62	algorithms, 537
CD Audio Library, 63 DAT Audio Library, 63	API, 491
DM Library, 15	applications, 476
Movie Library, 589-590	buffered interface, 492, 506
•	data formats, 483
composite	error handling, 493
video	example of adding algorithms, 567
illustrated, 257	features, 478
composite video, 256	file I/O, 493
COMPRESSED_BUFFER_SIZE, 499	parameters, 543
compressed images	purpose, 8
inserting into movies, 633	sequential interface, 492, 497
compression	standards, 479
audio files, 131, 136, 156	still-frame interface, 492, 494
default, 132	compressor, 497
Aware, 475	Compuserve GIF images in movies, 609
computer versus camera images, 479	2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

computers music	controllers MIDI, 225
references, 40	controlling
concurrent	DAT drives, 206
audio, 57, 59	headphone and speaker volume, 100
concurrent audio processes	Indy audio volume, 47
querying, 107-108	conversions
	audio, 72
confidence tests, 60	input, 95
configurations	output, 96
audio	CD
cloning, 85	ISRC to ASCII, 195
audio default, 20 audio files	locations, 188-190
	time codes, 195
defaults, 132	DATs, 207, 217
default, 18	MIDI, 224
image default, 24 instrument, 130	coordinate systems
	movies, 646
configuring	copying
ALports, 74-87	movie frames, 633-635
example, 76 audio	parameters, 30
4-channel mode, 101	parameter-value lists, 30
queue size, 79	copyrights
audio files, ??-133	audio libraries
audio file tracks, 134-135	audio
MIDI, 227	
movie	copyrights, 62
audio tracks, 603	Cosmo Compress
image tracks, 606	capturing input formovies, 631
playback window, 643	counting
parameter-value lists, 18-25	ALports, 100
connections	audio file frames, 158
audio, 68, 74	parameter-value list entries, 30
	CPU resources, 57
video formats, 258	creating
	ALconfigs, 76
consuming, 506, 511	memory-mapped movies, 618
contacting	movie playback window, 641-648
Aware, inc., 61	example, 642
Prosonus 62	

movies, 617-619	DAT drives
example, 619	audio mode, 206
parameter-value lists, 17	confidence tests, 60
critical regions of memory, 123	controlling, 206
,	playing and recording DATs, 209-217 workaround, 213
D	datman, 60
D1, 255 D1 525 (YUV), 258 D1 625 (YUV), 258 D2 525 (digital NTSC), 254	DAT parser, 213-217 allocating and initializing, 214 basics, 205 freeing, 217 datplay.c, 217-220
D2 525 (digital NTSC), 254	DATs
D2 525 (NTSC), 258 D2 625 (digital PAL), 254 D2 625 (PAL), 258 daemon video, 264-265 daemons	byte ordering, 204 conversions, 217 device driver, 203, 206 example, 217-220 frame counts, 207 frames, 204
media library, 63	lead-in, 212
data	navigating, 206-209
dependencies, 123	nonaudio information, 204 parsing, 213-217
two's complement, 80 sample widths, 81	callbacks, 215-216
data formats	frames, 216
Compression Library, 483	playing, 209-210
data structures	datman, 60
AF Library, 132	playing and recording, 209-217
Audio Library, 68	reading, 211
CD Audio Library	recording, 210 examples, 213
CDFRAME, 184	recording from CD, 136
cdtimecode, 186	samples, 204
subcodeQ, 185	seeking, 208-209
DAT Audio Library DTFRAME, 204	end user locations, 207
•	statusing, 217
data types Audio Library, 82	subcodes, 204
DAT Audio Library, 203-220	subdivisions, 205
compiling, 63	time codes, 205, 207
purpose, 7	checking and setting, 211

timestamps, 211	device drivers
writing, 211	DAT, 203, 206
debugging	devices
DM Library, 16	audio, 68
Movie Library, 590-591	getting defaults, 104
Movie Library I/O, 591	CD-ROM
decompressor, 500	controlling, 187
defaults	MIDI, 225
audio, 20	SCSI, 55
channels, 74	library, 63
floating point range, 84	video
getting, 104	managing, 264
I/O rate, 59	digital
ports, 74	audio
audio files, 132	rates, 101
configuring, 18	Digital Data Storage (DDS) tapes, 212
images, 24	digital media
instruments, 133	parameter types, 14
movie	type definitions, 14
audio, 603	Digital Media (DM) Library. See DM Library
audio volume, 604	digital video
global, 597	formats, 258
images, 606	YUV (CCIR 601), 255
loop limit, 595	
loop mode, 594	digital video formats, 254
definitions	digital video recording, 254
Movie Library, 585-586	digitizing
delay	audio, 69
audio, 95	discontinuities
delays	audio, 97
Movie Library I/O, 591	cause, 91
deleting	disk space
movie frames, 630	audio, 50
movie tracks, 624	distortion
parameters, 31	audio, 107
dependencies	dm_audio.h, 15
data, 123	dm_image.h, 15
detuning, 130	DM_MEDIUM, 15
detaining, 100	
	dm_params.h, 14, 15

dmAudioFrameSize()	dmParamsSetFract(), 26
for movie buffers, 628	dmParamsSetInt(), 26
dmedia.h, 14, 15	dmParamsSetParams(), 26
dmImageFrameSize()	dmParamsSetString(), 26
for movie buffers, 628	dmParamsSetTocEntry(), 27
DM Library, 13-32	dmSetAudioDefaults, 603
assertions, 16	dmSetAudioDefaults(), 20
compiling and linking, 15	dmSetImageDefaults(), 24
debugging, 16 features, 13	drains
getting and setting parameters, 25-29	video, 267
example, 28	DTaddcallback(), 215
header files, 15	DTatohmsf(), 207
include files, 15	DTcreateparser(), 214
initializing, 16-32	DTdeleteparser(), 217
parameter-value lists, 16-32	DTframetohmsf(), 207
defined, 15	DThmsftoframe(), 207
example, 32 purpose, 6	DTparseframe(), 216
supported libraries, 13	DTpnotodec(), 217
type definitions, 14	DTremovecallback(), 216
dmParamsCopyAllElems(), 30	DTresetparser(), 214
dmParamsCopyElem(), 30	DTsbtoa(), 217
dmParamsCreate(), 17	DTtctoframe(), 207
dmParamsGetElem(), 31	DTtimetoa(), 217
dmParamsGetElemType(), 31	dynamically tuning audio applications, 97
dmParamsGetEnum(), 27	dynamicany taning addit applications, 27
dmParamsGetFloat(), 28	
dmParamsGetFract(), 28	E
dmParamsGetInt(), 27	EA HEE OF 101
dmParamsGetNumElems(), 30	EA IFF 85, 131
dmParamsGetParams(), 28	EA IFF 85 standard, 131
dmParamsGetString(), 27	editing
dmParamsGetTocEntry(), 28	movies, 626-635
dmParamsIsPresent(), 31	ejecting CDs, 187
dmParamsRemoveElem(), 31	embedded movies, 617-621
dmParamsSetEnum(), 26	opening, 617 seeking, 617
dmParamsSetFloat(), 26	, o.r.

emphasis audio files, 135	F
emulating Movie Library I/O errors, 591	failures movie
encoding video illustrated, 256 end users	emulating, 591 features ALports, 74 Audio Library, 68
CD control, 188 environment variables Movie Library, 591	CD Audio Library, 184 DM Library, 13 Indigo audio, 43-44
equations YIQ, 256 YUV, 255	Movie Library, 579 VL, 8 features of the Compression Library, 478
error handlers audio, 73	fields video, 251
error handling Audio Library, 72 Compression Library, 493	file descriptors audio, 119
errors AF Library, 128	getting, 120 movies, 617-621
allocating audio configurations, 77 audio channels, 77 audio I/O, 97-98 types, 98	file formats AF Library, 134 AIFF, 131 audio, 129 audio files
event handling MIDI, ??-246	default, 132 initializing, 134 EA IFF 85, 131
events Video Library (VL), 261	movie, 587 getting, 596 Silicon Graphics, 587
events, handling GL events, 462	SiliconGraphics, 587 file I/O
examples audio, 61	Movie Library, 615-626 illustrated, 616
external buffer, 507	overview, 616 file I/O in the Compression Library, 493
	files access mode for movies, 617 AIFF-C, 129-131

formats	frame counts
AF Library, 127	audio files, 158
fillable, 92	CD, 188
filled, 92	DATs, 207
fill points	frame rate
audio, 120	defined, 586
illustrated, 121	video, 251
finding	frames
movie tracks, 624	audio, 70
FIT images in movies, 609	illustrated, 70
<i>a</i>	audio file, 130
AES channel status bytes 126	audio files, 161
AES channel status bytes, 136	CD, 184
POLLIN, 120 POLLOUT, 120	DAT, 204
	deleting from movies, 630
floating point	illustrated, 586
audio formats, 80	numbering, 586
floating point range	video, 251
getting and setting, 83-85	freeing
flushing	AFfilesetup, 133
movie edits, 636	ALconfigs, 85
formats	CD parser, 194
audio, 71	DAT parser, 217
default, 74	parameter-value lists, 18
floating point, 80	full-scale audio ranges, 81
getting and setting, 80	
native, 71	G
two's complement, 80 broadcast video, 258	9
CD	gain, 130
locations, 188	Galileo Video, 7
consumer video, 258	
digital video, 258	getting
movies, 587	file descriptors audio, 120
parameter-value lists, 15	movie
video, 253	audio track properties, 603-606
videotape, 257-258	audio channels, 605
frame count	
defined, 586	audio compression, 606
	audio format, 605

audio sample width, 604 default volume, 604 background color, 644 file format, 596 general track properties, 601-602 length, 602 medium, 602 SMPTE time code string, 601 global properties, 593-598 file format, 596 loop mode, 595 optimization, 596 title, 595 image track properties, 606-611 frame rate, 607 image format, 609 image orientation, 608	audio sample rate, 604	grayscale, 538
default volume, 604 background color, 644 file format, 596 general track properties, 601-602 length, 602 medium, 602 SMPTE time code string, 601 global properties, 593-598 file format, 596 loop limit, 595 loop mode, 595 optimization, 596 title, 595 image track properties, 606-611 frame rate, 607 image format, 609 image orientation, 608 image packing format, 609  H  AH  H  H  H  H  H  H  H  H  H  H  H		guaranteeing movie view size, 645
background color, 644 file format, 596 general track properties, 601-602 length, 602 medium, 602 SMPTE time code string, 601 global properties, 593-598 file format, 596 loop limit, 595 loop mode, 595 optimization, 596 title, 595 image track properties, 606-611 frame rate, 607 image format, 609 image orientation, 608 image orientation, 608 image packing format, 609  H  handles AFfilehandle, 132 ALconfigs, 76 CDFRAME, 184 DAT frame, 204 parameter-value lists, 18 hardware Indigo audio, 43-45 hardware acceleration compression, 493 Head, 506 header reading, 501 structure, 502 header files	*	
file format, 596 general track properties, 601-602 length, 602 medium, 602 SMPTE time code string, 601 global properties, 593-598 file format, 596 loop limit, 595 loop mode, 595 optimization, 596 title, 595 image track properties, 606-611 frame rate, 607 image format, 609 image orientation, 608 image orientation, 608 image packing format, 609  handles AFfilehandle, 132 ALconfigs, 76 CDFRAME, 184 DAT frame, 204 parameter-value lists, 18 hardware Indigo audio, 43-45 hardware acceleration compression, 493 Head, 506 header reading, 501 structure, 502 header files		
length, 602 medium, 602 SMPTE time code string, 601 global properties, 593-598 file format, 596 loop limit, 595 loop mode, 595 optimization, 596 title, 595 image track properties, 606-611 frame rate, 607 image format, 609 image orientation, 608 image orientation, 608 image packing format, 609  AFfilehandle, 132 ALconfigs, 76 CDFRAME, 184 DAT frame, 204 parameter-value lists, 18 hardware Indigo audio, 43-45 hardware acceleration compression, 493 Head, 506 header reading, 501 structure, 502 header files		Н
medium, 602  SMPTE time code string, 601  GDFRAME, 184  global properties, 593-598  file format, 596  loop limit, 595  loop mode, 595  optimization, 596  title, 595  image track properties, 606-611  frame rate, 607  image format, 609  image orientation, 608  image orientation, 608  image packing format, 609  AFfilehandle, 132  ALconfigs, 76  CDFRAME, 184  DAT frame, 204  parameter-value lists, 18  hardware  Indigo  audio, 43-45  hardware acceleration  compression, 493  Head, 506  header  reading, 501  structure, 502  header files	general track properties, 601-602	handles
SMPTE time code string, 601  global properties, 593-598  file format, 596  loop limit, 595  loop mode, 595  optimization, 596  title, 595  image track properties, 606-611  frame rate, 607  image format, 609  image orientation, 608  image packing format, 609  ALconfigs, 76  CDFRAME, 184  DAT frame, 204  parameter-value lists, 18  hardware  loop audio, 43-45  hardware acceleration  compression, 493  Head, 506  header  reading, 501  structure, 502  header files	length, 602	
SMPTE time code string, 601  global properties, 593-598  file format, 596  loop limit, 595  loop mode, 595  optimization, 596  title, 595  image track properties, 606-611  frame rate, 607  image format, 609  image orientation, 608  image packing format, 609  CDFRAME, 184  DAT frame, 204  parameter-value lists, 18  hardware  ludigo  audio, 43-45  hardware acceleration  compression, 493  Head, 506  header  reading, 501  structure, 502  header files	medium, 602	
file format, 596     loop limit, 595     loop mode, 595     optimization, 596     title, 595     image track properties, 606-611     frame rate, 607     image format, 609     image orientation, 608     image packing format, 609	SMPTE time code string, 601	
loop limit, 595 loop mode, 43-45 loop mode, 43-45 loop mode, 43-45 loop mode, 43-45 loop mode, 595 loop mode, 595 loop mode, 595 loop mode, 595 loop mode, 43-45 loop mode, 43-45 loop mode, 43-45 loop mode, 43-45 loop mode, 595 loop mode, 43-45 loop mode, 493 loop mode, 43-45 loop mo	global properties, 593-598	
loop mode, 595 optimization, 596 title, 595 hardware acceleration compression, 493  frame rate, 607 image format, 609 image orientation, 608 image packing format, 609  Indigo audio, 43-45 hardware acceleration compression, 493 Head, 506 header reading, 501 structure, 502 header files	file format, 596	parameter-value lists, 18
optimization, 596  title, 595  image track properties, 606-611  frame rate, 607  image format, 609  image height, 608  image orientation, 608  image packing format, 609  audio, 43-45  hardware acceleration  compression, 493  Head, 506  header  reading, 501  structure, 502  header files	loop limit, 595	
title, 595 image track properties, 606-611 frame rate, 607 image format, 609 image height, 608 image orientation, 608 image packing format, 609  hardware acceleration compression, 493 Head, 506 header reading, 501 structure, 502 header files	loop mode, 595	· · · · · · · · · · · · · · · · · · ·
image track properties, 606-611  frame rate, 607  image format, 609  image height, 608  image orientation, 608  image packing format, 609  compression, 493  Head, 506  header  reading, 501  structure, 502  header files	optimization, 596	
frame rate, 607  image format, 609  image height, 608  image orientation, 608  image packing format, 609  Head, 506  header  reading, 501  structure, 502  header files	title, 595	
image format, 609 image height, 608 image orientation, 608 image packing format, 609 header reading, 501 structure, 502 header files	image track properties, 606-611	-
image height, 608  image orientation, 608  image packing format, 609  reading, 501  structure, 502  header files	frame rate, 607	
image orientation, 608 image packing format, 609 image packing format, 609	image format, 609	
image orientation, 608 header files	image height, 608	
imago packing tormat 600	image orientation, 608	
dm narams h 14	image packing format, 609	dm_params.h, 14
image width, 608 dmedia.h, 14	image width, 608	•
parameters, 591-598 DM Library, 15		
view size, 646 Video Library (VL), 266		Video Library (VL), 266
parameters, 27 headphones	•	headphones
name, 31 controlling volume, 100		controlling volume, 100
total, 30 muting, 100 type, 31		_
GIF images in movies, 609  H: 9 mm 259	7.1	hertz (Hz), 69
GL Hi-8mm, 258		Hi-8mm, 258
events handling 462 hints		
global audio device state, 98 multiple movie playback, 640	e e	multiple movie playback, 640
global audio parameters, 99	~	
global audio state, 59	•	1
changing, 107	S .	•
global movie properties, 593 icons	8 8	icons
Graphics Library, recommended reading, xl audio, 44		audio, 44

IEEE	jacks, 44
double-precision floating point data, 80	queue size limits, 79
image	MIDI ports, 227
compression, 494	Indigo <sup>2</sup>
image compression, 492	audio
images	4-channel, 46
buffer size, 25	features, 46
compressed	hardware, 46-47
editing in movies, 631-633	jacks, 47
inserting into movies, 633	queue size limits, 79
compression in movies, 610-611	MIDI ports, 228
defaults, 24	Indigo <sup>2</sup> Video, 7
FIT in movies, 609	IndigoVideo Library
format for movies, 609	purpose, 8
formats	Indy
SGI, 609	audio
getting	4-channel, 46
compression for movies, 611	features, 46-50
height for movies, 608	hardware, 46-50
inserting into movies, 629	jacks, 48
Kodak Photo CD in movies, 609	queue size limits, 79
movie	volume buttons, 48
defaults, 606	MIDI ports, 228
movie frame rate, 607	muting, 47
orientation for movies, 608	Indy Video, 7
packing format for movies, 609-610	initializing
parameters, 22	Audio Library, 74-89
properties in movies, 606	CD parser, 192
TIFF in movies, 609	DM Library, 16-32
width for movies, 608	MIDI, 229
ImageVision Library	input
reading and writing movie images, 609	audio, 94-95
include files	4-channel, 94
DM Library, 15	conversions, 95
indexes	source, 99
CD tracks, 186	inserting
Indigo	movie data, 629
audio	instances
features, 43-44	movie, 587
hardware, 43-45	1110 (10, 00)

instrument configurations, 163-167	IRIX kernel, 57
audio files, 130	
default, 132	
defined, 129	J
parameters, 133	
instruments	jacks
samples, 62	AES
interlacing	Indigo
video, 251	audio, 44 Indigo <sup>2</sup>
interleaving	audio, 46, 47
audio, 70	
internal buffer, 507	JPEG, 479, 496 movies, 611
internal format, 484	movies, off
International Standard Recording Code (ISRC), 185	
converting to ASCII, 195	K
interprocess communication, 123	
example, 124-??	keyframes, 484
intializing	defined, 585
audio files, 128	key velocity, 130
I/O	Kodak Photo CD images in movies, 609
audio, 93-96	Ŭ
error parameters, 98	
errors, 97-98	L
measuring and matching sample rates, 99	
multiplexing, 119	–laudio, 62
nonblocking, 92	–laudiofile, 62
querying rates, 108-110	-laudioutil, 62
MIDI, 232	Launchword
Movie Library	/usr/sbin/jot
testing, 591	/usr/people/4Dgifts/examples/dmedia/aud
ioctl(), 206, 208	io/ratequery.c, 108
IRIS Digital Media Development Environment, 5-9	-lcdaudio, 63
IRIS GL	-ldataudio, 63
screen origin, 646	-ldmedia, 15
IRIS Media Libraries	-ldmedia_d, 16
digital audio libraries, 39	lead-in
IRIX	DATs, 212
file system, 55	libaudiofile.so. See AF Library
real-time programming, 119	

libaudioutil.a, 63	getting, 104
libcdaudio. See CD Audio Library	queue size, 79
libcl. See Compression Library	CD tracks, 186
libdataudio. See DAT Audio Library	linear pulse code modulation (PCM), 71
libdmedia.so. SeeDM Library	linking
libds.a, 63	Movie Library, 589-590
libm.a, 63	debugging version, 590
libmalloc.a, 63	non-ANSI-compliant code, 63
-libmediad, 63	-lm, 62
	locating
libmidi. See MIDI Library	movie tracks, 624
libmovie_d.a, 590	locations
libmovie. See Movie Library	CD, 187-190
libraries	current, 189
Audio File Library, 7	formats, 188
Audio Library, 7, 67-??	locking
Audio Utility Library, 7	arenas, 123
CD audio, 51	CD-ROM eject, 187
CD Audio Library, 7	looping
Compression Library, 8	modes
DAT Audio Library, 7, 203-220	defined, 586
DM Library, 6, 13-32	movies, 586
IndigoVideo Library, 8	loops
math, 63	audio files, 130
MIDI, 6, 223-??	default, 132
Movie Library, 9	default
Prosonus, 62	limit, 595
Video Library, 7	mode, 594
Video Library (VL), 261	lossless
license	definition, 479
installing, 743	lossy
nodelocked, 743	definition, 479
querying for, 543	lseek(2), 617
licensing Aware compression software, 730	luminance, 255
limiting	iummance, 200
audio, 57	
limits	
andio	

floating point range, 83-85

M	resolution, 71
M(agnetic) T(ape) I/O C(ontrol) T(ape) OP(eration).	stereo, 46, 101
See MTIOCTOP	MIDI, 227
malloc(), 542	C++ references, 40
malloc(), 63	configuring, 227
for movie buffers, 628	controllers, 225
managing	conversions, 224
video devices, 264	devices, 225
mapping	handling events, ??-246
movie tracks, 624-626	I/O, 232
illustrated, 625	patchbays, 224 peripherals, 224
markers	ports, 232
audio files, 129, 159	references, 40
default, 132	serial ports
loops, 130	Indigo <sup>2</sup> , 228
names, 159	Indy, 228
position, 160	starting and stopping, 229
audio file tracks, 130	timestamping, 236
matching	timestamps, 231
audio I/O rates, 99	MIDI Library, 6, 223-??
math library, 63	event handling, ??-246
measuring	MII video, 258
audio	miscellaneous chunks, 168-171
digital I/O rates, 101 sample rates, 99	modes, 101
	DAT drive, 206
media type definitions 14	microphone, 101
type definitions, 14 types, 15	monitor
media library daemon, 63	Silicon Graphics, 252
•	monitoring
audio requirements, 50	audio, 100
critical regions, 123	Motif, recommended reading, xli
memory-mapped movies	movie editing, 476
creating, 618	movie frames
opening, 622	deleting, 630
memory requirements	Movie Library
digital audio, 50	adding user parameters, 598
memtovid. 265	example, 599, 600

applications, 580	example, 619
assertions, 590	defaults, 597
compiling and linking, 589-590	audio, 603
debugging version, 590	audio volume, 604
debugging, 590-591	image, 606
definitions, 585-586	loop limit, 595
environment variables, 591	loop mode, 594
features, 579	defined, 585
file formats, 587	editing, 626-635
file I/O, 615-626	compressed images, 631-633
illustrated, 616	embedded, 617-621
overview, 616	file access mode, 617
programming guidelines, 588-589	finding tracks, 624
programming model, 587	FIT images, 609
purpose, 9	flushing edits, 636
Movie Maker	formats, 587
file formats, 587	frame rate, 607
movies	getting
adding	image compression, 611
tracks, 622-623	global properties, 593
adding tracks	illustrated, 586
example, 623	image compression, 610-611
allocating buffers, 627	image format, 609
example, 628	image height, 608
audio	image orientation, 608
channels, 605	image packing format, 609-610
compression, 606	image width, 608
formats, 605	inserting
native formats, 605	compressed images, 633
ports, 649	track data, 629
sample rate, 604	instances, 587
sample width, 604	JPEG, 611
background color, 644	keyframes, 585
binding to windows, 648-650	looping
multiple movies, 650	defined, 586
capturing input from Cosmo Compress, 631	loop modes
closing, 636	defined, 586
comments, 594	mapping tracks, 624-626
copying and pasting, 633-635	illustrated, 625
illustrated, 633	multiple movie playback hint, 640
creating, 617-619	

opening, 620	view
from file descriptors, 617-621	offset, 646-648
from filenames	size, 644-646
filenames	getting, 646
for opening movies, 621	guaranteeing, 645
from memory, 622	write-only, 617
optimization	zooming, 645
getting, 596	MPEG, 480, 538
parameters	Aware noise-to-mask ratio, 738
setting and getting, 591	MTIOCTOP, 206
PCD images, 609	multimedia applications, choosing a compression
playback	method, 538
controlling, 651	
playback window, 641-648	multiplexing
configuration, 643	synchronous audio I/O, 119
example, 642	multiprocessing compression, 506
playing, 639-678	example, 519
outlined, 639	MultiRate
properties, 591	Aware, Inc., 61
QT_ANIM compression, 611	MultiRate Aware compression algorithm, 733, 734,
QT_VIDEO compression, 611	735, 736, 737, 738
reading and writing from ImageVision Library,	specifications, 741
609	music, 62
reading compressed images, 632	references, 40
reading frames	music-quality audio, 70
storing	muting
movie frames in a buffer, 630	headphones and speakers, 100
read-only, 617	Indy, 47
read-write, 617	mvAddTrack(), 622
removing tracks, 624	mvAddUserParam(), 598
seeking, 617	
SGI image formats, 609 Silicon Craphics format defined, 587	mvBindWindow(), 649
Silicon Graphics format defined, 587 SMPTE time codes, 601-602	MVC1, 481, 538, 539
TIFF images, 609	MVC1 compression, 481, 611
titles, 595	MVC2 compression, 611
track operations, 622	mvClose(), 636
uncompressed, 611	mvCreateFD(), 618
verifying, 619	mvCreateFile(), 618
<i>y G</i>	myCreateMem(), 618

mvDeleteFrames, 630 when pasting movie frames, 635 mvFindTrackByMedium(), 624 mvOpenFD(), 620, 621 mvGetAudioChannels(), 605 mvOpenFile(), 620, 621 mvGetAudioCompression(), 606 mvOpenMem(), 620, 621 mvGetAudioFormat(), 605 mvOptimize(), 627 mvGetAudioRate(), 604 mvPasteFrames, 635 mvGetAudioWidth(), 604 mvPlay(), 651 mvGetCompressedImageSize, 631 mvQueryViewOffset, 647 mvGetDefaultVol(), 604 mvQueryViewSize(), 645 mvGetFileFormat. 596 mvReadCompressedImage, 631 mvGetImageCompression(), 611 mvReadFrames(), 630 mvGetImageFormat(), 609 mvRemoveTrack(), 624 mvGetImageHeight(), 608 mvSetDefaultVol(), 604 mvGetImageOrientation(), 608 mvSetImageRate(), 607 mvGetImagePacking(), 609-610 mvSetLoopLimit(), 595 mvGetImageRate, 608 mvSetLoopMode(), 594 mvGetImageWidth(), 608 mvSetMovieDefaults(), 596 mvGetLoopLimit(), 595 mvSetNumMoviesHint(), 640 mvGetLoopMode(), 595 mvSetParams(), 592 for setting user parameters, 598 mvGetNumMoviesHint(), 640 mvSetSMPTEStart(), 602 mvGetOptimized(), 596 mvSetTitle(), 595 mvGetParams(), 592 mvSetViewBackground(), 644 mvGetSMPTEStart(), 602 mvSetViewOffset, 647 mvGetTitle(), 595 mvSetViewSize(), 645 mvGetTrackLength(), 602 mvStop(), 651 mvGetTrackMedium(), 602 mvWrite, 636 mvGetViewBackground(), 644-?? mvGetViewOffset(), 648 mvGetViewSize(), 646 N mvInsertCompressedImage(), 632 mvInsertFrames, 629 names mvIsMovieFD(), 619 audio ports, 88 mvIsMovieFile(), 619 parameters, 31 navigating mvIsMovieMem(), 619 CDs, 187-190 mvMapBetweenTracks, 624-625, 634 DATs, 206-209

NetLS, 743	output
nodelocked licenses, 743	audio, 95-96
nodelock file, 743	conversions, 96
noise-to-mask ratio	overflow
Aware MPEG, 738	audio, 93
non-ANSI-compliant code, 63	
nonblocking	Р
audio I/O, 92	
NTSC, 253-254, 256	PAL, 253-254, 256
digital recording, 254	digital recording, 254
illustrated, 252	illustrated, 252
resolution, 253	resolution, 253
YIQ encoding, 255	YUV, 255
Nyquist Theorem, 69	parameters, 132
	adding to the Compression Library, 572
	audio, 19
0	current value, 106
	getting and setting, 102-107
O_RDONLY, 617	I/O errors, 98
O_RDWR, 617	names, 105
O_WRONLY, 617	setting, 106
offsets	system-dependent, 101
movie view, 646	audio core global, 99
open(), 206	audio tracks, 134
open(2), 617	checking, 31
•	checking bounds, 104
opening	Compression Library, 543
memory-mapped movies, 622	copying from parameter-value lists, 30
movies, 620	deleting, 31
optimization	getting
movie	type, 31
getting, 596	getting and setting, 27
origin	images, 22 instruments, 133
screen	movie
IRIS GL, 646	adding, 598, 599, 600
X Window System, 646	setting and getting, 591-598
original format, 484	names, 31
oserror(), 72	removing, 31
	icinoving, or

parameter-value buffer	paths
audio, 102	video
parameter-value lists	blending, 268
configuring, 18-25	VL
audio, 20	defined, 267
image, 24	illustrated, 267
copying, 30	pausing
creating and destroying, 17-18	CDs, 190
example, 18	PCD images in movies, 609
defined, 15	performance tuning
destroying, 18	audio, 119-??
DM, 16-32	
example, 32	peripherals
formats, 15	audio, 51
getting and setting values, 25-29	MIDI, 224
number of elements, 30	playback
removing parameters, 31	movies
parser	controlling, 651
CD	non-blocking, 515
basics, 186	playing
callbacks, 193	CDs, 190-194
frames, 194	tracks, 191
freeing, 194	DATs, 209-210
initializing, 192	example, 217-220
DAT, 213-217	movies, 639-678
allocating and initializing, 214	outlined, 639
basics, 205	multiple movies in one window, 650
freeing, 217	poll(), 119, 120
	POLLIN, 120
parsing CD <sub>2</sub> 102 104	
CDs, 192-194	polling
callbacks, 193	example, 124-??
frames, 194	POLLOUT, 120
DATs, 213-217	pops
frames, 216	audio, 78
pasting	ports
movie frames, 633-635	audio, 74-89
illustrated, 633	allocating and initializing, 88
patchbays	closing and deallocating, 88
MIDI, 224	configuring, 74-87
	defaults, 74

defined, 68	global, 593
example, 76	Prosonus
formats, 80	audio libraries, 51
names, 88	contacting, 62
opening and closing, 87-89	sound libraries, 62
example, 89	
static settings, 75	
MIDI, 232	Q
postproduction	
video formats, 258	qdevice(), 462
prctl(), 122	in sample program, 463
precision	QT_ANIM compression, 611
audio, 81	QT_VIDEO compression, 611
previewing	quantization steps
audio files, 60	audio, 84
prioritizing	querying
audio, 122	audio
priority	4-channel, 111
audio, 57	concurrent processes, 107-108
process controls	I/O rates, 108-110
audio, 122	supported features, 99
processes	querying algorithms, 540
audio	querying for a license, 543
concurrent, 107-108	queues
producing, 506, 511	audio, 90-93
professional	defaults, 74
video formats, 258	illustrated, 91
programming	size, 78-80, 92
guidelines	size limits, 79
audio, 59	status, 92
Movie Library, 588-589	thresholds, 120
models	
Audio Library, 68	R
Movie Library, 587	·
outlines	ranges
Audio Library, 73	audio
video hardware, 7	floating point, 83-85
properties	full-scale, 81
movie, 591	

read(), 211 reading audio data, 94-95 audio from CDs, 191 compressed movie images, 632 DATs, 211 read-only movies, 617 read-write movies, 617 real-time programming audio, 119-?? example, 124-??	8-bit, 387 .rgba images in movies, 609 .rgb images in movies, 609 ring buffer, 506 ring buffers, 506 RLE, 481 RTR1, 482, 538, 539 run-length encoding movies, 611 R-Y video signal, 255
recordexample.c, 177 recording DATs, 210 examples, 213	<b>S</b> samplers
digital video, 254 using buffers for non-blocking compression, 517 using buffers to compress for, 516 reel-to-reel (Type C) videotape, 258	audio, 130 samples CD, 184 illustrated, 185
references audio, 40 MIDI, 40 MIDI C++, 40 release loops	DAT, 204 instruments, 62 sample widths audio
release loops audio files, 130	default, 74 getting and setting, 81
removing CD callbacks, 194 movie frames, 630 movie tracks, 624 parameters, 31	sampling audio, 69 sampling rates audio, 44, 69 measuring, 99
resetting CD parser, 192	audio files, 152 CD, 185
resolution video, 253	DAT, 204 scaling
resolutions AES, 71 audio, 71, 82 RGB, 254 32-bit, 387	audio floating point ranges, 83 schedctl(), 122 scheduling audio, 122

screen	movie
origin	audio track properties, 603-606
IRIS GL, 646	default volume, 604
X Window System, 646	audio track properties defaults, 603
SCSI devices, 55	general track properties, 601-602
library, 63	SMPTE time code string, 601
SECAM, 253	global properties, 593-598
seeking	loop limit, 595
audio files, 161	loop mode, 594
miscellaneous chunks, 170	title, 595
CD, 187-190	image track properties, 606-611
blocks, 190	defaults, 606
defined, 186	frame rate, 607
tracks, 189	parameters, 591-??
DATs, 208-209	view size, 645
movies, 617	multiple movie playback hint, 640
select(), 119	parameters, 27
audio	by copying, 30
illustrated, 121	.sgi images in movies, 609
for multiplexing audio I/O, 119	shared
selecting	arenas, 123
audio I/O sources, 99	example, 124-??
semaphores, 123	audio resources, 57
example, 124-??	illustrated, 57
sequential interface of the Compression Library, 492	GL, 63
serial ports	process synchronization, 123
Indigo, 227	signal
server-client environment, CL, 477	composite video
setting	illustrated, 257
audio	signals
parameters, 106	video, 255
audio defaults, 20	Silicon Graphics
example, 21	noninterlaced monitor, 252
audio file format, 134	Sirius Video, 7, 262
audio file parameters, 134	sizing
audio fill points, 120	audio
image defaults, 24	buffers, 21
example, 25	queues, 78-80

images	stereo
buffers, 25	audio frames
SMPTE time codes	illustrated, 70
movies, 601-602	microphone, 46
soundeditor, 57, 58, 60	microphones, 101
sound effects, 62	stereo frames
soundfiler, 57, 58, 60	CD, 184
soundtracks	storing
finding, 624	compressed movie frames, 632
removing, 624	structures
sources	AF Library, 132
video, 267	CDFRAME, 184
speakers	subcodeQ, 184
controlling volume, 100	CD, 185
muting, 100	subcodes
specifications	CD, 184
AES3-1985, 136	modes, 185
audio hardware, 723, 726	DAT, 204
Aware audio compression software, 741	subdivisions
EA IFF 85, 131	CD tracks, 186
spinlocks, 123	DAT, 205
sproc(), 518, 519	sustain loops
sproc()	audio files, 130
example, 124-??	SV_IN_REPLACE
standards	in sample program, 463
AES3-1985, 136	SvActiveAttribute
CCIR 601, 255	in sample program, 463
compression, 479	svBindGLWindow()
video broadcast, 253	in sample program, 463
state	S-VHS, 256, 258
audio, 59	S-Video, 256, 258
status	SvVideoPreempted
CD-ROM drives, 194	in sample program, 463
CDs, 189	switching
statusing	4-channel audio, 46
DATs, 217	sync burst, 257
stderr, 72	

synchronizing	timestamping
movie tracks, 624-626	MIDI, 236
illustrated, 625	timestamps
system calls	DATs, 211
IRIX, 119	MIDI, 231
systems	titles
audio software, 55	movie, 595
	tools
<b>T</b>	audio, 60
Т	Video Library (VL), 265
table of soutoute	tracks
table of contents	adding, 622-623
CD, 185 DAT, 204	example, 623
	audio
Tail, 506	properties, 603
tape formats, ??-258	audio file, 130
tapes	defined, 129
DDS, 212	initializing, 134-135
tasks	audio files, 162
AF Library, 127	channels, 135
telecommunications	default, 132 CD, 186
choosing a compression method, 538	playing, 191
teleconferencing, 8, 476	seeking, 189
testing	subdivisions, 186
Movie Library I/O, 591	CD lead-in, 185
third-party audio software, 61-62	defined, 585
thresholds	finding, 624
audio	general properties, 601
ports, 120	length
TIFF images in movies, 609	getting, 602
time	mapping, 624-626
	illustrated, 625
required for audio hardware to play samples, 95	medium
time codes	getting, 602
CD, 186	operations, 622
conversions, 195 DATs, 205, 207	removing, 624
checking and setting, 211	setting and getting
checking and setting, 211	general properties, 601-602

trapping	V
audio I/O errors, 98	
troubleshooting	vcp, 265
audio	verifying
configurations, 77	movies, 619
distortion, 107	VHS, 258
I/O, 97-98	video
overflow and underflow, 93	Betacam, 255
pops and clicks, 78	broadcast standards, 253
Movie Library I/O, 591	B-Y, 255
	composite, 256
tuning	illustrated, 257
audio applications dynamically, 97	D1, 255
audio performance, 119-??	daemon, 264-265
two's complement data, 80	device management, 264
sample widths, 81	digital recording, 254
Type B video format, 258	drains, 267
Type C (reel-to-reel) videotape, 258	encoding, 254-??
types	illustrated, 256 RGB, 254
digital media parameters, 14	fields, 251
media, 14, 15	formats, 253
parameters	illustrated, 256
getting, 31	frame rate, 251
	frames, 251
U	hardware, 7
U	interlacing, 251
U-Matic (SP), 258	luminance, 255
	NTSC
uncompressed movies, 611	illustrated, 252
underflow	PAL
audio, 93, 98	illustrated, 252
user interface, xli	paths
usinit(), 123	blending, 268
uspsema(), 123	resolution, 253
usvsema(), 123	R-Y, 255
U-V signal. See chrominance	sources, 267 S-Video, 256
	sync burst, 257
	teleconferencing, 8
	tereconnectencing, o

YIQ, 255 YUV, 255 VIDEO, GL pseudo device in sample program, 463 videod, 264-?? video data formats, 386 video formats and color encoding methods, 256 and tape formats, 258 videoin, 265 Video Library	VL features, 8 vlBeginTransfer(), 296 vlCreatePath(), 274 vlEndTransfer(), 302 vlinfo, 265 vlNextEvent(), 350 vlOpenVideo(), 271 vlSelectEvents(), 277 vlSetvnPaths(), 276
purpose, 7 Video Library (VL) device management, 264 events, 261 features, 261 header files, 266	vlSetupPaths(), 276 voice-quality audio, 69 volume Indy, 47
multiple clients, 264 paths defined, 267 system software architecture, 263 tools, 265 videoout, 265 videopanel, 265 videotape formats, 257-258 video/voice mail, 476 vidtomem, 265 viewing size movie, 644-?? views movie	windows movie playback configuring, 643 example, 642 workarounds DAT drives, 213 wrap, 511 write-only movies, 617 writing audio files, 162 audio samples, 95-96 DATs, 211 movie data. See inserting movie data
getting size, 646 guaranteeing, 645 offset, 646-648 size, ??-646	X X11, recommended reading, xl X Window System coordinates, 646

## Υ

YC, 256 YC-358, 256 YC-443, 256 YIQ, 255 equations, 256 Y signal. *See* luminance YUV, 255, 388 digital, 255 equation, 255

## Z

zooming movies, 645