OCTANE™ Digital Video Programmer's Guide

Document Number 007-3513-001

CONTRIBUTORS

Written by Carolyn Curtis
Illustrated by Cheri Brown, Scott Pritchett, and Carolyn Curtis
Document Production by Kirsten Johnson
Engineering contributions by Judy Ting, Rick Davis, Matthew Hall, Howard Chartock, Chris Pirazzi, Grant Dorman, Girish Goyal, Bruno Wolf, Michael Minakami, Tony Masterson, and Scott Pritchett
St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

OCTANE™ Digital Video Programmer's Guide
Document Number 007-3513-001

Table of Contents

List of Figures ix

List of Tables xi

About This Guide xiii

Audience xiii

Structure of This Document xiii

Other Documents xv

Conventions xv

1. Features of the OCTANE Digital Video Option 1

OCTANE Digital Video Board Capabilities 1

Video Library Capabilities 3

VL System Software Architecture 4

- Video Daemon 5
- Generic Video Tools 6
- Library and Header Files 7

VL Architectural Model of Video Devices 8

- Node 9
- Path 9
- Port 10

OCTANE Digital Video Formats 12

2. Creating Video Programs With the Video Library 13
The VL Programming Model 13
Performing Preliminary Steps 15
Opening a Connection to the Video Daemon 15
Specifying Nodes on the Data Path 15
Creating and Setting Up the Data Path 17
  Creating the Path 17
  Getting the Device ID 18
  Adding a Node 18
  Setting Up the Data Path 18
  Specifying the Path-Related Events to Be Captured 20
Setting Parameters for Data Transfer to or From Memory 22
  Synchronizing Data Streams 37
Displaying Video Data Onscreen 41
Transferring Video Data to and From Devices 43
  Using Buffers 43
  Transferring Video Data Using DMbuffers 46
  Transferring Video Data Using VL Buffers 52
Ending Data Transfer 58
Example Programs 60

3. Using VL Controls 61
VL Control Type and Values 64
VL Control Fraction Ranges 65
VL Control Classes 65
VL Control Groupings 66

4. Event Handling 69
OCTANE Digital Video VL Events 70
Querying VL Events 75
Creating a VL Event Loop 77
Creating a Main Loop With Callbacks 78

5. Managing Connections 81

Specifying Connectivity 81

- Getting Connections 84

Avoiding Dynamic Switching Problems 89

- Crosspoint Mux Timing Restrictions 90
- VGI1 Memory Source Timing Restrictions 91

Using the Internal Video Sync Signal 91

- Internal Video Sync Producers and Consumers 92
- Setting the Internal Video Sync Signal Producer 93

6. Video Real-Time Capture and Playback 97

Video Library Buffers 97

Caching 98

Buffer Alignment 99

Direct I/O to Disk 99

syssgi 100

Asynchronous I/O 101

Capture and Playback Examples 102

7. Blending, Keying, and Transitions 103

The Blender Node 104

- Setting Up the Blender Node 106
- Setting Normalization 108
- Setting and Turning Off Flat Background 111
- Adding Shadows 111

Keying 112

- Luma Keying 114
- Chroma Keying 116
- Fades, Tiles, and Wipes 119

The Keyer 122

VL Blending Examples 123

- Blending Video and Graphics 123
- Creating a Simple Wipe Effect 124

8. Using Color-Space Conversion 125
Features of the Color-Space Conversion Node 126
Performing Standard Color-Space Conversions 128
Color-Space Conversion Packings 129
Color-Space Conversion Ranges 130
Constant Hue 132
VL_CSC Ports 133
Full-Range and Limited-Range Video 134
Specifying Standard Color-Space Conversion Node Controls 135
Using the Color-Space Converter for Image Processing 136
Using Custom LUTs and Matrix Multiplier Coefficients 136
Using Image-Processing Controls 137
Examples 140

9. Using Video Texture Mapping 147
Performing Video Texture Mapping 148
Real-Time Mipmap Generation 150
Timing Issues 151
Controls for Video Texture Mapping 153
VL_CAP_TYPE 153
VL_OFFSET 153
VL_PACKING 154
VL_RATE 154
VL_SIZE 155
VL_ZOOM 155
VL_MGV_DOMINANCE_FIELD 157
VL_MGV_HASPECT 157
VL_MGV_VASPECT 157
VL_MGV_TEXTURE_ROUND_MODE 158
VL_MGV_TEXTURE_MIPMAP_MODE 158
VL_MGV_TEXTURE_INPUT_LINK 158
VL_MGV_TEXTURE_AUTOSWAP 159
OpenGL Functions for Video Texture Mapping 159
Example Program: vidtotex.c 161

A. Return Codes 163

B. OCTANE Digital Video Nodes and Their Controls 169

VL_DEVICE 171

VL_BLENDER 175

VL_CSC 179

VL_FB 181

VL_MEM 183

- VL_MEM Source 183
- VL_MEM Drain 189

VL_SCREEN 194

- VL_SCREEN Source 194
- VL_SCREEN Drain 196

VL_TEXTURE 199

VL_VIDEO 202

- VL_VIDEO Source 202
- VL_VIDEO Drain 204

C. OCTANE Digital Video Color-Space Conversions 209

OCTANE Digital Video Color Spaces 209

- RGB 210
- YUV 210
- CCIR 211

Mathematical Operations Performed During Conversions 211

Implications of Color-Space Conversions 212

- Precision of Color Conversions Done by the OCTANE Digital Video Option 212
- Range Issues For Color Conversions Done by Any Means 213

Example Color Conversions 216

- Example 1: 100% Color Bars 216
- Example 2: Luminance Ramp 220
- Example 3: Simultaneous Chroma/Luma Ramp 224

Index 261

List of Figures

Figure 1-1	VL System Components 4
Figure 1-2	Simple VL Path 9
Figure 1-3	Simple VL Blending 10
Figure 2-1	Zoom and Decimation 29
Figure 2-2	Clipping an Image 31
Figure 2-3	Zoom, Size, Offset, and Origin 32
Figure 2-4	vlGetNextValid() and vlGetLatestValid() 55
Figure 5-1	Hardware Representation 86
Figure 5-2	Software Representation 87
Figure 7-1	Blender Node 104
Figure 7-2	Keyer and Flat-Background Generator Locations on Source Nodes 105
Figure 7-3	Setting Up the Blender Node 106
Figure 7-4	Binary Compositing 109
Figure 7-5	OCTANE Digital Video Keying, Wipe, and Blender Control Relationships 113
Figure 7-6	Luma Keying Application: Titling 114
Figure 7-7	Relationships Between OCTANE Digital Video Luma Keying Controls 115
Figure 7-8	Chroma Keying Application: TV Weather Map 116
Figure 7-9	Relationships Between OCTANE Digital Video Chroma Keying Controls 118
Figure 7-10	Value, Range, and Transition (Keyer Detail) for a Channel 122
Figure 8-1	Color-Space Conversion Software Model 126
Figure 8-2	Color-Space Conversion Input to Output Paths: YUV to G’B’R’ 128

Figure 9-1 Video Texture-Mapping Hardware Configuration Block Diagram 148
Figure 9-2 Video Delay Through Texture-Map Interface Field Buffer 152
Figure 9-3 Video Delay Through Graphics TRAM 152
Figure 9-4 Video Delay Through Graphics Framebuffer 152
Figure 9-5 Zoom, Size, and Offset for Video Texture Mapping 156
Figure B-1 Rounding for Memory Drain 189
Figure C-1 RGB Cube in CCIR Space 214
Figure C-2 Color Cube With Luminance/Chrominance Ramp Vector 215
Figure C-3 100% Color Bars: Cr/R 217
Figure C-4 100% Color Bars: Y/G 218
Figure C-5 100% Color Bars: Cb/B 219
Figure C-6 Luminance Ramp: Cr/R 221
Figure C-7 Luminance Ramp: Y/G 222
Figure C-8 Luminance Ramp: Cb/B 223
Figure C-9 Chroma/Luma Ramp: Cr/R 225
Figure C-10 Chroma/Luma Ramp: Y/G 226
Figure C-11 Chroma/Luma Ramp: Cb/B 227
Figure Gl-1 SMPTE Color Bars (75%) 234
Figure Gl-2 Color Burst and Chrominance Signal 235
Figure Gl-3 Component Video Signals 237
Figure Gl-4 Horizontal Blanking 243
Figure Gl-5 Horizontal Blanking Interval 244
Figure Gl-6 Waveform Monitor Readings With and Without Setup 251
Figure Gl-7 SMPTE Time Code 252
Figure Gl-8 Red or Blue Signal 256
Figure Gl-9 Y or Green Plus Sync Signal 256
Figure Gl-10 Video Waveform: Composite Video Signal With Setup (Typical NTSC) 257
Figure Gl-11 Video Waveform: Composite Video Signal (Typical PAL) 258

List of Tables

Table 1-1	Video Formats for OCTANE Digital Video 12
Table 2-1	VL Event Masks 21
Table 2-2	Data Transfer Controls 23
Table 2-3	Dimensions for Timing Choices 24
Table 2-4	VL_FORMAT 25
Table 2-5	Packing Types for Eight Bits per Component 26
Table 2-6	Packing Types for Ten Bits per Component 27
Table 2-7	OCTANE Digital Video Packing Types for Ten Bits per Component 28
Table 2-8	VL_RATE Values (Items per Second) 34
Table 2-9	VL Buffer and DMBuffer API Functions 45
Table 2-10	Buffer Size Requirements 53
Table 2-11	Buffer-Related Calls 55
Table 2-12	Calls for Extracting Data From a Buffer 56
Table 3-1	Device-Independent Controls for the OCTANE Digital Video Option 62
Table 3-2	VL Control Groupings 67
Table 4-1	VL Events for the OCTANE Digital Video Device 71
Table 4-2	VL Event Handling Routines 75
Table 5-1	Dynamic Effects of Various Video Data Path Controls 90
Table 5-2	Internal Video Sync Signal Producers and Consumers 92
Table 7-1	General Blender Controls 107
Table 7-2	Choices for Blend Functions A and B 110
Table 7-3	OCTANE Digital Video Luma Keying Controls 115
Table 7-4	OCTANE Digital Video Chroma Keying Controls 117
Table 7-5	Controls for Fades, Tiles, and Wipes 120
Table 7-6	OCTANE Digital Video Controls Specific to Wipes 120

Table 8-1	Supported Video Output Formats	126
Table 8-2	Supported Packing Types for Color-Space Conversion	129
Table 8-3	Supported Ranges	131
Table 8-4	Color-Space Conversion Node Ports	133
Table 8-5	Controls for Standard Color-Space Conversion	135
Table 8-6	Image-Processing Controls	137
Table 8-7	Coefficient Formats	138
Table B-1	Device Node Controls	171
Table B-2	Blender Node Controls	176
Table B-3	Controls for Standard Color-Space Conversion	180
Table B-4	Image-Processing Controls	181
Table B-5	Memory Source Node Controls	184
Table B-6	Memory Drain Node Controls	190
Table B-7	Screen Source Node Controls	195
Table B-8	Screen Drain Node Controls	198
Table B-9	Controls for Video Texture Mapping	200
Table B-10	Video Source Node Controls	203
Table B-11	Video Drain Node Controls	205
Table Gl-12	Videotape Formats	255

About This Guide

The OCTANE™ Digital Video option board lets a Silicon Graphics® O2™ (OCTANE) workstation input and output graphic and video images and record them to disk or videotape.

The OCTANE Digital Video option utilizes calls and controls in the Silicon Graphics Digital Media library, such as the Video Library. This guide explains features of the Video Library (VL) that pertain to the OCTANE Digital Video option and gives step-by-step instructions for creating VL programs that make use of the OCTANE Digital Video capabilities.

Audience

This guide is written for the sophisticated video user with a background in C programming who wishes to develop video programs for OCTANE Digital Video capabilities.

Structure of This Document

This guide contains the following chapters and appendixes:

- Chapter 1, “Features of the OCTANE Digital Video Option,” introduces the features and capabilities of the OCTANE Digital Video board. It explains VL features and architecture, and presents the VL programming model.
- Chapter 2, “Creating Video Programs With the Video Library,” explains how to open a connection to the video daemon and set up a data path, how to set data transfer parameters, how to display video data onscreen, how to transfer video data, and how to end data transfer by presenting an annotated sample program that displays live video input in a graphics window.
- Chapter 3, “Using VL Controls,” explains VL control type and values, VL control fraction ranges, VL control classes, and VL control groupings.

- Chapter 4, “Event Handling,” presents the VL events for the OCTANE Digital Video option and details querying VL events, creating a VL event loop, and creating a main loop with callbacks.
- Chapter 5, “Managing Connections,” explains how to set up more complex paths in OCTANE Digital Video programs by specifying connectivity and avoiding dynamic switching problems. It explains connectivity for the OCTANE Digital Video option by presenting details of board and software architecture.
- Chapter 6, “Video Real-Time Capture and Playback,” gives guidelines for optimizing capture or playback to system memory or disk.
- Chapter 7, “Blending, Keying, and Transitions,” explains how to use VL to perform chroma keying, luma keying, alpha keying, and transitions. It explains the blend node, keying, the keyer, and blending controls for the OCTANE Digital Video option.
- Chapter 8, “Using Color-Space Conversion,” describes features of the color-space conversion node and explains how to perform standard and nonstandard color-space conversions.
- Chapter 9, “Using Video Texture Mapping,” describes features of the texture nodes and explains how to capture video fields into the O2 graphics texture memory, from where they can be used as textures, just like images loaded into texture memory.
- Appendix A, “Return Codes,” lists and explains VL return messages for the OCTANE Digital Video board.
- Appendix B, “OCTANE Digital Video Nodes and Their Controls,” gives information on the OCTANE Digital Video nodes and their controls.
- Appendix C, “OCTANE Digital Video Color-Space Conversions,” explains OCTANE Digital Video color spaces, mathematical operations performed during conversions, and implications of color-space conversions.

A glossary and an index complete this guide.

Other Documents

The following documents are also included with the OCTANE Digital Video option:

- *OCTANE Digital Video and OCTANE Compression Owner's Guide* (007-3466-001)
- *Digital Media Programming Guide* (007-1799-060) (online only)

Conventions

These type conventions and symbols are used in this guide:

Helvetica Bold	Hardware labels
Italics	Executable names, filenames, IRIX commands, manual or book titles, new terms, program variables, tools, utilities, variable command-line arguments, variable coordinates, and variables to be supplied by the user in examples, code, and syntax statements
Bold	Function names
`Fixed-width type`	Error messages, prompts, and onscreen text
`Bold fixed-width type`	User input, including keyboard keys (printing and nonprinting); literals supplied by the user in examples, code, and syntax statements
""	(Double quotation marks) Onscreen menu items and references in text to document section titles
[]	(Brackets) Surrounding optional syntax statement arguments

Chapter 1

Features of the OCTANE Digital Video Option

The OCTANE Digital Video option board and the Video Library (VL) provide video input and output for OCTANE workstations.

This chapter introduces

- "OCTANE Digital Video Board Capabilities"
- "Video Library Capabilities"
- "VL System Software Architecture"
- "VL Architectural Model of Video Devices"
- "OCTANE Digital Video Formats"

For an introduction to video, see the latest version of the *Digital Media Programming Guide* (007-1799-060 or later).

OCTANE Digital Video Board Capabilities

Building on its broadcast-quality, 10-bit digital video architecture, the OCTANE Digital Video option provides a solid foundation for unlimited applications. You can use O2 graphics with real-time video and keyed or alpha output of the 32-bit, double-buffered graphics for broadcast applications. Post-production professionals can use OCTANE Digital Video to capture and play back uncompressed 10-bit video to and from main system memory.

You can send and receive live component video from any serial CCIR-601/SMPTE-259M-compliant device. Compatible with 525-line (NTSC) and 625-line (PAL) standards, OCTANE Digital Video accommodates all major formats of serial digital video I/O:

- two channels of YUV 4:2:2 (8- or 10-bit) (single-link)
- one channel of YUVA 4:2:2:4 (8- or 10-bit) (dual-link)
- one channel of YUVA 4:4:4:4 (8- or 10-bit) (dual-link)
- one channel RP175 (RGBA 8- or 10-bit) (dual-link)
- two channels of arbitrary 8-bit data (single-link)

Additional connections provide genlock input, genlock loopthrough, and GPI trigger signals. You can use the genlock to lock output to analog house sync or to either digital input. For conversion to or from component or composite analog video, use third-party digital-to-analog and analog-to-digital solutions.

Note: The *OCTANE Digital Video and OCTANE Compression Installation Guide* (document number 007-3466-001) contains instructions on connecting D/A and A/D converters and other options to connectors on the OCTANE Digital Video board’s I/O panel.

The OCTANE Digital Video serial digital I/O produces valid output only when it operates in CCIR non-square pixel mode. The I/O ports are not usable when the device operates in square pixel mode, although the remainder of the device is usable. The non-square pixel modes are used with the OCTANE Compression™ option card to capture digital video signals to disk in real time. Serial digital video I/O coupled with low compression ratios dramatically reduces storage and network bandwidth requirements, thus facilitating demanding applications such as spot playback and nonlinear video editing.

For input, you can use, in any combination

- graphics screen
- OCTANE Compression
- digital video
- main memory

The real-time 8-bit alpha blender and key generator enable live creation of many fundamental video effects, including overlays, dissolves, fades, wipes, chroma and luma keying, and shadow.

Video Library Capabilities

The Video Library provides a software interface to the OCTANE Digital Video 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 full-rate video output

The Video Library (VL) is a collection of device-independent and device-dependent C language calls for Silicon Graphics workstations equipped with video options. The VL provides generic video tools, including simple tools for importing and exporting digital data to and from Silicon Graphics systems, 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 directories */usr/share/src/dmedia/video/vl* and */usr/share/src/dmedia/video/vl/OpenGL*).

The VL works with other Silicon Graphics libraries, such as OpenGL®. 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.

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. The VL provides an API that enables applications to capture or play back video from system memory.

The software for the OCTANE Digital Video board includes a graphical user interface, */usr/sbin/vcp*, that makes it convenient to access VL capabilities.

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 1-1 diagrams the interaction between the VL, the video daemon, the kernel, the hardware, and the X Window System server.

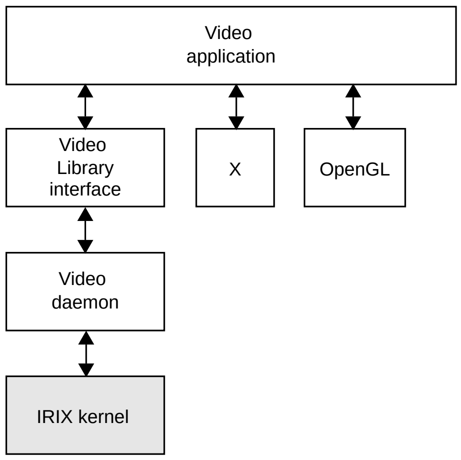


Figure 1-1 VL System Components

The VL communicates with the IRIX kernel for device initialization, vertical retrace, setup, and maintenance of any device-supported direct memory access (DMA). See Chapter 1 of the *Digital Media Programming Guide* for more information on interfacing to other libraries.

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 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.
vlcmd	Use the Video Library command-line interface to enter Video Library shell-level and other commands.
videoin	Use the video input window tool to view input video in a window.
videoout	Use the video output tool to output video from a rectangular 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 X server, number of devices on the server, and the types and ID numbers of nodes, sources, and drains on each device.
vintovout	Use this tool to display video input on the device attached to video output.
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, which can be translated or left as raw data, can be used by the *memtovid* tool.
memtovid	Use this tool to output 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 reference pages, these tools have explanations in the *Media Control Panels User's Guide*. Similar applications are supplied in source-code form as examples in the directories */usr/share/src/dmedia/video/vl* and */usr/share/src/dmedia/video/vl/OpenGL*).

Library and Header Files

The client library is */usr/lib/libvl.so*. The header files for the VL are in */usr/include/dmedia*. The header file for the VL, *vl.h*, contains the main definition of the VL API and controls. The header files for OCTANE Digital Video are */usr/include/dmedia/vl_mgv.h* (linked to */usr/include/vl/dev_mgv.h*) and */usr/include/dmedia/vl_impact.h* (linked to */usr/include/vl/dev_impact.h*).

The header file */usr/include/dmedia/vl_impact.h* contains definitions common to the OCTANE Digital Video and OCTANE Compression devices.

VL Architectural Model of Video Devices

The VL recognizes these classes of objects:

- *devices*, each including sets of nodes

  A video device can be internal, such as the OCTANE Digital Video board, or external, such as a videotape recorder connected to the OCTANE Digital Video board.
- *nodes*: sources, drains, and internal nodes
- *paths*, connecting sources and drains
- *ports*, the entities on nodes that produce or consume video data
- *events*, for monitoring video I/O status
- *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
- *buffer*s: for sending video data to and receiving video data from host memory; these can be either VL buffers or DMbuffers

Central concepts for VL are *node*, *path*, and *port*.

Node

The node is an endpoint or internal processing element of the path, such as a video *source* like a VTR, video *drain* (such as to the O2 screen), a *device* (video), or the *blender* in which video sources are combined for output to a drain.

Path

The path is an abstraction for a way of moving data around. A path is a set of nodes with video routes (connections) between the ports on the nodes. A path defines the useful connections between video sources and video drains. Figure 1-2 shows a simple path in which a frame from a videotape is displayed in a workstation window.

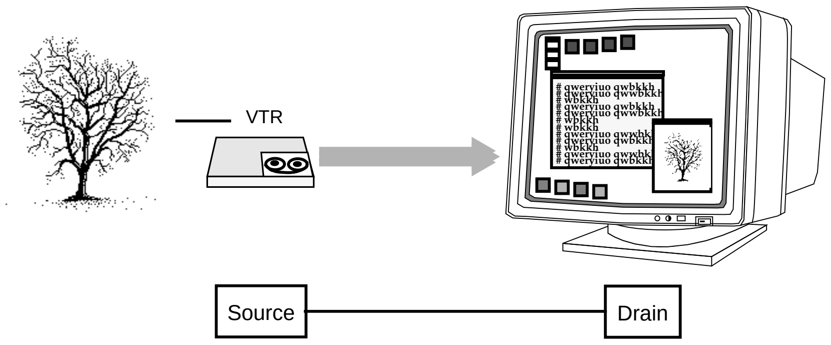
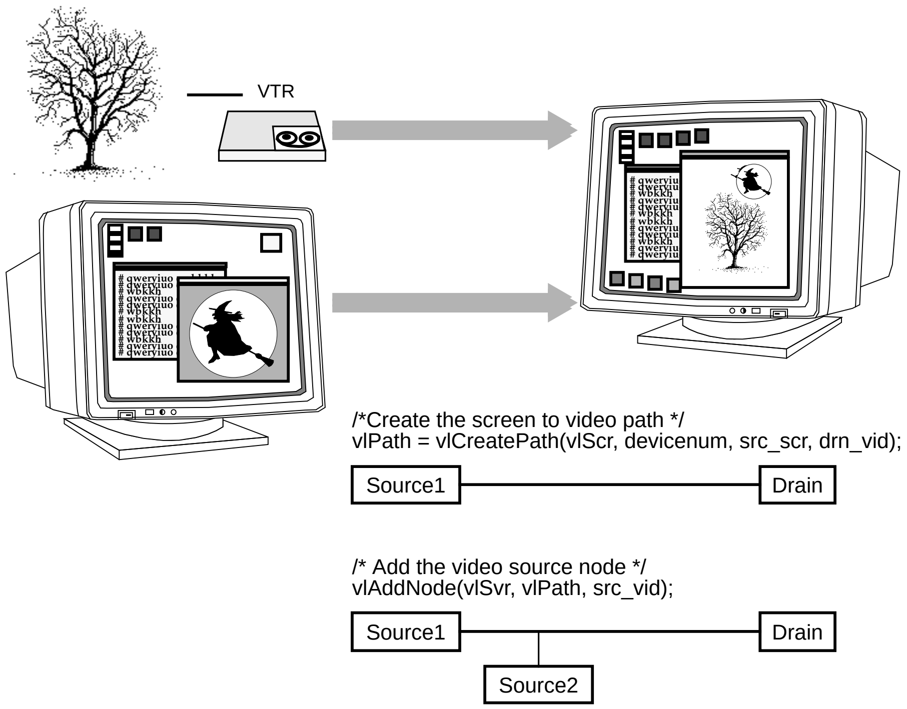


Figure 1-2 Simple VL Path

Figure 1-3 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.



```
/*Create the screen to video path */
vlPath = vlCreatePath(vlScr, devicenum, src_scr, drn_vid);
```

```
/* Add the video source node */
vlAddNode(vlSvr, vlPath, src_vid);
```

Figure 1-3 Simple VL Blending

Port

The port is an entity on a node that produces or consumes video data.

Most nodes have only one port, such as the video in or video out nodes. Each internal node has at least two ports, input (drain) and output (source). The blend node has several ports (A alpha in, A pixel in, B alpha in, B pixel in, pixel out, alpha out).

Ports have several attributes:

- *link type*: single-link or dual-link
- *data type*: alpha, pixel, or pixel-alpha (dual-link)

  A device can use this attribute internally to handle data conversions or routing. For example, the OCTANE Digital Video board includes an alpha LUT to convert CCIR-range pixel data to full-range alpha values.
- *direction*: source or drain
- *enumerator*: A, B, C, and so on, used if a path has several ports with the same link type, data type, and direction

Ports produce or consume various types of data: pixel, alpha, or dual-link data. The identification of the port as pixel or alpha may cause the video stream to be treated differently. For example, alpha data, which can be transmitted or received in the CCIR range only, is internally expanded to full range before it is used. No range expansion is performed for pixel data. Dual-link channels carry both alpha and pixel data, although one data type may be ignored depending on the format.

Ports have generic names; for example:

- VL_IMPACT_PORT_PIXEL_SRC_A: source of a pixel stream (first, or only, port instance)
- VL_IMPACT_PORT_ALPHA_DRN_B: drain of an alpha stream (second port instance)

For the symbolic names for ports, see */usr/include/dmedia/vl_impact.h*. Appendix B, “OCTANE Digital Video Nodes and Their Controls,” in this guide gives the ports associated with each node.

The connections between ports on nodes determine the topology of a path. Single-link ports can be connected to single-link ports only; dual-link ports can be connected to dual-link ports only.

Data flows from a source port to a drain port. It is not permissible to connect a source port to another source port, or a drain port to another drain port. Chapter 5, “Managing Connections,” provides more information about specifying connections.

OCTANE Digital Video Formats

The OCTANE Digital Video board translates video signals into a form usable by the OCTANE workstation. It also does the reverse, translating graphics from the OCTANE display into video signals. Table 1-1 summarizes the formats that the OCTANE Digital Video board supports.

Table 1-1 Video Formats for OCTANE Digital Video

Format	Signal	Nodes
Digital component YCrCbA serial (VL_FORMAT_DIGITAL_COMPONENT_SERIAL)	YCrCb 4:2:2 serial digital signal with 8- or 10-bit words. Component ranges are 16 to 235 (8-bit) or 64 to 940 (10-bit). This format is also used to specify YCrCbA 4:4:4:4. Two streams are required to carry this format. The first is 4:2:2 YCrCb (u0, y0, v0, y1, u2, y2...). The second is 4:2:2 ACrCb (u1, a0, v1, a1, u3, a2...). Conforms to the CCIR-601/656 specification.	All memory nodes
SMPTE YUV (VL_FORMAT_SMPTE_YUV)	Contains YUV components in the range 1-254; superblack and superwhite values can be present.	All memory nodes
Digital component RGB serial (VL_FORMAT_DIGITAL_COMPONENT_RGB_SERIAL)	Dual-link RGBA signal with GBR 4:2:2 (b0, g0, r0, g1, b2, g2, r2...) on the first link and ABR 4:2:2 (b1, a0, r1, a1, b3, a2, r3...) on the second link. Component ranges are 16 to 235 (8-bit) or 64-940 (10-bit). Conforms to the RP175 specification.	All VGI1 memory nodes
RGB (VL_FORMAT_RGB)	Full-range 8-bit or 10-bit per component RGBA. Component range is 0 to 255 (8-bit) and 0-1023 (10-bit).	All VGI1 memory nodes
Raw data (VL_FORMAT_RAW_DATA)	Used for encoding arbitrary 8-bit data values (0 to 255 range) in a 10-bit video signal. Within the coded 10-bit word, bit 9 is 0, bit 8 is 1, and bits 7 through 0 carry the 8-bit data value. When this format is used, the packing is irrelevant.	All VGI1 single-link nodes

For information on the requirements for recording to and from video, see the *Digital Media Programming Guide*.

Chapter 2

Creating Video Programs With the Video Library

Video Library (VL) calls let you perform video teleconferencing, blend computer-generated graphics with frames from videotape or any video source, and output the input video source to the graphics monitor, to a video device such as a VCR, or both.

This chapter explains the basics of creating video programs for OCTANE Digital Video:

- "The VL Programming Model"
- "Performing Preliminary Steps"
- "Opening a Connection to the Video Daemon"
- "Specifying Nodes on the Data Path"
- "Creating and Setting Up the 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"
- "Example Programs"

The VL Programming Model

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()**

Data transfers fall into two categories:

- transfers involving memory (video to memory, memory to video), which require setting up a buffer
- transfers not involving memory (such as video to screen and graphics to video), which do not require a buffer

For the two categories of data transfer, based on the VL programming model, the process of creating a VL application consists of these steps:

1. Open a connection to the video daemon (**vlOpenVideo()**); if necessary, determine which device the application will use (**vlGetDevice()**, **vlGetDeviceList()**).
2. Specify nodes on the data path (**vlGetNode()**).
3. Create the path (**vlCreatePath()**).
4. (Optional step) Add more connections to a path **(vlAddNode()**).
5. Set up the hardware for the path (**vlSetupPaths()**).
6. Specify path-related events to be captured (**vlSelectEvents()**).
7. Set input and output parameters (controls) for the nodes on the path (**vlSetControl()**).
8. For transfers involving memory, create a VL buffer to hold data for memory transfers (**vlGetTransferSize()**, **dmBufferCreatePool()** or **vlCreateBuffer()**).
9. For transfers involving memory, register the buffer (**vlRegisterBuffer()**) or (video-to-memory only) **vlDMBufferPoolRegister()**
10. Set the path topology (**vlSetConnection()**).
11. Start the data transfer (**vlBeginTransfer()**).
12. For transfers involving memory, get the data and manipulate it (DMbuffers: **vlDMBufferGetValid()**, **vlGetActiveRegion()**, **dmBufferFree()**; VL buffers: **vlGetNextValid()**, **vlGetLatestValid()**, **vlGetActiveRegion()**, **vlPutFree()**).
13. Clean up (**vlEndTransfer()**, **vlDeregisterBuffer()**, **vlDestroyPath()**, **dmBuffer()** or **vlDestroyBuffer()**, **vlCloseVideo()**).

Performing Preliminary Steps

To build programs that run under VL, you must

- install the *dmedia_dev* option
- link with *libvl.so*
- include *vl.h* and *dev_mgv.h*

The client library is */usr/lib/libvl.so*. The header files for the VL are in */usr/include/dmedia*. The header file for the VL, *vl.h*, contains the main definition of the VL API and controls. The header files for OCTANE Digital Video are */usr/include/dmedia/vl_mgv.h* (linked to */usr/include/vl/dev_mgv.h*) and */usr/include/dmedia/vl_impact.h* (linked to */usr/include/vl/dev_impact.h*).

Note: When building a VL-based program, you must add *-lvl* to the linking command.

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:

```
vlSvr = 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 vlSvr, int type, int kind, int number)
```

where

VLNode	is a handle for the node, used when setting controls or setting up paths
vlSvr	names the server (as returned by **vlOpenVideo()**)

type specifies the type of node:

- VL_SRC: source
- VL_DRN: drain
- VL_INTERNAL: internal node, such as the blend node
- VL_DEVICE: device for device-global controls

  Note: If you are using VL_DEVICE, the kind should be set to 0.

kind specifies the kind of node:

- VL_BLENDER; see Chapter 7, "Blending, Keying, and Transitions,"for an explanation of this node
- VL_CSC: color-space conversion, available if the color-space conversion option board is installed; see Chapter 8, "Using Color-Space Conversion," for an explanation of this node
- VL_FB: internal framebuffer node for freezing video
- VL_MEM: region of workstation memory
- VL_SCREEN: workstation screen
- VL_VIDEO: connection to a video device; for example, a video tape deck or camera

Note: Appendix B, "OCTANE Digital Video Nodes and Their Controls," gives full details of all OCTANE Digital Video nodes.

number is the number of the node in cases of two or more identical nodes, such as two video source nodes

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(vlSvr, path, VL_ANY, VL_DEFAULT_SOURCE, &ctrlval);
nodehandle = vlGetNode(vlSvr, VL_SRC, VL_VIDEO, ctrlval.intVal);
```

In the first line above, the last argument is a struct that retrieves the value. Corresponding to VL_DEFAULT_SOURCE, the control VL_DEFAULT_DRAIN gets the default VL_SRC node.

Creating and Setting Up the Data Path

Once nodes are specified, use VL calls to

- create the path
- get the device ID
- add nodes (optional step)
- set up the data path
- specify the path-related events to be captured

Creating the Path

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

```
VLPath vlCreatePath(VLServer vlSvr, VLDev vlDev
    VLNode src, VLNode drn)
```

This code fragment creates a path if the device is unknown:

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

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

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

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

Getting the Device ID

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

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

For example:

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

Adding a Node

For this optional step, use **vlAddNode()**. Its function prototype is

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

where

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

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

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

where

vlSvr	names the server to which the path is connected
paths	specifies a list of paths you are setting up

count specifies the number of paths in the path list

ctrlusage specifies usage for path controls:

- 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, only 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 node, 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(vlSvr, (VLPathList)&path, 1, VL_SHARE,
    VL_LOCK) < 0)
{
    vlPerror(_progName);
    exit(1);
}
```

Note: The Video Library infers the connections on a path if **vlBeginTransfer()** is called and no drain nodes have been connected using **vlSetConnection()** (implicit routing). To specify a path that does not use the default connections, use **vlSetConnection()** (explicit routing). Chapter 5, “Managing Connections,” explains the use of this function and related requirements.The following rules are used in determining the connections:

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

Note: Do not combine implicit and explicit routing.

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 vlSvr, VLPath path, VLEventMask eventmask)
```

where

vlSvr	names the server to which the path is connected
path	specifies the data path.
eventmask	specifies the event mask; Table 2-1 lists the possibilities

Table 2-1 lists and describes the VL event masks.

Table 2-1 VL Event Masks

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

For example:

```
vlSelectEvents(vlSvr, path, VLTransferCompleteMask);
```

Event masks can be Or'ed; for example:

```
vlSelectEvents(vlSvr, path, VLTransferCompleteMask |
      VLTransferFailedMask);
```

For more details on VL event handling, see Chapter 4, "Event Handling."

Setting Parameters for Data Transfer to or From Memory

Transferring data to or from memory requires creating a DMbuffer or VL buffer, as explained in "Transferring Video Data to and From Devices," later in this chapter. This section explains how to set node controls for data transfer.

To set frame data size and to convert from one video format to another, apply controls to the nodes. The use of source node controls and drain node controls is explained separately in this section.

Important data transfer controls for source and drain nodes are summarized in Table 2-2. They should be set in the order in which they appear in the table.

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.

Note: For drain nodes, VL_PACKING must be set first. 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.

Table 2-2 Data Transfer Controls

Control	Basic Use	Video Nodes	Memory Nodes	Screen Nodes
VL_FORMAT	Video format on the physical connector	See "Using VL_FORMAT" in this chapter	N/A	N/A
VL_TIMING	Video timing	See Table 2-3 for values	Not applicable	Not applicable
VL_CAP_TYPE	Setting type of field(s) or frame(s) to capture	Not applicable	VL_CAPTURE_NONINTERLEAVED VL_CAPTURE_INTERLEAVED VL_CAPTURE_EVEN_FIELDS VL_CAPTURE_ODD_FIELDS VL-CAPTURE_FIELDS	Not applicable
VL_PACKING	Pixel packing (conversion) format	Not applicable	Changes pixel format of captured data; see Table 2-5 for values	Not applicable
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
VL_WINDOW	Setting window ID for video in a window	Not applicable	Not applicable	Window ID
VL_RATE	Field or frame transfer speed	Depends on capture type as specified by VL_CAP_TYPE	Not applicable	Not applicable

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 is not what you requested.

To specify the controls, use **vlSetControl()**, for which the function prototype is

```
int vlSetControl(VLServer vlSvr, VLPath vlPath, VLNode node,
      VLControlType type, VLControlValue * value)
```

The use of VL_TIMING, VL_FORMAT, VL_PACKING, VL_SIZE, VL_OFFSET, VL_CAP_TYPE, and VL_RATE is explained in more detail in the following sections.

Using VL_TIMING

Timing type expresses the timing of video presented to a source or drain. Table 2-3 summarizes dimensions for VL_TIMING.

Table 2-3 Dimensions for Timing Choices

Timing	Maximum Width	Maximum Height
VL_TIMING_525_SQ_PIX (12.27 MHz)	640	486
VL_TIMING_625_SQ_PIX (14.75 MHz)	768	576
VL_TIMING_525_CCIR601 (13.50 MHz)	720	486
VL_TIMING_625_CCIR601 (13.50 MHz)	720	576

Using VL_FORMAT

To specify video input and output formats of the video signal on the physical connector, use VL_FORMAT.

Table 2-4 VL_FORMAT

Format	Explanation	Supported by Node
VL_FORMAT_DIGITAL _COMPONENT_SERIAL	8- or 10-bit YCrCb	Single-link and dual-link
VL_FORMAT_SMPTE_YUV	Backwards compatibility: 8- or 10-bit YCrCb	Single-link and dual-link
VL_FORMAT_RAW_DATA	Arbitrary 8-bit data (non-video format)	Single-link only
VL_FORMAT_RGB	Full-range 8-bit (0-255) or 10-bit (0-1023) RGBA	Dual-link only
VL_FORMAT_DIGITAL _COMPONENT_RGB_SERIAL	RP175 standard RGBA	Dual-link only

Using VL_PACKING

A video *packing* describes how a video signal is stored in memory, in contrast with a video format, which describes the characteristics of the video signal. For example, the memory source nodes—CC1 and both VGI1 nodes—accept packed video from a DMbuffer or VL buffer and output video in a given format.

Packings are specified through the VL_PACKING control on the memory nodes. This control also converts one video output format to another in memory, within the limits of the nodes.

Note: On dual-linked VGI1 memory nodes, only native packings are available; no conversions can be performed.

Packing types for eight bits per component are summarized in Table 2-5. In this table, the Native To column lists the nodes to which the packing is native; no software conversion is required, so these packings are fastest.

Table 2-5 Packing Types for Eight Bits per Component

Type	Native To	63-56	55-48	47-40	39-32	31-24	23-16	15-8	7-0
VL_PACKING_YVYU_422_8 YUV 4:2:2, single-link	All memory nodes	U0	Y0	V0	Y1	U2	Y2	V2	Y3
VL_PACKING_YUVA_4444_8 YUVA 4:4:4:4, dual-link	VGI1 memory nodes	A0	U0	Y0	V0	A1	U1	Y1	V1
VL_PACKING_AUYV_4444_8 AUYV 4:4:4:4, dual-link	VGI1 memory nodes	V0	Y0	U0	A0	V1	Y1	U1	A1
VL_PACKING_UYV_8_P	VGI1 memory nodes	V0	Y0	U0	V1	Y1	U1	V2	Y2
VL_PACKING_RGBA_8 RGBA, dual-link	VGI1 memory nodes	A0	B0	G0	R0	A1	B1	G1	R1
VL_PACKING_ABGR_8 ABGR, dual-link	VGI1 memory nodes	R0	G0	B0	A0	R1	G1	B1	A1
VL_PACKING_RGB_332_P RGB, single-link Each 8-bit pixel, P*n*, is shown as BBGGGRRR	None	P7	P6	P5	P4	P3	P2	P1	P0
VL_PACKING_Y_8_P Grayscale (luminance only), single-link	None	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
VL_PACKING_RGB_8 RGB, single-link 24-bit word, X*n* are ignored	None	X0	B0	G0	R0	X1	B1	G1	R1
VL_PACKING_BGR_8_P RGB, dual-link	VGI1 memory nodes	R0	G0	B0	R1	G1	B1	R2	G2

Packing types for ten bits per component are summarized in Table 2-6. The ten data bits are left-aligned within a 16-bit word. The hardware sets the lower six bits to zero before it writes them to memory. When reading from memory, the lower six bits are ignored. All are native to VGI1 memory nodes.

Table 2-6 Packing Types for Ten Bits per Component

Type	**63-48**	**47-32**	**31-16**	**15-0**
VL_PACKING_YVYU_422_10 YUV 4:2:2, single-link	[U0]000000	[Y0]000000	[V0]000000	[Y1]000000
VL_PACKING_YUVA_4444_10 YUVA 4:4:4:4, dual-link	[A0]000000	[U0]000000	[Y0]000000	[V0]000000
VL_PACKING_AUYV_4444_10 AUYV 4:4:4:4, dual-link	[V0]000000	[Y0]000000	[U0]000000	[A0]000000
VL_PACKING_ABGR_10 AUYV 4:4:4:4, dual-link	[V0]000000	[Y0]000000	[U0]000000	[A0]000000
VL_PACKING_RGBA_10 RGBA, dual-link	[A0]000000	[B0]000000	[G0]000000	[R0]000000
VL_PACKING_ABGR_10 ABGR, dual-link	[R0]000000	[G0]000000	[B0]000000	[A0]000000

In addition, the OCTANE Digital Video option also supports dual-link AYUAYV, a packed format with three 10-bit components per 32-bit word, with the lowest two bits set to 0. It is native to VGI1 memory nodes. Bits are

- 63-32: [U0][Y0][A0]00
- 31-0: [V0][Y1][A1]00

Finally, OCTANE Digital Video option supports two 10-bit formats that have two bits for alpha, as summarized in Table 2-7.

Table 2-7 OCTANE Digital Video Packing Types for Ten Bits per Component

Type	63-54	53-44	43-34	33-32	31-22	21-12	11-2	1-0
VL_PACKING_A_2_UYV_10 YUVA packed 4:4:4:4 to 32 bits, dual-link	V0	Y0	0	A0	V1	Y1	U1	A1
VL_PACKING_A_2_BGR_10 ABGR, dual-link	R0	G0	B0	A0	R1	G1	B1	A1

Note: Other libraries may use different packing names.

For example:

```
VLControlValue val;

val.intVal = VL_PACKING_RGBA_10;
vlSetControl(vlSvr, path, memdrn, VL_PACKING, &val);
```

If the single-link packings VL_PACKING_RGB_332_P, VL_PACKING_RGB_8, and VL_PACKING_Y_8_P are requested of a memory drain node, the Video Library performs a software conversion to translate the data from a native packing and format. The application receives data in the requested packing and format, although the capture rate may be degraded.

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 2-1 illustrates zooming and decimation.

Note: OCTANE Digital Video screen drain nodes support the full range of VL_ZOOM (7/1, 6/1, 5/1, 4/1, 3/1, 2/1, 1/1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8). Screen source nodes support 1/1 and 1/2. The texture node supports decimation only. The remaining nodes do not support zoom or decimation (the ratio 1/1 only).

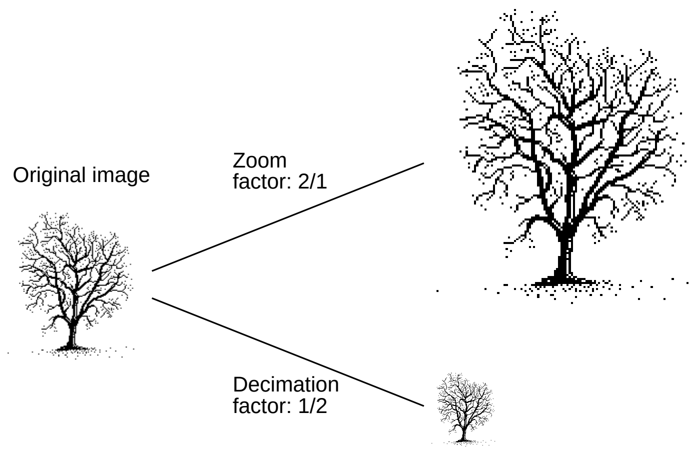


Figure 2-1 Zoom and Decimation

VL_ZOOM takes a nonzero fraction as its argument; do not use negative values. For example, this fragment captures half-size decimation video to the screen:

```
val.fractVal.numerator = 1;
val.fractVal.denominator = 2;
if (vlSetControl(server, screen_path, screen_drain_node, VL_ZOOM,
&val)){
   vlPerror("Unable to set zoom");
   exit(1);
}
```

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 the screen, with clipping to 320 × 243 (NTSC size minus overscan):

```
val.fractVal.numerator = 1;
val.fractVal.denominator = 2;
if (vlSetControl(server,screen_path, screen_drain_node,
   VL_ZOOM, &val))
{
    vlPerror("Unable to set zoom");
    exit(1);
}
```

```
val.xyVal.x = 320;
val.xyVal.y = 243;
if (vlSetControl(server, screen_path, screen_drain_node,
   VL_SIZE, &val))
{
    vlPerror("Unable to set size");
    exit(1);
}
```

This fragment captures *xsize* × *ysize* video with as much decimation as possible, assuming the size is smaller than the video stream:

```
if (vlGetControl(server, screen_path, screen_source, VL_SIZE, &val))
{
   vlPerror("Unable to get size");
   exit(1);
}
if (val.xyVal.x/xsize < val.xyVal.y/ysize)
   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, screen_path, screen_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, screen_path, screen_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 2-2 illustrates clipping.

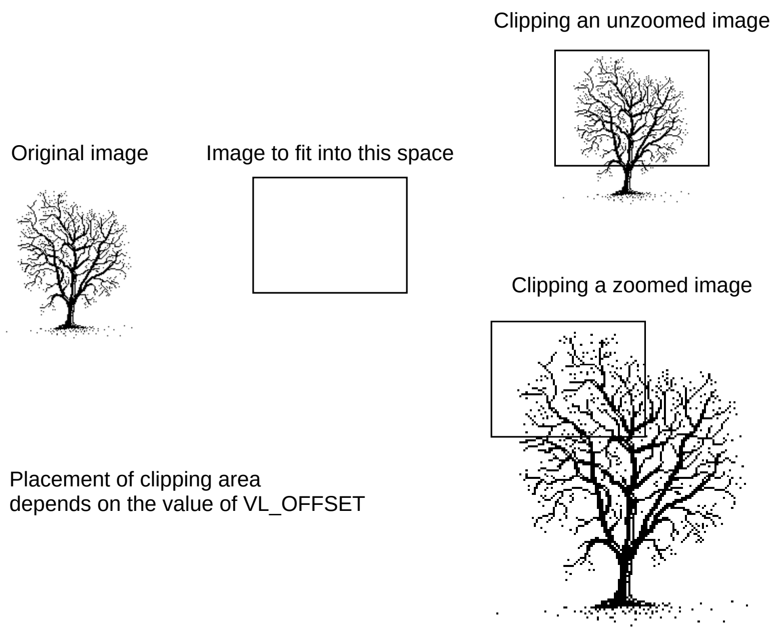

Clipping an unzoomed image
Original image
Image to fit into this space
Clipping a zoomed image
Placement of clipping area
depends on the value of VL_OFFSET

Figure 2-2 Clipping an Image

For example, to display PAL video in a 320 × 243 space, clip the image to that size, as shown in the following fragment:

```
VLControlValue value;
value.xyval.x=320;
value.xyval.y=243;
vlSetControl(vlSvr, 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.

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(vlSvr, path, drn, VL_OFFSET, &value);
```

To capture the blanking region, set offset to a negative value.

Figure 2-3 shows the relationships between the source and drain size, offset, and origin.

Note: For memory nodes, VL_OFFSET and VL_SIZE in combination define the active region of video that is transferred to or from memory.

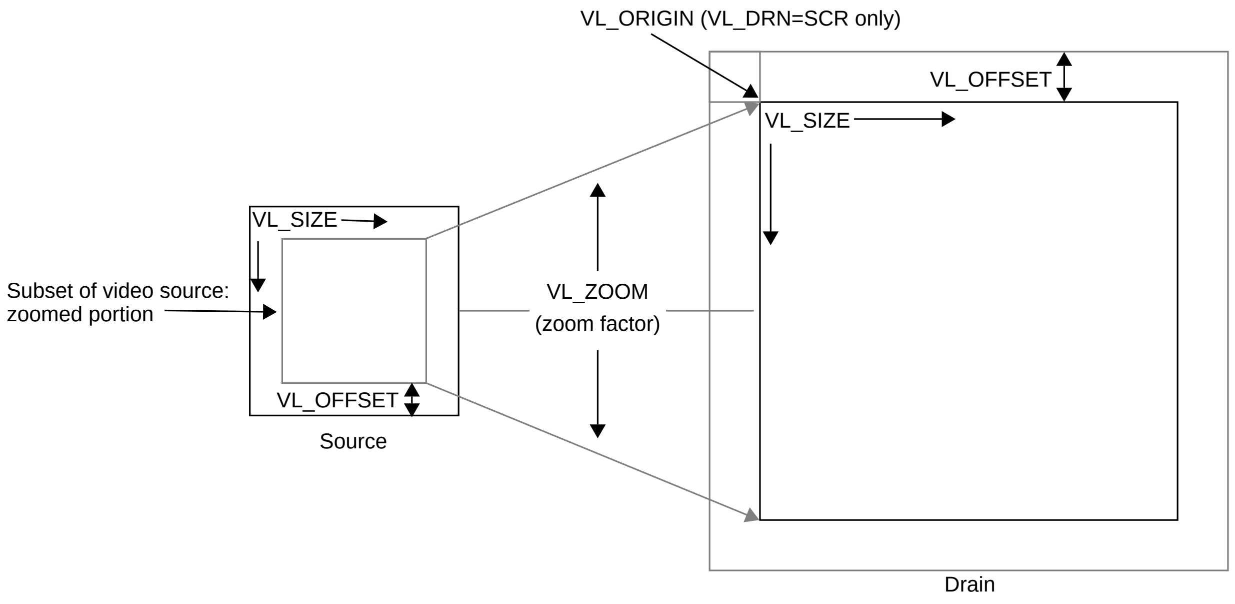

VL_ORIGIN (VL_DRN=SCR only)
VL_OFFSET
VL_SIZE
VL_SIZE
Subset of video source: zoomed portion
VL_ZOOM
(zoom factor)
VL_OFFSET
Source
Drain

Figure 2-3 Zoom, Size, Offset, and Origin

Using VL_CAP_TYPE and VL_RATE

An application can request that the OCTANE Digital Video option capture or play back a video stream in a number of ways. For example, the application can request that each field be placed in its own buffer, that each buffer contain an interleaved frame, or that only odd or even fields be captured. This section enumerates the capture types that the OCTANE Digital Video option supports.

A *field mask* is useful for identifying which fields will be captured and played back and which fields will be dropped. A field mask is a bit mask of 60 bits for NTSC or 50 bits for PAL (two fields per frame). A numeral 1 in the mask indicates that a field is captured or played back, while a zero indicates that no action occurs.

For example, the following field mask indicates that every other field will be captured or played back:

```
10101010101010101010...
```

Capture types are as follows:

- VL_CAPTURE_NONINTERLEAVED
- VL_CAPTURE_INTERLEAVED (not used for texture node VL_TEX)
- VL_CAPTURE_EVEN_FIELDS
- VL_CAPTURE_ODD_FIELDS
- VL_CAPTURE_FIELDS

These capture types apply to both VL buffers and DMbuffers.

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 2-8.

Table 2-8 VL_RATE Values (Items per Second)

VL_CAP_TYPE Value	VL_RATE Value
VL_CAPTURE_NONINTERLEAVED, VL_CAPTURE_INTERLEAVED	NTSC: 1-30 frames/second PAL: 1-25 frames/second
VL_CAPTURE_EVEN_FIELDS, VL_CAPTURE_ODD_FIELDS	NTSC: 1-30 fields/second PAL: 1-25 fields/second
VL_CAPTURE_FIELDS	NTSC: 1-60 fields/second PAL: 1-50 fields/second

Note: Not all rates are supported on all memory nodes; see Appendix C, “OCTANE Digital Video Color-Space Conversions,” for details. The buffer size must be set in accordance with the capture type, as listed in Table 2-10 later in this chapter.

VL_CAPTURE_NONINTERLEAVED

The VL_CAPTURE_NONINTERLEAVED capture type specifies that frame-size units are captured noninterleaved. Each field is placed in its own buffer, with the dominant field in the first buffer. If one of the fields of a frame is dropped, all fields are dropped. Consequently, an application is guaranteed that the field order is maintained; no special synchronization is necessary to ensure that fields from different frames are mixed.

The rate (VL_RATE) for noninterleaved capture is in terms of fields and must be even. For NTSC, the capture rate may be from 2-60 fields per second, and for PAL 2-50 fields per second. Because a frame is always captured as a whole, a rate of 30 fields per second results in the following field mask:

```
1100110011001100...
```

The first bit in the field mask corresponds to the dominant field of a frame. The OCTANE Digital Video option waits for a dominant field before it starts the transfer.

If VL_CAPTURE_NONINTERLEAVED is specified for playback, similar guarantees apply as for capture. If one field is lost during playback, it is not possible to "take back" the field. The OCTANE Digital Video option resynchronizes on the next frame boundary, although black or "garbage" video might be present between the erring field and the frame boundary.

The rate during playback also follows the rules for capture. For each 1 in the mask above, a field from the VL buffer is output. During the 0 fields, the previous frame is repeated. Note that the previous *frame* is output, not just the last field. If there are a pair of buffers, the dominant field is placed in the first buffer.

VL_CAPTURE_INTERLEAVED

Interleaved capture interleaves the two fields of a frame and places them in a single buffer; the order of the frames depends on the value set for VL_MGV_DOMINANCE_FIELD (see Table B-5 or Table B-6 in Appendix B for details). The OCTANE Digital Video option guarantees that the interleaved fields are from the same frame: if one field of a frame is dropped, then both are dropped.

The rate for interleaved frames is in frames per second: 1-30 frames per second for NTSC and 1-25 frames per second for PAL. A rate of 15 frames per second results in every other frame being captured. Expressed as a field mask, the following sequence is captured:

```
1100110011001100....
```

As with VL_CAPTURE_NONINTERLEAVED, the OCTANE Digital Video option begins processing the field mask when a dominant field is encountered.

During playback, a frame is deinterleaved and output as two consecutive fields, with the dominant field output first. If one of the fields is lost, the OCTANE Digital Video option resynchronizes to a frame boundary before playing the next frame. During the resynchronization period, black or "garbage" data may be displayed.

Rate control follows similar rules as for capture. For each 1 in the mask above, a field from the interleaved frame is output. During 0 periods, the previous frame is repeated.

This option is not applicable to the texture node VL_TEX.

VL_CAPTURE_EVEN_FIELDS

In the VL_CAPTURE_EVEN_FIELDS capture type, only even (F2) fields are captured, with each field placed in its own buffer. Expressed as a field mask, the captured fields are

```
1010101010101010...
```

The OCTANE Digital Video option begins processing this field mask when an even field is encountered.

The rate for this capture type is expressed in even fields. For NTSC, the range is 1-30 fields per second, and for PAL 1-25 fields per second. A rate of 15 fields per second (NTSC) indicates that every other even field is captured, yielding a field mask of

```
1000100010001000...
```

During playback, the even field is repeated as both the F1 and F2 fields, until it is time to output the next buffer. If a field is lost during playback, black or “garbage” data might be displayed until the next buffer is scheduled to be displayed.

VL_CAPTURE_ODD_FIELDS

The VL_CAPTURE_ODD_FIELDS capture type works the same way as VL_CAPTURE_EVEN_FIELDS, except that only odd (F1) fields are captured, with each field placed in its own buffer. The rate for this capture type is expressed in odd fields. A rate of 15 fields per second (NTSC) indicates that every other odd field is captured. Field masks are the same as for VL_CAPTURE_EVEN_FIELDS.

VL_CAPTURE_FIELDS

The VL_CAPTURE_FIELDS capture type captures both even and odd fields and places each in its own buffer. Unlike VL_CAPTURE_NONINTERLEAVED, there is no guarantee that fields are dropped in frame units. Field synchronization can be performed by examining the UST (Unadjusted System Time), the MSC (Media Stream Count), or the dmedia info sequence number associated with each field.

The rate for this capture type is expressed in fields. For NTSC, the range is 1-60 fields per second, and for PAL 1-50 fields per second. A rate of 30 fields per second (NTSC) indicates that every other field is captured, resulting in the following field mask:

```
101010101010101010...
```

Contrast this with the rate of 30 for VL_CAPTURE_NONINTERLEAVED, which captures every other frame.

Field mask processing begins on the first field after the transfer is started; field dominance, evenness, oddness plays no role in this capture type.

Synchronizing Data Streams

This section explains:

- "Using UST, MSC, and Buffered Media Streams for Synchronization"
- "Media Library Interfaces for UST and MSC"

Using UST, MSC, and Buffered Media Streams for Synchronization

Whenever a VL path is open in continuous mode, the Octane Digital Video board and certain other Silicon Graphics video devices continuously try to dequeue media stream samples from the path's buffer for input, or to enqueue media stream samples onto the path's buffer for output. If the buffer between the application and each device never underflows or overflows, then the application can measure and schedule the timing of input and output signals to 100% of the accuracy of the underlying device.

Occasionally, the application is held off and audio, video, or both come out late. Buffer underflow on output and overflow on input can result from the application not keeping the buffer adequately filled for the following reasons:

- The application is busy with other tasks, allowing too much time between putting fields into the buffer.
- Processes are subject to various interruptions (10-80 ms for some processes) under IRIX™ because
  - the process for filling the buffer is running at too low a priority
  - the process cannot get a resource from IRIX that it needs, such as memory pages

To get around this problem, a mechanism built into the VL helps keep track of data flow into and out of buffers by providing accurate timing information for each frame of video that enters or leaves the system. This mechanism, which can be called UST/MSC, produces matched pairs of two numbers:

- Unadjusted System Time (UST), a time value that is used to state timing measurements to applications
- Media Stream Count (MSC), a count value that identifies a particular media stream sample (a video field or frame)

The device keeps a counter called the device media stream count (device MSC), which increments by one every time the device attempts to enqueue or dequeue a media stream sample, whether or not the enqueue or dequeue attempt is successful. UST/MSC was designed to return timing information in a form that is valid whenever the buffer is not underflowing or overflowing.

The UST/MSC capability and the buffering that goes with it are appropriate for applications and devices such as movie players and digital video editing devices.

UST/MSC affords maximally accurate synchronization when scheduling cannot be guaranteed and some buffering is acceptable. Also, if scheduling becomes reliable at some later point, UST/MSC continues to function the same way with no code changes required; the buffers can be made smaller, and the result is a low-latency application with the same accurate synchronization.

Note that UST/MSC itself

- does not add any latency to an application

  The buffer adds latency: it increases the time the application would take to respond to some output event by changing its input (and vice versa). This solution to the synchronization problem is useful for applications in which a small latency can be sacrificed for more accuracy.
- does not require that an application trade off latency for accuracy
- does not require that an application use any particular size buffer
- delivers the full accuracy of the underlying hardware's timing support regardless of the scheduling characteristics of the application
- could be useful for graphics and texture even for low-latency applications

Following is a high-level algorithm to maintain synchronization of two buffered media streams that send data from memory to hardware outputs; a corresponding one is necessary for the other direction.

```
create video buffer between me and the audio output;
create audio buffer between me and the video output;
while (1)
{
   sleep until one of the buffers is getting empty;
   for (video buffer)
      {
         use UST/MSC to determine:
            “at what time (what UST) will the next video data I enqueue
            on the buffer actually go out the jack of the machine?”;
      }

   for (audio buffer)
      {
         (exact same thing as above, except for audio)
      }

   From the predicted video and audio USTs, determine
      “what is the synchronization error between the audio and video
      streams?”

   Enqueue more frames to fill up the audio and video buffer queues.
   If there is synchronization error, enqueue new frames to either skip
   frames on the stream that is behind or repeat frames on the stream
   that is ahead.
     }
}
```

The answers to the questions in the pseudocode above are obtained with three VL calls that manipulate UST and MSC and are explained in the next section.

Media Library Interfaces for UST and MSC

UST/MSC calls allow you to associate a UST with a particular piece of data that just left a buffer or is about to enter a buffer. The VL calls for determining the MSC and UST—vlGetUSTMSCPair(3dm), vlGetFrontierMSC(3dm), and vlGetUSTPerMSC(3dm)—help synchronize input and output of different data streams in cases where the application is getting data from or putting data into each device via a buffer. The application is at the "frontier" end of this buffer and the devices are at the "device" end of the buffer.

- **vlGetUSTMSCPair()** gets the timing information for each frame/field as it enters or leaves the physical jack of a device.

  This call returns an atomic UST/MSC pair for the jack (specified with the VL_NODE and the VLPort for that node) for a given path that contains a VL_MEM node. The returned MSC is not guaranteed to be the one currently at the jack, nor is it even guaranteed to be the number of any media stream sample currently in the application's buffer. To relate the returned MSC to a particular item in the application's buffer, you must use **vlGetFrontierMSC()**.
- **vlGetFrontierMSC()** gets the frontier MSC associated with a particular VL_MEM node.

  The frontier MSC, at the application end of the media stream, is the MSC of the next item that the application removes from or puts into the buffer.
- **vlGetUSTPerMSC()** gets the time spacing of fields/frames in a path (the nominal average UST time elapsed between media stream samples in a given VLPath that includes a VL_MEM node).

These calls are used for extrapolating a UST/MSC pair as shown in **vlGetFrontierMSC()**. For other types of media streams, a similar mechanism extrapolates the UST/MSC pair; for example, for audio, use equivalent AL calls.

Once you have calculated the extrapolated UST/MSC pairs for both media streams, you can determine the synchronization error. The difference in the audio and video USTs for matching frame numbers is the amount they are out of sync. To resynchronize them, you must enqueue new frames to either skip frames on the stream that is behind or repeat frames on the stream that is ahead. The number of frames to be skipped or repeated is the difference in USTs divided by the frame rate.

To use UST/MSC, the application must have separate handles for each separate piece of data coming in or going out of some kind of buffer. The application can use these handles to specify, for example, a particular frame to output or pixels of a particular field to get.

Note: For complete details, including syntax, code examples, and caveats, see the references pages for these calls.

Displaying Video Data Onscreen

To set up a window for live video, follow these steps, as outlined in the example program *simplev2s.c*.

1. Open an X display window; for example:

   ```
   if (!(dpy = XOpenDisplay("")))
       exit(1);
   ```

2. Connect to the video daemon; for example:

   ```
   if (!(vlSvr = vlOpenVideo("")))
       exit(1);
   ```

3. Create a window to show the video; for example:

   ```
   vwin = XCreateSimpleWindow(dpy, RootWindow(dpy, 0), 10,
       10, 640, 486, 0,
       BlackPixel(dpy,DefaultScreen(dpy)),
       BlackPixel(dpy, DefaultScreen(dpy));
   XMapWindow(dpy, vwin);
   XFlush(dpy);
   ```

4. Create a source node on a video device and a drain node on the screen; for example:

   ```
   src = vlGetNode(vlSvr, VL_SRC, VL_VIDEO, VL_ANY);
   drn = vlGetNode(vlSvr, VL_DRN, VL_SCREEN, VL_ANY);
   ```

5. Create a path on the first device that supports it; for example:

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

6. Set up the hardware for the path and define the path use; for example:

   ```
   vlSetupPaths(vlSvr, (VLPathList)&path, 1, VL_SHARE,
       VL_SHARE);
   ```

7. Set the X window to be the drain; for example:

   ```
   val.intVal = vwin;
   vlSetControl(vlSvr, 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(vlSvr, path, drn, VL_ORIGIN, &val);
   XGetGeometry(dpy, vwin, &dummyWin, &x, &y, &w, &h, &bw,
      &d);
   val.xyVal.x = w;
   val.xyVal.y = h;
   vlSetControl(vlSvr, path, drn, VL_SIZE, &val);
   ```

10. Begin the data transfer:

    ```
    vlBeginTransfer(vlSvr, 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(vlSvr, path);
    vlDestroyPath(vlSvr, path);
    vlCloseVideo(vlSvr);
    ```

Transferring Video Data to and From Devices

This section explains

- "Using Buffers"
- "Transferring Video Data Using DMBuffers"
- "Transferring Video Data Using VL Buffers"

Using Buffers

The VL supports two buffering mechanisms for capturing or playing back video:

- VL buffers: the original buffering mechanism supported by the VL and specific to it
- Digital Media Buffers (DMbuffers): a buffering mechanism allowing video data to be exchanged among video, compression, and graphics devices

  For OCTANE, this buffering mechanism is supported by the Video, Image Converter (dmIC), and Movie libraries. It is available with IRIX 6.4 and subsequent releases.

Note: For complete information on DMbuffers and digital media image converters, see the *Digital Media Programming Guide*.

In general, VL buffers and DMbuffers differ in the following ways:

- buffer structure

  VL buffers are modeled after a ring buffer. The order of segments (buffers) in the ring is inflexible, and care must be taken to ensure that items are obtained and returned in the same order. For example, buffers obtained with **vlGetNextValid()** must be returned using **vlPutFree()** in the same order. Order and allocation of ring segments are intricately related.

  All operations on a VL buffer operate in FIFO order. That is, the first element retrieved by **vlGetNextValid()** is the first returned by **vlPutFree()**. This function does not take an element as a parameter and always puts back the oldest outstanding element.

DMbuffers, in contrast, are contained in a DMbufferpool. The pool itself is unordered; buffers can be obtained from and returned to the pool in any order. Ordering is achieved by a first-in-first-out queue, and is maintained only while the buffers are in the queue. The application or library is free to impose any processing order on buffers, once they have been dequeued.

- buffer size and alignment

  The Video Library is responsible for ensuring that VL buffers are of the appropriate size and alignment for the video device, and for allocating the buffers in the **vlCreateBuffer()** call. Except in rare cases, applications cannot modify these attributes to suit the needs of another library or device.

  Because DMbuffers can be used with libraries and devices besides video, the application queries each library for its buffering requirements. The exact DMbufferpool requirements are the union of all requested constraints and are enforced when the pool is created. For example, if one library requests alignment on 4K boundaries and another requests alignment on 16K boundaries, the 16K alignment is used. By specifying its own pool requirements list, the application can set minimum buffer sizes (such as for in-place processing of video) or cache policies.

- buffers and memory nodes

  With VL buffers, a particular ring buffer is strictly tied to a particular memory node; a DMbufferpool is not necessarily tied to a memory node. A memory source node can receive DMbuffers allocated from any DMbufferpool that meets the memory node's pool requirements. Memory drain nodes obtain DMbuffers from a DMbufferpool specified by the application; this pool is fixed for the duration of a transfer.

Each buffering mechanism has a set of API functions for creating, registering, and manipulating buffers. A mismatch between a buffer mechanism and an API call, for example, applying a VL buffer call to a DMbuffer, results in a VLAPIConflict error return.

Applications can use either VL buffers or DMbuffers, as long as a memory node is used with only one buffering mechanism at a time. If an application uses multiple memory paths, each path can use a different buffering mechanism. To switch buffering mechanisms, the VL path should be torn down and reconstructed.

Table 2-9 shows correspondences between VL buffer and DMbuffer API functions.

Table 2-9 VL Buffer and DMBuffer API Functions

VL Buffer API	dmBuffer API
vlCreateBuffer()	dmBufferCreatePool()
vlPutValid()	vlDMBufferPutValid()
vlRegisterBuffer()	vlDMBufferPoolRegister()
vlDeregisterBuffer()	No equivalent
vlPutFree()	dmBufferFree()
vlGetNextValid()	vlDMBufferGetValid()
vlGetLatestValid()	No equivalent
vlGetFilled()	vlGetFilledByNode()
vlDestroyBuffer()	dmBufferDestroyPool()

vlBufferGetFd()	dmBufferGetPoolFD()
	dmBufferSetPoolSelectSize()
	vlNodeGetFd()

vlBufferAdvise()	dmSetPoolDefaults()
vlBufferReset()	vlDMBufferNodeReset()
vlBufferDone()	Not applicable

Transferring Video Data Using DMbuffers

The DMbuffer is created through the **dmBufferCreatePool()** routine and is associated with a memory node by the **dmPoolRegister()** routine.

When the OCTANE Digital Video option transfers data from the Video Library to an application, it places data in a buffer element and marks the element as *valid*. The application can retrieve the element through the **vlDMBufferGetValid()** routine. When the application is done, it uses the **dmBufferFree()** routine to alert the video device that the buffer element can be reused. For complete details on using DMbuffers, see Chapter 5 of the *Digital Media Programming Guide* (007-1799-060 or later).

This section explains

- “Obtaining DMbufferpool Requirements”
- “Registering a DMBufferpool With the Video Library”
- “Creating a DMbufferpool”
- “Starting Data Transfer”
- “Receiving Buffers From the Video Library”
- “Sending DMbuffers to the Video Library”

Obtaining DMbufferpool Requirements

Before a DMbufferpool is created, you must obtain the pool requirements of any library that will interact with the pool. Pool requirements are maintained in a DMparams list, created using **dmParamsCreate()** and initialized by calling **dmBufferSetPoolDefaults()**. See Chapter 3 in the *Digital Media Programming Guide* for an overview of DMparams. The function prototype for this call is

```
DMstatus dmBufferSetPoolDefaults(DMparams *poolParams, int
bufferCount, int bufferSize, DMboolean cacheable, DMboolean mapped)
```

where

poolParams	specifies the DMparams list to use for gathering pool requirements
bufferCount	specifies the number of buffers the pool should contain
bufferSize	specifies the size of each buffer in the pool

cacheable	specifies whether buffers allocated from the pool can be cached (DM_TRUE) or not (DM_FALSE). For more information on caching, see“Caching” in Chapter 6.
mapped	specifies whether the memory allocated for the pool should be mapped as soon as the pool is created (TRUE), or only when **dmBufferMapData()** is called (FALSE) If an application requires a pointer to buffer contents, for example, to process or store the contents to disk, then the pool should be created mapped. This option improves the performance of the **dmBufferMapData()** call.

The Video Library pool requirements are obtained by calling **vlDMBufferGetParams()** on a memory node:

```
int vlDMBufferGetParams(VLServer svr, VLPath path, VLNode node,
DMparams *params)
```

where

svr	names the server to which the path is connected
path	specifies the data path containing the memory node
node	specifies the memory node with which the DMbufferpool will be used
params	specifies the pool requirements list

As with similar calls in other libraries, **vlDMBufferGetParams** takes as input a DMparams list initialized by **dmBufferSetPoolDefaults,** and possibly other libraries' pool requirements functions. On output, the Video Library's requirements are merged with the input requirements.

Creating a DMbufferpool

After all libraries that will use the pool have been queried for their requirements, the application can create a DMbufferpool by calling **dmBufferCreatePool**. Its function prototype is:

```
DMstatus dmBufferCreatePool(const DMparams *poolParams, DMbufferpool
*returnPool)
```

where

poolParams	specifies the requirements for the pool
returnPool	points to a location where the DMbufferpool handle will be stored

Registering a DMBufferpool With the Video Library

If the application captures video data, it specifies the DMbufferpool the memory node should use by calling **vlDMBufferPoolRegister**:

```
int vlDMBufferPoolRegister(VLServer svr, VLPath path, VLNode node,
DMbufferpool pool)
```

where

svr	specifies the server that the path is attached to
path	specifies the path containing the memory node
node	specifies the memory node
pool	specifies the pool that the memory node should use

When the video device is ready to capture a new frame or field, it will allocate a DMbuffer from the specified pool, place the field or frame in it, then send the buffer to the application.

Starting Data Transfer

To begin data transfer (for either type of buffer), use **vlBeginTransfer()**. Its function prototype is

```
int vlBeginTransfer(VLServer vlSvr, VLPath path, int count,
      VLTransferDescriptor* xferDesc)
```

where

vlSvr	names the server to which the path is connected
path	specifies the data path
count	specifies the number of transfer descriptors
xferDesc	specifies an array of transfer descriptors

Tailor the data transfer by means of *transfer descriptors*. Multiple transfer descriptors are supplied; they are executed in order. 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. VLDeviceEvent specifies an external trigger.

If *xferDesc* is NULL, then VL_TRIGGER_IMMEDIATE and VL_TRANSFER_CONTINUOUS_MODE are assumed and one transfer is performed.

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;
```

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;
```

Receiving Buffers From the Video Library

After the transfer has been started, captured video may be retrieved using **vlDMBufferGetValid**:

```
int vlDMBufferGetValid(VLServer svr, VLPath path, VLNode node,
DMbuffer* dmbuffer)
```

where

svr	specifies the server the path is attached to
path	specifies the path on which data is received from
node	specifies the memory drain node data is received from
dmbuffer	points to a location where a DMbuffer handle is stored

The DMbuffer handle returned by **vlDMBufferGetValid** is an opaque reference to the captured video. **dmBufferMapData** can be used to obtain a pointer to the actual image data so that it can be processed or written to disk. **dmBufferMapData** does not have to be called if the buffer will be directly sent to another device or library.

Sending DMbuffers to the Video Library

Applications can use **vlDMBufferPutValid** to send buffers to a video device:

```
int vlDMBufferPutValid(VLServer svr, VLPath path, VLNode node,
DMbuffer dmbuffer)
```

where

svr	specifies the server to which the path is attached
path	specifies the path on which video is sent
node	specifies the memory source node to send the buffer to
dmbuffer	specifies the buffer to send

The DMbuffer may have been obtained from another library, such as dmIC, or generated by the application itself. See Chapter 5 in the *Digital Media Programming Guide* for an explanation of how to allocate a DMbuffer from a DMbufferpool.

Freeing a DMbuffer

Once the application is done with a buffer, it should call **dmBufferFree** to indicate that it no longer intends to use the buffer. After all users of a buffer have called **dmBufferFree** on it, the buffer is considered free to be reallocated. The Video Library never implicitly releases the application's access to a buffer. Consequently, an application can send the same buffer to a memory node multiple times, or hold a captured image for an indefinite period.

Transferring Video Data Using VL Buffers

The processes for data transfer using VL buffers are as follows:

- "Creating a Buffer for Video Data"
- "Registering the VL Buffer"
- "Starting Data Transfer"
- "Reading Data From the VL Buffer"

Each process is explained separately.

Creating a Buffer for Video Data

Once you have specified frame parameters in a transfer involving memory (or have determined to use the defaults), create a VL buffer for the video data. In this case, video data is frames or fields, depending on the capture type:

- frames if the capture type is VL_CAPTURE_NONINTERLEAVED
- fields if the capture type is anything else

VL buffers provide a way to read and write varying sizes of video 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 VL buffer is created, constraints are specified that control the total size of the data segment and the number of frame or field buffers (sectors) to allocate. A head and a tail flag are automatically set in a VL 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 VL buffer is written to and all sectors are occupied, data transfer stops. The sector last written to remains locked down until it is released.

All sectors in a VL buffer must be of the same size, which is the value returned by **vlGetTransferSize()**. Its function prototype is

```
long vlGetTransferSize(VLServer vlSvr, VLPath path)
```

For example:

```
transfersize = vlGetTransferSize(vlSvr, path);
```

where *transfersize* is the size of the data in bytes.

To create a VL buffer for the frame data, use **vlCreateBuffer()**. Its function prototype is

```
VLBuffer vlCreateBuffer(VLServer vlSvr, VLPath path, VLNode node,
     int numFrames)
```

where

VLBuffer	is the handle of the buffer to be created
vlSvr	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 VL buffer
numFrames	specifies the number of sectors in the buffer (fields or frames, depending on the capture type)

For example:

```
buf = vlCreateBuffer(vlSvr, path, src, 1);
```

Table 2-10 shows the relationship between capture type and minimum VL buffer size.

Table 2-10 Buffer Size Requirements

Capture Type	**Minimum Sectors for Capture**	**Minimum Sectors for Playback**
VL_CAPTURE_NONINTERLEAVED	2	4
VL_CAPTURE_INTERLEAVED	1	2
VL_CAPTURE_EVEN_FIELDS	1	2
VL_CAPTURE_ODD_FIELDS	1	2
VL_CAPTURE_FIELDS	1	2

Note: For VGI1 memory nodes, real-time memory or video transfer can be performed only as long as buffer sectors are available to the OCTANE Digital Video device.

Registering the VL Buffer

Use **vlRegisterBuffer()** to register the VL buffer with the data path. Its function prototype is

```
int vlRegisterBuffer(VLServer vlSvr, VLPath path,
     VLNode memnodeid, VLBuffer buffer)
```

where

vlSvr	names the server to which the path is connected
path	specifies the data path
memnodeid	specifies the memory node ID
buffer	specifies the VL buffer handle

For example:

```
vlRegisterBuffer(vlSvr, path, drn, Buffer);
```

Starting Data Transfer

Start data transfer the same way as for DMbuffers; see “Starting Data Transfer” in “Transferring Video Data Using DMbuffers.”

Reading Data From the VL Buffer

If your application uses a VL buffer, use various VL calls for reading frames, getting pointers to active buffers, freeing buffers, and other operations. Table 2-11 lists the buffer-related calls.

Table 2-11 Buffer-Related Calls

Call	Purpose
vlGetNextValid()	Returns a handle on the next valid frame or field of data
vlGetLatestValid()	Reads only the most current frame or field in the buffer, discarding the rest
vlPutValid()	Puts a frame or field into the valid list (memory to video)
vlPutFree()	Puts a valid frame or field 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 2-4 illustrates the difference between **vlGetNextValid()** and **vlGetLatestValid()**, and their interaction with **vlPutFree()**.

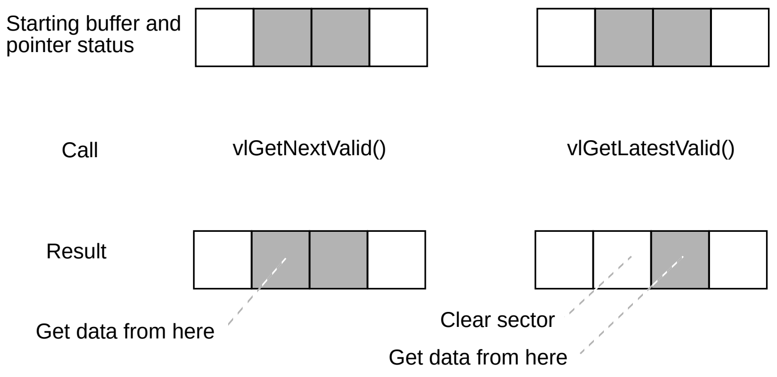


Figure 2-4 vlGetNextValid() and vlGetLatestValid()

Table 2-12 lists the calls that extract information from a buffer.

Table 2-12 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(vlSvr, path, memnode1);
vlRegisterBuffer(vlSvr, path, memnode1, buffer);
vlBeginTransfer(vlSvr, path, 0, NULL);
info = vlGetNextValid(vlSvr, buffer);
/* OR vlGetLatestValid(vlSvr, buffer); */
dataptr = vlGetActiveRegion(vlSvr, buffer, info);

/* use data for application */
…
vlPutFree(vlSvr, buffer);
```

For memory-to-video transfer, use

```
buffer = vlCreateBuffer(vlSvr, path, memnode1);
vlRegisterBuffer(vlSvr, path, memnode1, buffer);
vlBeginTransfer(vlSvr, path, 0, NULL);
buffer = vlGetNextFree(vlSvr, buffer, bufsize);
/* fill buffer with data */
…
vlPutValid(vlSvr, buffer);
```

To read 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 vlSvr, VLBuffer vlBuffer)
```

Use **vlGetLatestValid()** to read only the most current frame in the buffer, discarding the rest. Its function prototype is

```
VLInfoPtr vlGetLatestValid(VLServer vlSvr, VLBuffer vlBuffer)
```

After removing interesting data, return the buffer for use with **vlPutFree()** (video to memory). Its function prototype is

```
int vlPutFree(VLServer vlSvr, VLBuffer vlBuffer)
```

To send frames from memory to video, use **vlGetNextFree()** to get a free buffer to which to write data. Its function prototype is

```
VLInfoPtr vlGetNextFree(VLServer vlSvr, 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 vlSvr, 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.

To get DMediaInfo and Image Data from the buffer, use **vlGetActiveRegion()** to get a pointer to the active buffer. Its function prototype is

```
void * vlGetActiveRegion(VLServer vlSvr, VLBuffer vlBuffer,
     VLInfoPtr ptr)
```

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

```
DMediaInfo * vlGetDMediaInfo(VLServer vlSvr, VLBuffer vlBuffer,
     VLInfoPtr ptr)
```

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

```
DMImageInfo * vlGetImageInfo(VLServer vlSvr, VLBuffer vlBuffer,
     VLInfoPtr ptr)
```

Ending Data Transfer

To end data transfer for either VL buffers or DMbuffers, use **vlEndTransfer()**. Its function prototype is

```
int vlEndTransfer(VLServer vlSvr, VLPath path)
```

A discrete transfer is finished when the last frame of the sequence is output. The two types of memory nodes behave differently at the last frame:

- The CC1 memory source stops transferring data from main memory to the OCTANE Digital Video device, but continues to output to video the last frame transferred, which is held in a framebuffer associated with the CC1 memory node.
- The VGI1 memory nodes have no associated framebuffer and consequently emit black video output after a transfer (discrete or continuous) has been completed.

To accomplish the necessary cleanup to exit gracefully, use the following functions:

- for transfers involving memory:
  - DMbuffers: **vlDMBufferPoolDeregister()**, **vlDestroyPath()**, **dmBuffer()**
  - VL buffers: **vlDeregisterBuffer()**, **vlDestroyPath()**, **vlDestroyBuffer()**
- for all transfers: **vlCloseVideo()**

The function prototype for **vlDeregisterBuffer()** is

```
int vlDeregisterBuffer(VLServer vlSvr, VLPath path,
    VLNode memnodeid, VLBuffer ringbufhandle)
```

where

vlSvr	is the server handle
path	is the path handle
memnodeid	is the memory node ID
ringbufhandle	is the VL buffer handle

The function prototypes for **vlDestroyPath()**, **vlDestroyBuffer()**, **dmBuffer()**, and **vlCloseVideo()** are, respectively

```
int vlDestroyPath(VLServer vlSvr, VLPath path)
```

```
int vlDestroyBuffer(VLServer vlSvr, VLBuffer vlBuffer)
```

```
int vlGetFilledByNode(VLServer vlSvr, VLPath path, VLNode node);
```

```
int vlDMBufferNodeReset(VLServer vlSvr, VLPath path, VLNode node);
```

```
int vlCloseVideo(VLServer vlSvr)
```

where *vlSvr* specifies the server to which the application is attached, and *path* and *node* identify the memory node on which information is requested.

This example ends a data transfer that used a buffer:

```
vlEndTransfer(vlSvr, path);
vlDeregisterBuffer(vlSvr, path, memnodeid, buffer);
vlDestroyPath(vlSvr, path);
vlDestroyBuffer(vlSvr, buffer);
vlCloseVideo(vlSvr);
```

For DMbuffers, **vlDMBufferPoolDeregister** disassociates a DMbufferpool from a memory node. It should be called to clean up the memory node or allow a new DMbufferpool to be used after a transfer has been stopped.

Once the application is done with a DMbufferpool, the pool should be destroyed using the **dmBufferDestroyPool** call.

Example Programs

The directory */usr/share/src/dmedia/video/vl* includes a number of example programs. These programs illustrate how to create simple video applications; for example:

- a simple screen application: *simplev2s.c*

  This program shows how to send live video to the screen.
- a video-to-memory frame grab: *simplegrab.c*

  This program demonstrates video frame grabbing.
- a memory-to-video frame output *simplem2v.c*

  This program sends a frame to the video output.
- a continuous frame capture: *simpleccapt.c*

  This program demonstrates continuous frame capture.

Note: To simplify the code, these examples do not check returns. However, you should always check returns.

See Chapter 4 for a description of *eventex.c* and Chapter 7 for descriptions of *simpleblend.c* and *simplewipe.c*.

The directory */usr/share/src/dmedia/video/vl/OpenGL* contains three example OpenGL programs:

- *contcapt.c*: performs continuous capture using buffering and *sproc*
- *mtov.c:* uses the Silicon Graphics Movie Library to play a movie on the selected video port
- *vidtomem.c*: captures an incoming video stream to memory

These programs are the OpenGL equivalents of the programs with the same names in */usr/share/src/dmedia/video/vl*.

Chapter 3

Using VL Controls

Video Library (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, and horizontal sync
- specify sync source

This chapter explains

- "VL Control Type and Values"
- "VL Control Fraction Ranges"
- "VL Control Classes"
- "VL Control Groupings"

Device-independent controls are documented in */usr/include/dmedia/vl.h*. Device-dependent controls for the OCTANE Digital Video option are documented in the header files */usr/include/dmedia/vl_mgv.h* (linked to */usr/include/vl/dev_mgv.h*) and */usr/include/dmedia/vl_impact.h* (linked to */usr/include/vl/dev_impact.h*).

Note: For information on the controls used for specific nodes, see Appendix B, "OCTANE Digital Video Nodes and Their Controls." For information on controls for blending and keying, see Chapter 7, "Blending, Keying, and Transitions."

Table 3-1 lists device-independent VL controls alphabetically, along with their values or ranges.

Table 3-1 Device-Independent Controls for the OCTANE Digital Video Option

Control	Purpose	Comments
VL_BLEND_A	Input source for foreground (channel A) image	VLNode type derived from **vlGetNode()**; must be one of the source nodes
VL_BLEND_B	Input source for background (channel B) image	VLNode type derived from **vlGetNode()**; must be one of the source nodes
VL_BLEND_A_ALPHA	Input source for foreground (channel A) alpha	
VL_BLEND_B_ALPHA	Input source for background (channel B) alpha	
VL_BLEND_A_FCN	Blend function that controls mixing of foreground (channel A) signals	VL_BLDFCN_ZERO VL_BLDFCN_ONE VL_BLDFCN_A_ALPHA VL_BLDFCN_MINUS_A_ALPHA
VL_BLEND_B_FCN	Blend function that controls mixing of background (channel B) signals	VL_BLDFCN_ZERO VL_BLDFCN_ONE VL_BLDFCN_B_ALPHA VL_BLDFCN_MINUS_B_ALPHA
VL_BLEND_A_NORMALIZE	Follows Porter-Duff model (background [channel B]′ pixels premultiplied by their corresponding alphas before blending); premultiplies foreground (channel A) by alpha	1 = off
VL_BLEND_B_NORMALIZE	Follows Porter-Duff model (background [channel A]′ pixels premultiplied by their corresponding alphas before blending); premultiplies foreground (channel B) by alpha	0 = off 1 = on
VL_CAP_TYPE	Type of frame(s) or field(s) to capture	
VL_DEFAULT_SOURCE	Default source for the video path	
VL_DEFAULT_DRAIN	Default drain for the video path	
VL_FORMAT	Video format	

Table 3-1 (continued) Device-Independent Controls for the OCTANE Digital Video Option

Control	Purpose	Comments
VL_FREEZE	Data transfer freeze; suspends transfer at the drain node, used only for video out (FB node)	0 = off 1 = on
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;	Coordinates; default is 0,0
VL_PACKING	Packing of video data at source or drain	
VL_RATE	Transfer rate in fields or frames	
VL_SIZE	On VL_VIDEO nodes, the size of the video; on all other nodes, the clipped size of the video	
VL_SYNC	Sync mode	VL_SYNC_INTERNAL VL_SYNC_GENLOCK
VL_SYNC_SOURCE	Sets sync source for analog breakout box	Reference input: GEN_PORT Input 1: GEN_DIN1 Input 2: GEN_DIN2
VL_TIMING	Video timing	
VL_WINDOW	Window ID for video in a window (screen node only)	Integer
VL_ZOOM	Zoom and decimation	Screen drain nodes support 7/1, 6/1, 5/1, 4/1, 3/1, 2/1, 1/1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8. Screen source nodes support 1/1 and 1/2. Other nodes support zoom and decimation ratios of 1:1 only, that is, no zoom or decimation.

Note: For information on controls for keying, blending, or wipes, see Chapter 7. For detailed information on using VL_CAP_TYPE, VL_FORMAT, VL_OFFSET, VL_PACKING, VL_RATE, VL_SIZE, and VL_TIMING, see "Setting Parameters for Data Transfer to or From Memory" in Chapter 2.

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, BRIGHTNESS */
    VLControlClass ctlClass; /* SLIDER, DETENT, KNOB, BUTTON */
    VLControlGroup group;    /* BLEND, VISUAL QUALITY, SIGNAL, SYNC */
    VLNode node;             /* associated node */
    VLControlValueType valueType;       /* what kind of data do we have */
    int valueCount;             /* how many data items do we have */
    int numFractRanges;     /* number of ranges to describe control */
    VLFractionRange *ranges; /* range of values of control */

    int numItems;               /* number of enumerated items */
    VLControlItem *itemList;    /* the actual enumerations */
} VLControlInfo;
```

To store the value of different controls, *libvl.a* uses this struct:

```
typedef union {
    VLFraction  fractVal;
    VLBoolean   boolVal;
    int         intVal;
    VLXY        xyVal;
    char        stringVal[96];  /* beware of trailing NULLs! */
    float       matrixVal[3][3];
    uint        pad[24];        /* reserved */
} VLControlValue;

typedef struct {
    int numControls;
    VLControlInfo *controls;
} VLControlList;
```

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 information 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 range type 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 */
#define VL_CLASS_WRONLY         0x4000  /* control is write-only */
#define VL_CLASS_NO_DEFAULT     0x2000  /* don't save in default files */

#define VL_IS_CTL_RDONLY(x)     ((x)->ctlClass & VL_CLASS_RDONLY)
#define VL_IS_CTL_WRONLY(x)     ((x)->ctlClass & VL_CLASS_WRONLY)
#define VL_IS_CTL_RW(x)         (!(VL_IS_CTL_RDONLY(x) || VL_IS_CTL_WRONLY(x)))
```

The macros 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 3-2 summarizes the VL control groupings.

Table 3-2 VL Control Groupings

Grouping	Includes controls for...
VL_CTL_GROUP_BLENDING	Blending; for example, VL_BLEND_B_FCN
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_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_CLUT_COMPOSITE	Specifying properties of composite color lookup table (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

Chapter 4

Event Handling

The Video Library (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.

This chapter explains

- "OCTANE Digital Video VL Events"
- "Querying VL Events"
- "Creating a VL Event Loop"
- "Creating a Main Loop With Callbacks"

It concludes with an example illustrating a main loop and event loops.

OCTANE Digital Video VL Events

This section describes the events that the OCTANE Digital Video device generates. Each event has a standard header, which can be followed by additional data. The additional data can be accessed through the appropriate structure member of the VLEvent union, specified for each of the events listed below.

The VLEvent union and its structures are found in */usr/include/dmediavl.h*.

The standard header for a VL event contains

- int *reason*: the event ID, such as VLControlChanged
- VLServer *server*: the server from which the event originated
- VLDev *device*: the device from which the event originated
- VLPath *path*: the path on which the event originated
- uint *serial*: the serial number of the last request read from the server connection
- uint *time*: the time at which the event was generated

Note: Hardware-generated events, such as vertical retrace, are not available on pure video source-to-video drain paths. To receive these events, a path must make use of the screen, blender, framebuffer, or memory nodes. A path receives a VLBadPath error from **vlSelectEvents()** if it attempts to register for events it cannot receive.

Table 4-1 summarizes the VL events for the OCTANE Digital Video device.

Table 4-1 VL Events for the OCTANE Digital Video Device

Event	Structure	Description
VLStreamPreempted	vlstreampreempted	Generated when a path is preempted by another path that requires some resource that the first path also requires. The paths may be contending over a node (such as a video drain), a part of a node (such as a dual-link input node or one of the single-link nodes that comprise it), or other resource (such as a connector required to route a path). The preempted path is indicated by the path member of the vlstreampreempted structure. Once preempted, the path has a stream usage of VL_READ_ONLY. When the stream becomes available again, the path is downgraded to a control usage of VL_SHARE, unless control usage was at VL_READ_ONLY before the stream was preempted. In this case, the level remains at VL_READ_ONLY. A VLStreamAvailable event is delivered when the path can be set up again to a stream usage of VL_SHARE or VL_LOCK.
VLStreamAvailable	vlstreamavailable	Generated when all nodes required by a path become available for setup with a stream usage of VL_SHARE or VL_LOCK. Typically, such a path becomes available when another path that was using the nodes is set up with stream usage VL_READ_ONLY or VL_DONE_USING, or is destroyed. The path in question is indicated by the path member of the vlstreamavailable structure. VLStreamAvailable is delivered to all registered paths with a stream usage of VL_READ_ONLY. Consequently, a rare condition can occur in which several paths are set up when they receive this event, so that the last path that was set up "wins."
VLSyncLost	vlsynclost	Generated when a node on a path detects invalid timing. The path on which the timing error occurred is specified by the path member of the vlsynclost structure. Some memory nodes, such as the VGI1 memory nodes, have controls to abort a transfer when they detect invalid timing. In that case, a VLTransferFailed event is generated in its place.

Table 4-1 (continued) VL Events for the OCTANE Digital Video Device

Event	Structure	Description
VLSequenceLost	vlsequencelost	Generated when a video unit (field or frame, depending on the capture type) is dropped. The path on which the unit was dropped is specified by the path member of the vlsequencelost structure. If a group of contiguous units is dropped, only one VLSequenceLost event is generated. The client can register for VLTransferComplete events to determine when capture or playback resumes. Note that VLSequenceLost represents a “soft” error and video transfer continues on the path. This event is in contrast to VLTransferFailed, which signals a “hard” error that causes the transfer to abort. The event is delivered as soon as the missed unit is detected. Note that for VGI1 memory nodes; this event may not be generated until a valid unit is transferred.
VLControlChanged	vlcontrolchanged	Generated when a control’s value changes. In order for a path to receive this event, it must contain the node on which the control resides. The node is specified in the node member of the vlcontrolchanged structure, and the control’s ID is specified by the type member. Use vlGetControl to retrieve the new value of the control. This event is never delivered to the path causing the event, that is, the path on which vlSetControl was called. Note that the vlcontrolchanged structure contains a value member. This member is not currently used and does not contain the new value of the control.
VLTransferComplete	vltransfercomplete	Generated each time a video unit is captured or played back on a path. The video unit is a field or a frame, depending on the capture type. The path on which the event occurred is specified in the path member of the vltransfercomplete structure. This event is generated by paths containing memory nodes only. VLTransferComplete is not sent on “jack-to-jack” paths, for example, a video input to video output path.

Table 4-1 (continued) VL Events for the OCTANE Digital Video Device

Event	Structure	Description
VLTransferFailed	vltransferfailed	Generated when a catastrophic error occurs while a path is capturing or playing back a video unit. The memory transfer is halted. The path on which the failure occurred is specified by the path member of the vltransferfailed structure. Note that this event is in contrast to the VLSyncLost or VLSequenceLost events, which are generated when noncatastrophic errors are detected. This event is generated by paths containing memory nodes only. VLTransferFailed is not sent on "jack-to-jack" paths, for example, a video input to video output path.
VLEvenVerticalRetrace	vlevenverticalretrace	Generated at the vertical retrace for each even field in the video stream. The path on which the event occurred is specified by the path member of the vlevenverticalretrace structure. A path must contain a memory, screen, or blender node to receive VLEvenVerticalRetrace events.
VLOddVerticalRetrace	vloddverticalretrace	Generated at the vertical retrace for each odd field in the video stream. The path on which the event occurred is specified by the path member of the vloddverticalretrace structure. A path must contain a memory, screen, or blender node to receive VLOddVerticalRetrace events.
VLFrameVerticalRetrace	vlframeverticalretrace	Generated at the vertical retrace for each frame. The path to which the event is delivered is specified by the path member of the vlframeverticalretrace structure. A path must contain a memory, screen, or blender node to receive VLFrameVerticalRetrace events.
VLDeviceEvent	vldeviceevent	Generated when the external trigger fires. The event is delivered to all paths registered for it. The path to which an event record is delivered is specified by the path member of the vldeviceevent structure. Trigger polarity, trigger line, and other parameters controlling the trigger are specified by controls on the device node.
VLDefaultSource	vldefaultsource	Generated when a **vlSetControl()** on the VL_DEFAULT_SOURCE control changes the default video source. The new source is specified by the node member of the vldefaultsource structure. In order for a path to receive this event, it must contain the new default source node.

Table 4-1 (continued) VL Events for the OCTANE Digital Video Device

Event	Structure	Description
VLControlRangeChanged	vlcontrolrangechanged	Generated when the range for a control changes. In order for a path to receive this event, it must contain the node on which the control resides. The node is specified in the node member of the vlcontrolrangechanged structure, and the control's ID is specified by the type member.
VLControlPreempted	vlcontrolpreempted	Delivered to a path that has acquired a node with VL_SHARE control usage (the preempted path) when a path with VL_LOCK control usage (the preempting path) is set up. The preempted path retains VL_SHARE control usage, but is prevented from changing any controls while the preempting path is set up with control usage VL_LOCK. A VLControlAvailable event is sent when the controls are unlocked. The node whose controls have been locked is specified by the node member of the vlcontrolpreempted structure. The path containing the node is identified by the path member.
VLControlAvailable	vlcontrolavailable	Delivered to a path whose controls were previously preempted (see VLControlPreempted), when controls are unlocked, that is, when the control usage of the locking path is dropped to VL_SHARE, VL_READ_ONLY, or VL_DONE_USING. The node whose controls have been unlocked is specified by the node member of the vlcontrolavailable structure. The path containing the node is identified by the path member.
VLDefaultDrain	vldefaultdrain	Generated when a **vlSetControl()** changes the default video drain to VL_DEFAULT_DRAIN control. The new drain is specified by the node member of the vldefaultdrain structure. In order to receive this event, the path must contain the new default drain node.
VLStreamChanged	vlstreamchanged	Generated when the connectivity of the device is changed by **vlSetConnection()**, or by connections generated by the OCTANE Digital Video device on a path's behalf. This event is sent to all paths containing the drain node whose input has changed. Paths containing only the source node do not receive this event, nor does the path causing the connectivity change. The affected path is specified by the path member of the vlstreamchanged structure. The affected drain (node, port) pair are specified by the drnnode and drnport members. The new source (node, port) pair is specified by the srcnode and srcport members. If the source or drain (node, port) pair cannot be represented on the path because it does not contain the node in question, then the (node, port) pair has the value (VLUnknownNode, VLUnknownPort).

Querying VL Events

General VL event handling routines are summarized in Table 4-2.

Table 4-2 VL Event Handling Routines

Routine	Use
vlGetFD()	Retrieves a file descriptor for a VL server
vlNextEvent()	Obtains the next event; blocks until the next event from the queue is obtained
vlCheckEvent()	Like a nonblocking **vlNextEvent()**, 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()	Retrieves 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 2-1 in Chapter 2 summarizes VL event masks.

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);
```

The VLEvent structure returned by vlNextEvent or vlCheckEvent identifies the type of event that occurred and provides additional information on the event; for example, the VLControlChanged event, accompanied by the node on which the control resides and by the new value of the control. These additional pieces of information can be obtained through the members of the VLEvent union corresponding to each event.

Event masks can be Or'ed together. For example:

```
vlSelectEvents(svr, path, VLTransferCompleteMask |
               VLTransferFailedMask);
```

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 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 that are called when a particular VL event arrives, use **vlAddCallback()**. Its function prototype is

```
int vlAddCallback(VLServer vlServer, VLEvent * event,
     void * clientdata, VLEventMask events,
     VLCallbackProc callback, void *clientData)
```

Example 4-1 illustrates the use of **vlAddCallback()**.

Example 4-1 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 */
   vlAddCallback(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 */
void
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

```
int vlRemoveCallback(VLServer vlServer, VLPath * path,
       VLEventMask events, VLCallbackProc callback, void
       *clientData)
```

```
int vlRemoveAllCallbacks(VLServer vlServer, VLPath * path, VLEventMask events)
```

The functions **vlAddHandler()** and **vlRemoveHandler()** are analogous to **vlAddCallback()** and **vlRemoveCallback()**, respectively. Use them for non-VL events.

In */usr/share/src/dmedia/video/vl*, the example program *eventex.c* illustrates how to create a main loop and event loops.

Caution: To simplify the code, this example does not check returns. You should, however, always check returns.

Chapter 5

Managing Connections

You can use the Video Library to set up complex paths in OCTANE Digital Video programs. Connections obey stream-usage levels set with **vlSetupPaths()**. Usage is drain-centric: the usage levels of the path(s) using the drain node serve as the usage level of the connection.

The functions **vlSetConnection()** and **vlGetConnection()** manipulate connections:

- **vlSetConnection()** sets a connection between a source pair (node, port) pair and a drain pair (node, port).
- **vlGetConnection()** returns the set of connections entering or leaving a node or port.

This chapter explains

- "Specifying Connectivity"
- "Avoiding Dynamic Switching Problems"
- "Using the Internal Video Sync Signal"

Specifying Connectivity

The Video Library infers the connections on a path if **vlBeginTransfer()** is called and no drain nodes have been connected using **vlSetConnection()**. This situation simplifies application development for simple paths and supports the existing set of applications that do not use **vlSetConnection()**

Thus, the use of **vlSetConnection()** to specify the path connectivity is optional. The following rules are used in determining the connections:

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

Because existing connections are preserved, **vlSetConnection()** can be used to override part of the default routes, as long as all drain nodes remain unconnected.

The function prototype of **vlSetConnection()** is

```
int vlSetConnection(VLServer vlSvr, VLPath path, VLNode source_node, VLPort
source_port, VLNode drain_node, VLPort drain_port, VLBoolean preempt)
```

where

vlSvr	names the server to which the path is connected.
path	specifies the data path whose connectivity is being changed.
source_node	names the node that data will flow from. This is either a source or internal node.
source_port	specifies the port on the source node to use.
drain_node	specifies the node that data will flow to. This is either a drain or internal node.
drain_port	specifies the port on the drain node to use.
preempt	specifies whether other paths should be preempted (TRUE) or not (FALSE) in order to route the connection.

Connections must be specified only if

- the path contains multiple internal nodes; in this case, the ordering of the internal nodes is ambiguous and may not be inferred properly by the Video Library
- the default connections, described below, are not the ones that the application desires
- the application wants to change a path's topology after the path has started transferring (note that the change in hardware route may cause a timing glitch in the video stream, depending on the device)

Connections are set up one at a time using **vlSetConnection()** and take effect either immediately or at the next vertical interval, depending on the device. In other words, if **vlSetConnection()** completes successfully, the hardware connection has been established.

Paths may be preempted in order to set a connection since scarce connector resources may be required to route a connection from the source (node, port) to the drain (node, port). The ability to preempt a path follows the rules for stream usage defined with **vlSetupPaths()**. The ability to set a connection, as opposed to only getting (retrieving) it, follows the rules for control usage.

This example fragment sets the blender node's foreground pixel input, that is, input A, to come from the framebuffer node output:

```
if (vlSetConnection(vlSvr, path, fb_node, VL_IMPACT_PORT_PIXEL_SRC,
blender_node, VL_IMPACT_PORT_PIXEL_DRN_A, FALSE) < 0)
{
    vlPerror(_progName);
    exit(1);
}
```

If **vlSetConnection()** returns with -1, an error has occurred. In addition to the standard error codes, the following have special meaning for **vlSetConnection()**:

VLNoRoute	No physical route could be found from the source to the drain.
VLPathInUse	A physical route exists between the source and drain, but the required connector resources are in use. The application has requested that no paths be preempted by specifying FALSE as the *preempt* parameter, or another path has the resources locked.

Getting Connections

Use **vlGetConnection()** to retrieve the connections originating or terminating at a given (node, port). Its function prototype is

```
int vlGetConnection(VLServer vlSvr, VLPath path, VLNode node, VLPort
port, VLNode *nodelist, VLPort *portlist, int *n)
```

where

vlSvr	specifies the server the application is connected to
path	specifies the path whose connectivity is being checked
node	specifies the node on the path
port	specifies the port on the node
nodelist	is an array of VLNode where the connected nodes will be returned
portlist	is an array of VLPort where the connected ports are returned
**n*	specifies the size of the nodelist and portlist arrays; on exit, *n is updated to reflect the actual number of elements filled in

On successful exit, each (nodelist[i], portlist[i]) pair specifies one connection to (node, port). If port is a source port, then (nodelist[i], portlist[i]) specifies the drain ports it broadcasts to. If the port is a drain port, then (nodelist[0], portlist[0]) specifies the input. Because a drain port can have only one input, only the first entry is used.

If **vlGetConnection()** returns with -1, an error has occurred. In addition to the generic error codes, the error code VLNotEnoughSpace has special meaning to **vlGetConnection()**. It indicates that the array size, **n*, is too small to contain the list of connections. **n* is updated to reflect the required array length.

The fragment in Example 5-1 illustrates **vlGetConnection()**.

Example 5-1 **vlGetConnection()** Example

```
/*
 * Connect to the video daemon
 */
svr = vlOpenVideo(““);

/*
  * Acquire video source and drain nodes.
  */
srcnode = vlGetNode(svr, VL_SRC, VL_VIDEO, VL_ANY);
drnnode = vlGetNode(svr, VL_DRN, VL_VIDEO, VL_ANY);

/*
 * Create a path with these nodes.
 */
path = vlCreatePath(svr, VL_ANY, srcnode, drnnode);

/*
 * Connect the two nodes. Since this is a simple path with obvious
 * video routing, this step is optional. If the path contained
 * multiple internal nodes or the video route was not obvious, then
 * we would need to explicitly state the connections.
 */
vlSetConnection(svr, path, srcnode, VL_IMPACT_PORT_PIXEL_SRC, drnnode,
VL_IMPACT_PORT_PIXEL_DRN, FALSE);

/*
 * Begin the transfer
 */
vlBeginTransfer(svr, path, 0, NULL);
```

Figure 5-1 diagrams the OCTANE Digital Video board architecture and shows which data can flow between blocks. An ellipse joining two links indicates that the two links can be used either as separate nodes or as a single dual-link mode.

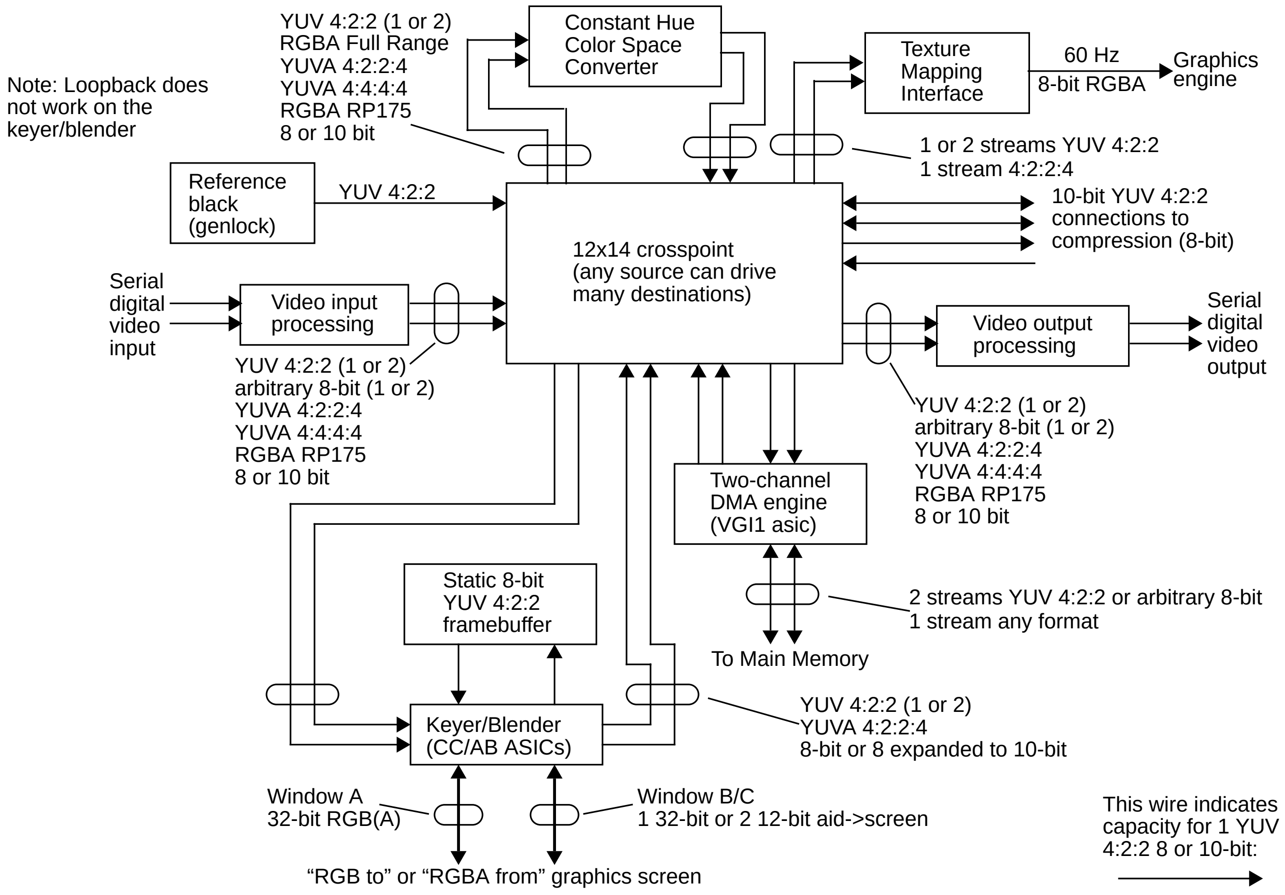

Note: Loopback does not work on the keyer/blender
YUV 4:2:2 (1 or 2)
RGBA Full Range
YUVA 4:2:2:4
YUVA 4:4:4:4
RGBA RP175
8 or 10 bit
Constant Hue Color Space Converter
Texture Mapping Interface
60 Hz
8-bit RGBA
Graphics engine
1 or 2 streams YUV 4:2:2
1 stream 4:2:2:4
Reference black (genlock)
YUV 4:2:2
10-bit YUV 4:2:2 connections to compression (8-bit)
12x14 crosspoint (any source can drive many destinations)
Serial digital video input
Video input processing
Video output processing
Serial digital video output
YUV 4:2:2 (1 or 2)
arbitrary 8-bit (1 or 2)
YUVA 4:2:2:4
YUVA 4:4:4:4
RGBA RP175
8 or 10 bit
YUV 4:2:2 (1 or 2)
arbitrary 8-bit (1 or 2)
YUVA 4:2:2:4
YUVA 4:4:4:4
RGBA RP175
8 or 10 bit
Two-channel DMA engine (VGI1 asic)
2 streams YUV 4:2:2 or arbitrary 8-bit
1 stream any format
To Main Memory
Static 8-bit YUV 4:2:2 framebuffer
Keyer/Blender (CC/AB ASICs)
YUV 4:2:2 (1 or 2)
YUVA 4:2:2:4
8-bit or 8 expanded to 10-bit
Window A
32-bit RGB(A)
Window B/C
1 32-bit or 2 12-bit aid->screen
"RGB to" or "RGBA from" graphics screen
This wire indicates capacity for 1 YUV 4:2:2 8 or 10-bit:

Figure 5-1 Hardware Representation

Figure 5-2 is a software model of OCTANE Digital Video node connectivity. Each multiplexer (mux) is a full crossbar: any source can simultaneously broadcast to all destinations. An ellipse joining two links indicates that the two links can be used either as separate nodes or as a single dual-link mode.

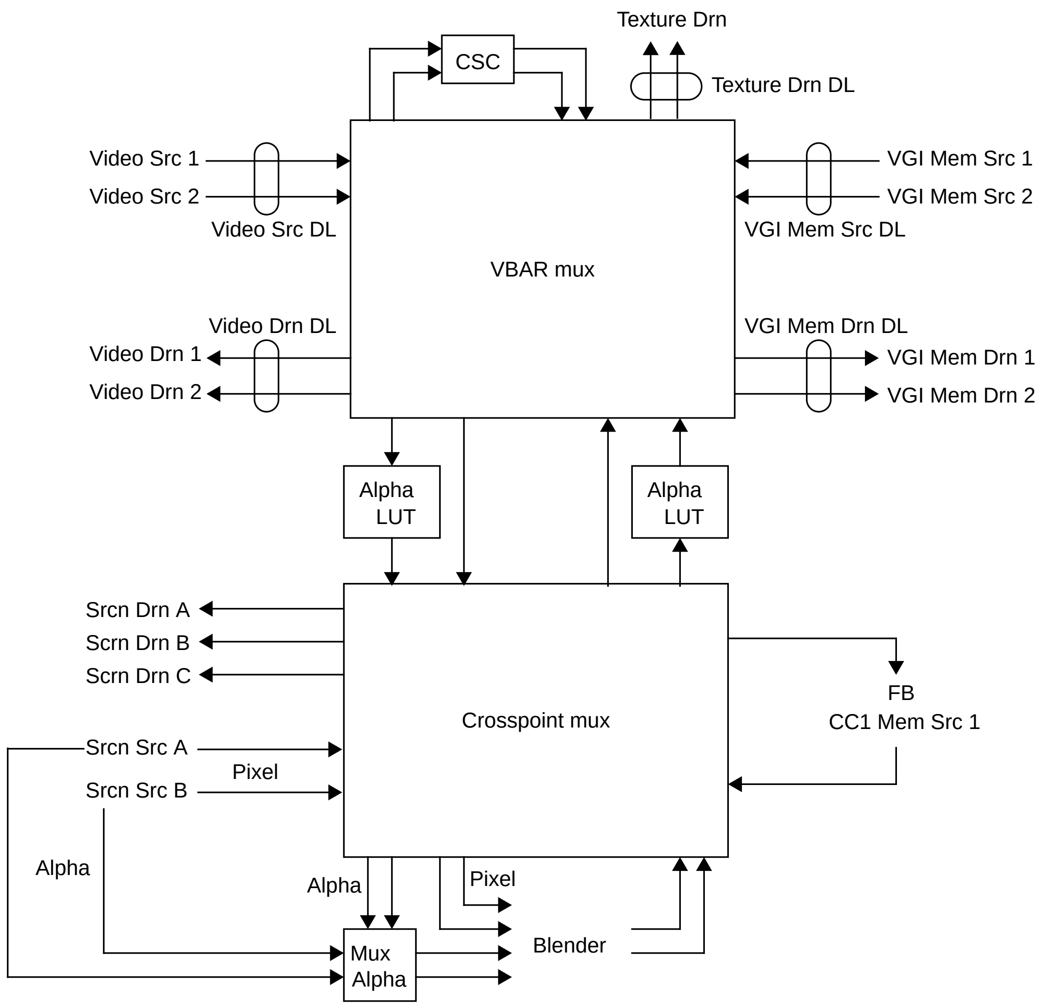


Figure 5-2 Software Representation

The link with the alpha is called the pixel-alpha link, since it can carry pixel or alpha streams. See the VL_MGV_ALPHA_LUT control in "VL_DEVICE" in Appendix B for an explanation of this LUT.

- At most two video streams can flow from the VBAR mux to the crosspoint mux, and at most two video streams can flow from the crosspoint mux to the VBAR mux. Attempts to set up paths that require more cross-mux streams result in VLPathInUse return codes, or in the path(s) using the links to be preempted.
- Because only one link in each direction has a specific alpha LUT, at most one alpha stream can be routed into and at most on one alpha stream routed out of the crosspoint mux.
- If a path uses a pixel-alpha link for a pixel stream and locks it (that is, sets stream usage to VL_LOCK), no alpha stream can be routed in that direction.

The allocation of the links is as follows:

- If the target is a blender alpha input, the pixel-alpha link is allocated.
- If the target is anything other than a blender alpha input, then the pixel link is used if it is available. Otherwise, the pixel-alpha link is used.

The dual-link video and memory nodes are composed of their single-link counterparts. Consequently, when a dual-link node is used, its single-link counterparts become unavailable. For example, when the dual-link video source is in use, video source 1 and video source 2 become unavailable.

Contention for the component single-link resources by the dual-link and single-link nodes follow normal preemption rules. A path using a dual-link node with stream usage VL_SHARE can be preempted by another path using one of its single-link constituents with stream usage VL_SHARE or VL_LOCK.

The CC1 memory source node and the framebuffer node share the same processing element. Consequently, only one can be in use at a time.

Unlike most sources, which feed directly into one of the muxes, the alpha from a screen source node can be fed only to blender alpha input. This restriction is enforced both by the VL_BLEND_A_ALPHA and VL_BLEND_B_ALPHA controls as well as by **vlSetConnection()**.

Note: When a screen source is designated as source A alpha, the blender can extract alpha from it, because a screen source "contains" alpha. Otherwise, any other pixel stream is 4:2:2. If a source other than screen is designated as source A alpha, the blender uses the Y value as alpha and discards the U and V values.

Avoiding Dynamic Switching Problems

vlSetConnection() allows path topology to be changed while the path is transferring, with certain restrictions. This section explains what connections should and should not be switched during a transfer (dynamic switching). Possible failures include the following:

- Picture glitch: Dynamic switching causes the output for one field to consist partly of the field before the switch (usually the top part), the field after the switch (usually the bottom part), and potentially a few garbage pixels at the transition. This glitch is due to the lack of vertical boundary resampling of the switching. However, there is no timing or synchronization glitch.
- Timing glitch: Dynamic switching causes a temporary loss of synchronization. Picture monitors can tear, VTRs can slew, and downstream equipment hiccups in one way or another but eventually recovers. This glitch is unacceptable for any "live" situation.

In general, a glitch ripples downstream to other nodes. Changing video standards (PAL/NTSC or square pixel/CCIR601) always causes a timing glitch. Table 5-1 shows the effects of changing various controls. Glitches caused by changes in the video routing are explained more fully in the following section.

Table 5-1 Dynamic Effects of Various Video Data Path Controls

At Video Node	Timing Glitch Possible Causes	Picture Glitch Possible Causes	Causes No Glitch
Serial digital video output	**Synchronizer bypass, horizontal phase, genlock reference sources**	Crosspoint selection to video out; various blanking	Freeze
Serial digital video input	**External interruption, autophase mode**	Truncation or rounding	N/A
VGI1 memory drain	**VBAR select to VGI1**	Truncation or rounding	N/A
VGI1 memory source	**VGI1 timing source**	N/A	N/A
Blender and screen source	**VBAR to crosspoint mux**	Alpha LUT select, various blanking and rounding, internal crosspoint	Keyer/blender controls

As indicated in Figure 5-1, the OCTANE Digital Video option has timing restrictions for the crosspoint mux and for the VGI1 memory source.

Crosspoint Mux Timing Restrictions

The two outputs of the VBAR mux going into the crosspoint mux must be locked to within six pixels of each other if they are to be blended or displayed in windows B and C. To do this, make sure that the two video input channels are locked by enabling the input autophaser: set the VL_MGV_AUTO_PHASE to a value other than VL_MGV_AUTOPHASE_OFF. (Autophasing locks the video inputs to each other if, on input, their phase differences are not too great.)

Note: "VL_DEVICE" in Appendix B describes the use of these controls.

When one of the VBAR-to-crosspoint links is not in use, the following is done:

- If the pixel-alpha link is not in use, it is set to the same input as the pixel link.
- If the pixel link is not in use, it is set to the same input as the pixel-alpha link.
- If neither links is in use, the output of the black generator is sent to both links.

This arrangement guarantees that if only one link is in use, the two links are locked. If two links are in use, then the application should ensure that the inputs are locked by enabling the input autophaser.

Note: The autophaser can affect only the two serial digital inputs.

Sending a video signal with bad timing into the crosspoint mux from the VBAR mux may cause the nodes attached to the crosspoint mux to operate inconsistently. These nodes derive master timing from the signal on the pixel-alpha link.

VGI1 Memory Source Timing Restrictions

The VGI1 memory sources derive their timing from the genlock source, or from an internal black generator if genlock is disabled. Consequently, changing controls associated with the genlock source may result in a transfer error on the memory node. Changing the autophase mode, for example, causes a timing glitch in the video stream.

Disconnecting the genlock source while a transfer is in progress causes the transfer to fail.

Using the Internal Video Sync Signal

Internal Video Sync refers to a synchronization signal produced or consumed by some audio and video devices. The purpose of the signal is to ensure that simultaneous audio and video signals are precisely synchronized.

This section explains

- “Internal Video Sync Producers and Consumers”
- “Setting the Internal Video Sync Signal Producer”

Internal Video Sync Producers and Consumers

While there may be multiple consumers of the Internal Video Sync signal, there can be only one Internal Video Sync producer (master of the Internal Video Sync line) in a system at any time. Table 5-2 lists Silicon Graphics options that produce or consume the Internal Video Sync signal.

Table 5-2 Internal Video Sync Signal Producers and Consumers

Producer	Consumer
Octane Digital Video	InfiniteReality™ graphics
Octane Compression	Radical Baseline Audio
Radical Audio	Radical Audio
DIVO digital video option for Origin2000™/Onyx2™	

When the kernel initializes, it generates a list of Internal Video Sync devices but does not set any as the master of the Internal Video Sync line.

A producer must call **ksync_attach()** from its attach routine. This call adds the producer to the hardware graph and stores the arguments passed into the soft state on the new vertex. Producers must implement the KS_CONTROL_IOCTL call, which takes a single integer argument, 0 for off (KSYNC_OFF) and 1 for on KSYNC_ON).

A consumer need only call **ksync_attach()** and implement the KS_CONTROL_IOCTL. To handle Internal Video Sync disruption, a consumer should register a callback function in **ksync_attach()**.

Setting the Internal Video Sync Signal Producer

Routines can use two Internal Video Sync calls, **ksyncstat()** and **ksyncset()**. **ksyncstat()** returns a list of Internal Video Sync-capable devices in the system. The devices are given as node names, not full pathnames; for example:

```
struct kstat_s ks_statbuf[64]
int i;

// Read system ksync configuration
ksyncstat( ks_statbuf, 64 );

// Find current Master
for( i=0; ks_statbuf[i].kName[0] != 0; i++ ) {
    if ( ks_statbuf[i].kFlags & KsyncIsProducer )

       // name of current master is in ks_statbuf[i].kName
}

// Search for potential producers..
for(i=0; ks_statbuf[i].kName[0] != 0; i++ ) {
    if( ks_statbuf[i].kFlags & KsyncProducerCapable ) {
            // found a producer, name is
            // in ks_statbuf[i].kName
    }
    else if ( ks_statbuf[i].kFlags & KsyncConsumerCapable ) {
            // found a consumer, name is
            // in ks_statbuf[i].kName
    }
}
```

The structure for **ksyncstat()** is as follows:

```
/*
       ** ksync flag values
       */

       #define KsyncIsProducer         0x1
       #define KsyncProducerCapable    0x2
       #define KsyncConsumerCapable    0x4
       #define KsyncActive             0x8

typedef struct {
           char        kName[ 64 ];
           int         kFlags;
       } kstat_t;

       int     ksyncstat(
                       kstat_t         *buf,
                       int             bufSz );        /* in bytes */
```

The buffer pointed to by *buf* is filled with as many kstat_t structures as there are Internal Video Sync devices on the system, or as many as the buffer holds. The element *kName* is the name of the device node on the hardware graph. Note that this name is the node name and not the full pathname.

ksyncset() causes a device to begin producing the Internal Video Sync signal. This call takes a string as an argument, for example:

```
ksyncset("Digital Video");
```

This example specifies a device. If another device is already producing the signal, that device immediately stops producing it and the device specified in the call begins producing it.

```
ksyncset("None");
```

Specifying None has the effect of turning off the Internal Video Sync signal. Also, if a device is specified that is not active in the system, Internal Video Sync signal generation is turned off and an error message is produced.

```
ksyncset(ks_statbuf[3].kName);
```

If the string corresponds to a string returned by **ksyncstat()**, and that name corresponds to a potential producer, that device becomes the new Internal Video Sync master. If there are no such correspondences, all producers are shut off. Using the string None (or any string that does not correspond to a potential producer) also shuts off all producers.

The Internal Video Sync feature is also implemented as a panel. This feature is incorporated into *vcp* and *apanel* as well, accessible in the Utilities menu.

Chapter 6

Video Real-Time Capture and Playback

The OCTANE Digital Video VGI1 memory nodes are capable of full video-rate capture and playback to the Video Library buffers. This chapter explains how to optimize capture or playback to system memory or disk.

- "Video Library Buffers"
- "Caching"
- "Buffer Alignment"
- "Direct I/O to Disk"
- "syssgi"
- "Asynchronous I/O"
- "Capture and Playback Examples"

Video Library Buffers

Data transfer between the VL and an application takes place through a DMbuffer or VL buffer. When the OCTANE Digital Video option transfers data from the application to the Video Library, the application retrieves an empty buffer using **vlGetNextFree()**. After placing data in the buffer, the application marks it as valid using the **vlDMGetValid()** or **vlPutValid()** routine. When the video device is finished reading from the buffer, it marks the buffer as free. For more details on the role of buffers in data transfer, see "Transferring Video Data to and From Devices" in Chapter 2.

Caching

To mark a DMbuffer as cacheable or not, use **dmBufferSetPoolDefaults()**; for VL buffers, use the **vlBufferAdvise()** routine to mark a VL buffer. They have the following prototypes:

```
int vlBufferAdvise(VLBuffer buffer, int advice)
```

where

buffer	specifies the ring buffer to be advised
advice	specifies the type of advisory being made: • VL_BUFFER_ADVISE_NOACCESS marks the buffer as non-cacheable • VL_BUFFER_ADVISE_ACCESS marks the buffer as cacheable

Marking the buffer non-cacheable indicates that the CPU cache does not have to be flushed or invalidated when data is read or written to system memory via DMA. However, any access to the buffer through the CPU must then bypass the cache and must always go to system memory. This arrangement can severely degrade the performance of an application that directly manipulates the video data.

Consequently, marking a buffer cacheable or noncacheable is application-dependent. In general:

- If the application manipulates the data, even if it is only to copy the data into or out of another region of system memory, the buffer should be set cacheable. This setting is the default for a VL buffer.
- If the application does not manipulate data, and all transfer is done strictly through DMA, then performance is optimized by setting the buffer to noncacheable. This is the case, for example, when video is read into a buffer and then written directly to disk with raw or direct I/O.

  Note: If raw or direct I/O is not used, the data is first copied into the filesystem cache. In that case, the buffer should be kept cacheable.

Buffer Alignment

The performance of the **memcpy()** and **bcopy()** routines is greatly affected by the alignment of the source and destination buffers. For copy operations between buffers with the same alignment, throughput is approximately 400% greater than between buffers with mismatched alignments. For **memcpy()** and **bcopy()**, the source and target buffers can be considered aligned if the following condition is met:

```
(src % 4) == (dest % 4)
```

In other words, the source and destination buffer addresses are equally distant from a word boundary.

Because the VL buffers used with the OCTANE Digital Video device are page-aligned, performance is maximized if the application's buffers are word-aligned. Note that the memory allocation routines such as **malloc()** return double-word (64-bit aligned) buffers. DMbuffers are guaranteed to be double-word aligned. All buffers received from the VL are guaranteed to be page-aligned, but not all DMbuffers are guaranteed to be page-aligned.

Direct I/O to Disk

Capture or playback from a disk subsystem can be greatly improved by using direct I/O. Direct I/O bypasses the filesystem's buffer cache, eliminating a data copy and other overhead. The buffer can also be marked noncacheable, yielding further performance gains.

Because the filesystem cache is bypassed, device buffer alignment and block size restrictions fall onto the application. These restrictions can be obtained using

```
fcntl(int fd, F_DIOINFO, struct dioattr *dioattr)
```

The device can, for example, require that the buffer be page-aligned. Disk devices usually require that the buffer's size be a multiple of 512 bytes (the disk sector size), or a multiple of the stripe size.

In addition, device performance can be improved with certain alignments or sizes. For example, a device operating on a non-page-aligned VL buffer can internally break the request into a nonaligned part and an aligned part, yielding the overhead of two requests instead of one. In striped disk subsystems, performance is usually improved by reading or writing entire stripes at a time.

VL buffer elements used with the OCTANE Digital Video device are always page-aligned, which satisfies the alignment constraints of most devices. DMbuffer alignment, on the other hand, is a union of all requested alignments; see “Using Buffers” in Chapter 2.

The VL_MGV_BUFFER_QUANTUM control is provided so that an application can specify the block size that should be applied to a video unit. (The video unit is a field or frame, depending on the capture type.) For example, setting this control to 512 rounds the frame or field size, as reported by **vlGetTransferSize()**, up to a multiple of 512. This control should be set to a multiple of the block size returned by *fcntl(fd, F_DIOINFO, ...)*, or to the optimal block size for the device.

When VL_MGV_BUFFER_QUANTUM is set to a value other than 1, the video data is padded at the end with random values. Consequently, it is important to use the same value for VL_MGV_BUFFER_QUANTUM on capture and on playback. Making the value the same can be a problem if a file is copied from one device to another with a different allowable block size. It is recommended that the control be set to a common multiple of the allowable sizes. For example, 4096 satisfies most devices. Otherwise, the file may need to be reformatted.

syssgi

Some of the standard I/O routines support files sizes only up to 2 GB because file position is expressed as a signed integer. **lseek**, for example, only operates up to a 2 GB range. (Note that it is possible to use the **read** or **write** system calls to read or write past the 2 GB mark, up to the filesystem size).

The **syssgi** system call can be used to read or write raw disk partitions greater than 2 GB when used with the following parameters:

```
int syssgi(int request, int fd, char *data, int blockoffset, int numblocks)
```

where

request	is SGI_READB for a read operation or SGI_WRITEB for a write operation
fd	is a file descriptor of a character special device, as obtained by the **open** system call
data	points to the buffer to be written from or read to
blockoffset	is the block position where reading or writing should commence
numblocks	is the number of blocks to read or write starting at blockoffset

Note that **syssgi** operates in units of device blocks as opposed to bytes. For disk subsystems, a block is usually 512 bytes, allowing 2^{40} bytes of disk space to be addressed.

As with direct I/O, the application is responsible for ensuring that the data buffer is properly aligned and that block size constraints are followed.

Asynchronous I/O

Asynchronous I/O allows an application to process multiple read or write requests simultaneously. On Silicon Graphics platforms, asynchronous I/O is available through the *aio* facility. The *aio64* facility additionally supports 64-bit file sizes and offsets.

Because multiple I/O requests might be outstanding when asynchronous I/O is used, the round-trip delay between making a request, having it serviced, and issuing another request is removed. Asynchronous I/O also eliminates any process-scheduling delay between these steps. In addition, the device being read from or written to might be able to optimize performance by carrying out the requests simultaneously.

For VL buffers only, keep the following points in mind when using asynchronous I/O:

- The VL buffer is a first-in first-out mechanism. When putting a buffer element back into the buffer using **vlPutValid()**, the "oldest" element retrieved by **vlGetNextFree()** is used. There is no way to specify that a different element should be used.
- Because asynchronous I/O operations can complete out of order, the application may need to keep a list of filled elements. When the oldest element is filled, the application can then call **vlPutValid()** to place it back into the buffer, and check to see if any other elements are also ready.
- The same restriction applies to **vlPutFree()** for elements obtained with **vlGetNextValid()** or **vlGetLatestValid()**.

Caution: Software conversion can severely degrade capture or playback performance.

Capture and Playback Examples

The following examples of real-time capture and playback are available in */usr/share/src/dmedia/video/vl*:

- *vidtodsk*: video to disk using direct I/O (up to the disk subsystem rate)
- *dsktovid*: disk to video using direct I/O (up to the disk subsystem rate)
- *vidtodsk_aio*: video to disk using asynchronous and direct I/O (up to the disk subsystem rate)
- *dsktovid_aio*: disk to video using asynchronous and direct I/O (up to the disk subsystem rate)

Chapter 7

Blending, Keying, and Transitions

This chapter explains how to combine video frame information and computer-generated graphics on the Indigo^2 workstation. Use the VL and the OCTANE Digital Video board 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.

The choice "Blend/Wipe Node" in the Pro menu of the panel *vcp*, a graphical user interface for VL and the OCTANE Digital Video board, provides convenient access to blending, keying, and transition controls.

This chapter explains

- "The Blender Node"
- "Keying"
- "The Keyer"
- "VL Blending Examples"

The Blender Node

Blending takes place in the VL's internal *blender node*, which mixes the foreground and background video signals by applying a blend function to the foreground and background pixels.

The blender node is supplied by four independent inputs:

- pixel from a foreground source (A)
- the alpha value for source A
- pixel from a background source (B)
- the alpha value for source B

Figure 7-1 diagrams the blender node.

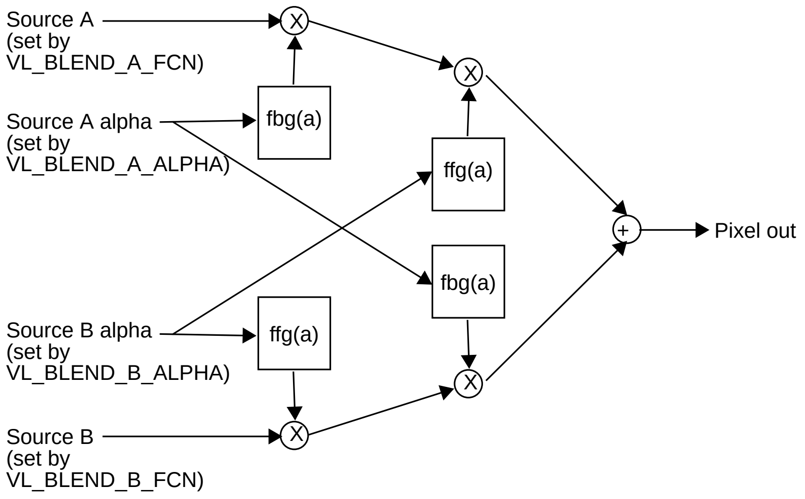

Source A (set by VL_BLEND_A_FCN)
Source A alpha (set by VL_BLEND_A_ALPHA)
Source B alpha (set by VL_BLEND_B_ALPHA)
Source B (set by VL_BLEND_B_FCN)
fbg(a)
ffg(a)
ffg(a)
fbg(a)
X
+
Pixel out

Figure 7-1 Blender Node

The blender node has four multiplier stages, indicated by ⊗ in Figure 7-1, and one adder stage, indicated by ⊕. The values in the four multiplier stages are based on the blending functions selected and on the input normalization controls.

Of the four inputs shown in Figure 7-1, two have alternate sources. The OCTANE Digital Video keyer is hard-wired to alpha source A and the flat-background generator is hard-wired to pixel source B, as diagrammed in Figure 7-2.

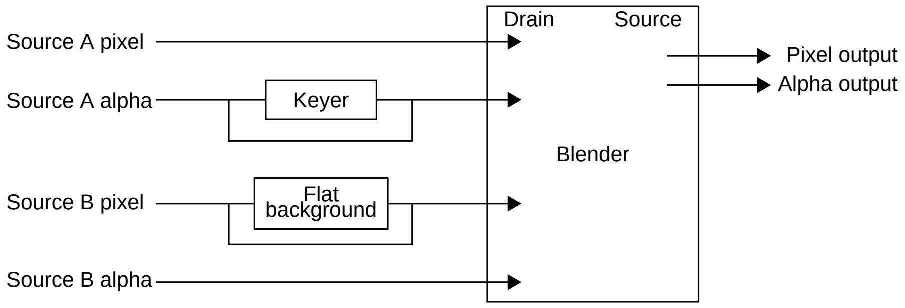


Figure 7-2 Keyer and Flat-Background Generator Locations on Source Nodes

The keyer produces an alpha stream from a pixel stream, generating a key for each pixel in each source node. It is described in detail in "The Keyer" later in this chapter. The flat-background generator sets the background pixel stream (source B pixel) to a default background color or to another color.

Note: When a screen source is designated as source A alpha, the blender can extract alpha from it, because a screen source "contains" alpha. Otherwise, any other pixel stream is 4:2:2. If a source other than screen is designated as source A alpha, the blender uses the Y value as alpha and discards the U and V values.

The rest of this section explains

- setting up the blender node
- setting normalization
- setting and turning off flat background
- adding shadows

Setting Up the Blender Node

Figure 7-3 diagrams setting up the blender node.

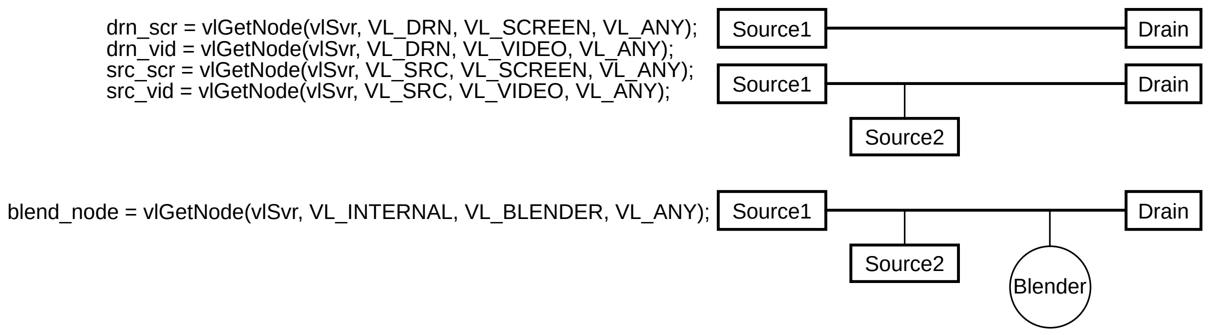

drn_scr = vlGetNode(vlSvr, VL_DRN, VL_SCREEN, VL_ANY);
drn_vid = vlGetNode(vlSvr, VL_DRN, VL_VIDEO, VL_ANY);
src_scr = vlGetNode(vlSvr, VL_SRC, VL_SCREEN, VL_ANY);
src_vid = vlGetNode(vlSvr, VL_SRC, VL_VIDEO, VL_ANY);
blend_node = vlGetNode(vlSvr, VL_INTERNAL, VL_BLENDER, VL_ANY);
Source1
Drain
Source1
Drain
Source2
Source1
Drain
Source2
Blender

Figure 7-3 Setting Up the Blender Node

The blender node is created with the **vlGetNode()** function. The code fragment in Example 7-1 sets up source, drain, and blender nodes. Notice that the drain nodes are set up before the source nodes.

Example 7-1 Setting Up Source, Drain, and Blender Nodes

```
/* variable definitions */
{
 VLServer vlSvr;
 VLPath path;
 VLNode drn_scr, drn_vid, src_scr, src_vid, blend_node;
}

/* Open a video device */
vlSvr = 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, VL_ANY);

/* Set up internal blending node */
blend_node = vlGetNode(vlSvr, VL_INTERNAL, VL_BLENDER,
                       VL_ANY);
```

Table 7-1 summarizes the generic VL blending controls. For all these controls, access is GST:

- G: The value can be retrieved through **vlGetControl().**
- S: The value can be set through **vlSetControl()** while the path is not transferring.
- T: The value can be set through **vlSetControl()** while the path is transferring.

Table 7-1 General Blender Controls

Control	Values	Selects
VL_BLEND_A_FCN type *intVal*	VL_BLDFCN_ZERO VL_BLDFCN_ONE (default) VL_BLDFCN_B_ALPHA VL_BLDFCN_MINUS_B_ALPHA	Blend function that controls mixing of foreground signals, and, with VL_BLEND_B_FCN, provides 16 possible blends of Pixel A and Pixel B, although not all are useful. This control is superseded by **vlSetConnection()**.
VL_BLEND_B_FCN type *intVal*	VL_BLDFCN_ZERO VL_BLDFCN_ONE VL_BLDFCN_A_ALPHA VL_BLDFCN_MINUS_A_ALPHA (default)	Blend function that controls mixing of background signals; see VL_BLEND_B_FCN for more information. This control is superseded by **vlSetConnection()**.
VL_BLEND_A type *intVal*	VLNode type, derived from **vlGetNode()**; must be one of the two pixel source nodes	Sets input source A pixel (foreground).
VL_BLEND_B type *intVal*	Same as for VL_BLEND_A	Sets input source B pixel (background).
VL_BLEND_A_ALPHA type *intVal*	VLNode type, derived from **vlGetNode()**; must be one of the two alpha source nodes	Sets input source A alpha (foreground). If this control is set to a screen node, screen alpha is used.
VL_BLEND_B_ALPHA type *intVal*	Same as for VL_BLEND_A_ALPHA	Sets input source B alpha (background). If this control is set to a screen node, screen alpha is used.
VL_BLEND_A_NORMALIZE type *boolVal*	(0,1) 0 = off (not supported), 1 = on (default)	Selects normalization for A pixel and alpha streams. The pixel and alpha of each stream are applied before the blend operation. Follows Porter-Duff model (background pixels premultiplied by their corresponding alphas before blending)

Table 7-1 (continued) General Blender Controls

Control	Values	Selects
VL_BLEND_B_NORMALIZE type *boolVal*	(0,1) 0 = off, 1 = on (default)	If set to TRUE, the pixel and alpha of each stream are applied before the blend operation. For screen inputs, set this control to FALSE if you are in VL_MGV_KEYERMODE_NONE mode; alpha has been applied once by the graphics in most cases. Otherwise, leave set to TRUE. Many special effects can be done by altering this control. Follows Porter-Duff model.

Some of the operations are not very useful, for example VL_BLDFCN_ONE, VL_BLDFCN_ONE adds the two images; the hardware clips pixels that are too bright.

Note: When sending the blender output to video, it is best to blank the chroma.

Setting Normalization

You can compose the 12 standard Porter-Duff operations by combining the values of VL_BLEND_A_FCN and VL_BLEND_B_FCN. Figure 7-4 gives some examples of compositing, assuming normalized inputs. Normalized background pixels for a frame are premultiplied by their corresponding alphas before they are blended.

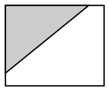
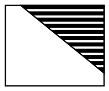
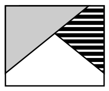
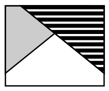
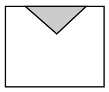
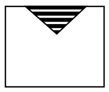
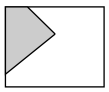
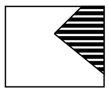
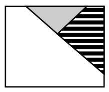
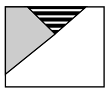
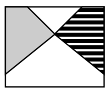
Operation	Diagram	f(A)=	f(B)=
Clear		0	0
A		1	
B			1
A over B		1	1−f(A)
B over A		1−f(B)	1
A in B		f(B)	
B in A			f(A)
A held out by B		1−f(B)	
B held out by A			1−f(A)
A atop B		f(B)	1−f(A)
B atop A		1−f(B)	f(A)
A xor B (union of A out B and B out A)		1−f(B)	1−f(A)

Table derived from Thomas Porter and Tom Duff, “Compositing Digital Images,” published by the Association of Computing Machinery, 1984.

Figure 7-4 Binary Compositing

Table 7-2 shows the choices for the two blend functions A and B, which correspond exactly.

Table 7-2 Choices for Blend Functions A and B

Blend Function A	Blend Function B
f(A) = 0.0	f(B) = 0.0
f(A) = 1.0	f(B) = 1.0
f(A) = f(A)	f(B) = f(B)
f(A) = 1 - f(A)	f(B) = 1 - f(B)

The value 0.0 sets the display to black (cut the foreground or background value); the value 1.0 sets the display to white (pass the foreground or background value):

- If both foreground and background are set to 0.0, the result is black (both foreground and background are cut).
- If foreground is set to 0.0 and background is set to 1.0, foreground is cut (ignored) and background is passed (displayed).
- If foreground is set to 1.0 and background is set to 1-f(A), background obscures and overlaps foreground, resulting in compositing.

Normally, chroma is multiplied (scaled) by the selected alpha. For example, the value on source A can be multiplied by its own alpha value or that from source B. In a normal blend, f(A), the incoming alpha of source A is applied to the value for A. In the inverse of this blend, f(A)=1-f(A), the region that was considered opaque (turned off), that is, outside the volume defined for keying, is applied to source A.

In another way of blending, the alpha from source B can be applied to the component represented by source A. In the inverse of this blend, f(A)=1-f(B), the region that was turned off for source B is applied to source A.

For screen inputs, set VL_BLEND_B_NORMALIZE to FALSE if the keyer mode is set to pass-through (VL_KEYERMODE_NONE), because the alpha has been applied once by the graphics in most cases. In other words, set VL_BLEND_B_NORMALIZE to FALSE if it is following another blender.

For foreground-to-background wipes, background alpha is set to a constant value of 1.0, so that the background shows through the foreground.

Setting and Turning Off Flat Background

For the OCTANE Digital Video option, pixel source B is normally a flat background, supplied by a flat-background generator on this wire (see Figure 7-2). However, you can use device-dependent controls to set the background as desired:

- VL_MGV_BLEND_B_FLAT (default: off)

  When this control is TRUE (on), the background pixel source is used for pixel timing only and live video from pixel source B goes to the blender.

  When this control is off, you can set three controls listed below to values of your choosing for the background. The default value for the background is gray.

  Note: Set this value before the background is turned on if you wish to avoid a flash.
- VL_MGV_BLEND_B_Y (default: 128)

  The legal range of Y is 16 to 235.
- VL_MGV_BLEND_B_U (default: 128 (50% gray))

  The legal range of U is 16 to 240.
- VL_MGV_BLEND_B_V (default: 128)

  The legal range of V is 16 to 240.

The software does not prevent you from using values outside of the range (1-254):

- The values of 1 and 254 are superblack and superwhite.
- The setting Y=235, U=128, V=128 is 100% white.

Adding Shadows

The shadow hardware adds back the Y information from the area that was cut by the keyer. If a shadow exists in the cut area, the effect is to dim the pixels in the area of that shadow in the replaced background.

Use these device-dependent controls to set the shadow values:

- VL_MGV_BLEND_SHADOW_ON

  Set this control to TRUE to activate the shadow hardware.

- VL_MGV_BLEND_SHADOW_GAIN

  The range for this control is 0.0 to 3.0, which shifts the value.

  Use this control and the next one to make the shadow darker or lighter than the "real" shadow you see in the input video.

  Note: Darkening a very light shadow can result in noise.

- VL_MGV_BLEND_SHADOW_OFFSET

  The range for this control is 0 to 255, which is added to the value.

Shadows that are very dark may be hard to key. These registers affect only the background pixels; foreground areas are passed through by the blender. Tune the values for the best effect.

Keying

For each kind of keying—luma keying, chroma keying, and transitions—further VL controls enable you to specify the properties of the blend.

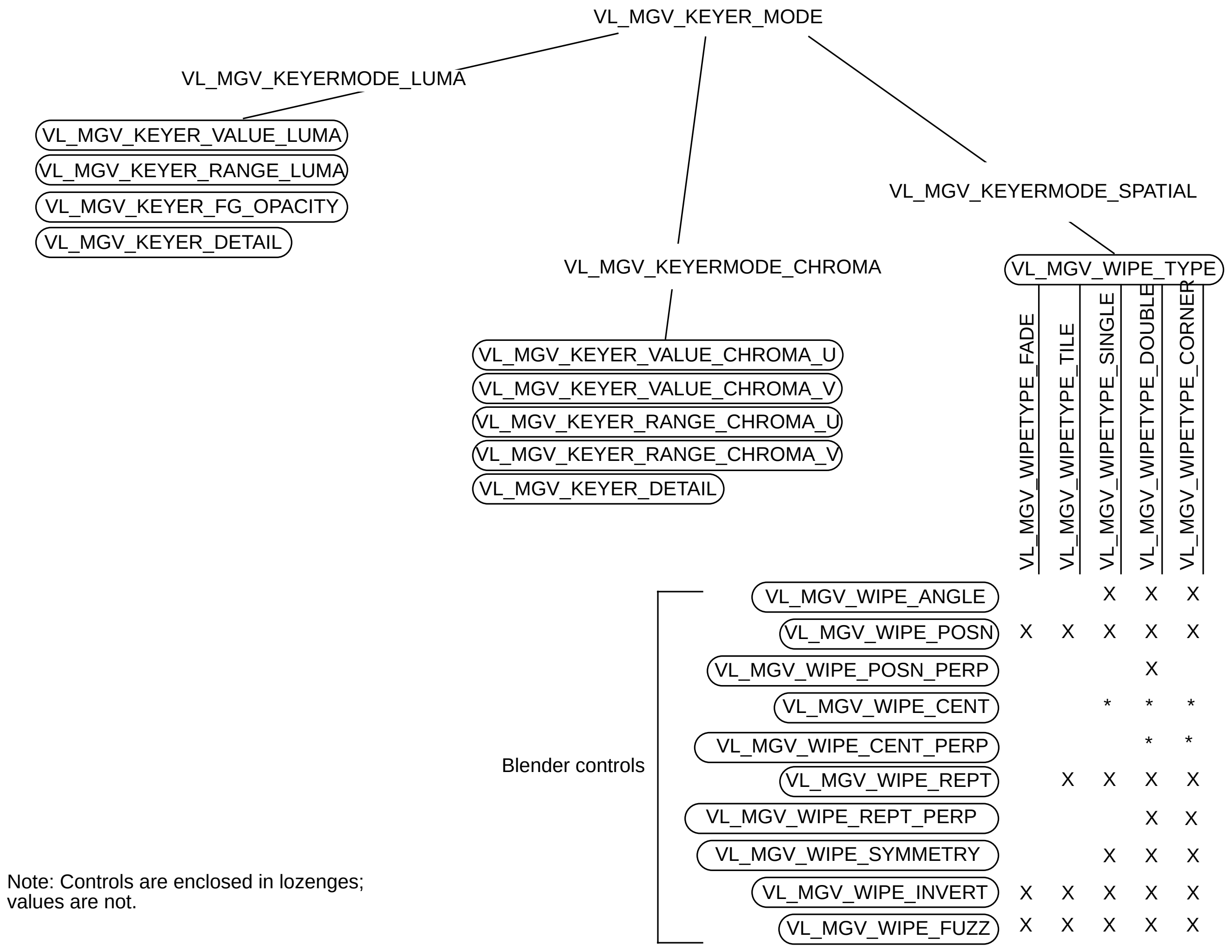
The values for the OCTANE Digital Video "master" keyer control, VL_MGV_KEYER_MODE, determine the type of keying performed:

- luma keying: VL_MGV_KEYERMODE_LUMA
- chroma keying: VL_MGV_KEYERMODE_CHROMA
- transitions, that is, fades, tiles, or wipes: VL_MGV_KEYERMODE_SPATIAL

For example, the following fragment specifies a fade:

```
VLControlType val;
val.intVal = VL_MGV_WIPETYPE_FADE;
vlSetControl(vlSvr, vlPath, blend_node, VL_MGV_WIPE_TYPE,
             &val);
```

Each type of keying is explained separately in this section. Figure 7-5 shows the relationships between the OCTANE Digital Video board keying and wipe controls.



	VL_MGV_WIPETYPE_FADE	VL_MGV_WIPETYPE_TILE	VL_MGV_WIPETYPE_SINGLE	VL_MGV_WIPETYPE_DOUBLE	VL_MGV_WIPETYPE_CORNER
VL_MGV_WIPE_ANGLE			X	X	X
VL_MGV_WIPE_POSN	X	X	X	X	X
VL_MGV_WIPE_POSN_PERP				X	
VL_MGV_WIPE_CENT			*	*	*
VL_MGV_WIPE_CENT_PERP				*	*
VL_MGV_WIPE_REPT		X	X	X	X
VL_MGV_WIPE_REPT_PERP				X	X
VL_MGV_WIPE_SYMMETRY			X	X	X
VL_MGV_WIPE_INVERT	X	X	X	X	X
VL_MGV_WIPE_FUZZ	X	X	X	X	X

* Applies only when VL_MGV_WIPE_SYMMETRY is set.

Figure 7-5 OCTANE Digital Video Keying, Wipe, and Blender Control Relationships

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 7-6 illustrates the results of luma keying.

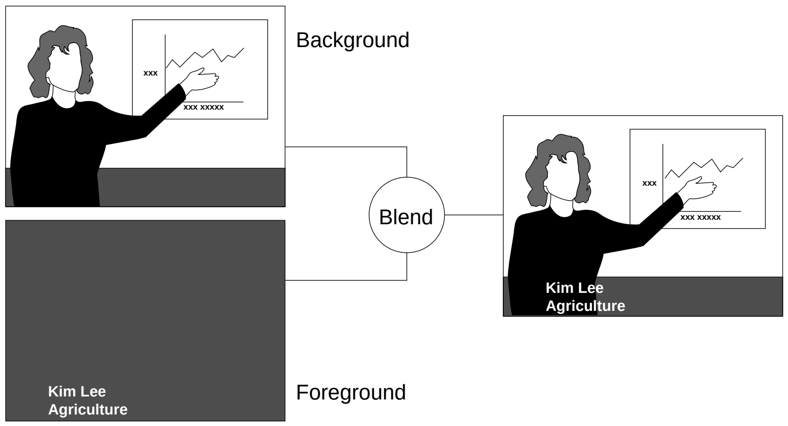


Figure 7-6 Luma Keying Application: Titling

The OCTANE Digital Video luma keying controls are summarized in Table 7-3. For each, the type is intVal and access is GST. The default value is persistent: it is initially obtained from the defaults file, but is never reset. Many controls available through the video control panel *vcp* (for example, the default video input) fall into this category. For this value, changes made by **vlSetControl()** are persistent across paths, even if the node goes into an unused state.

Table 7-3 OCTANE Digital Video Luma Keying Controls

Control	Range	Sets
VL_MGV_KEYER_VALUE_LUMA	(0,255)	Central luma value. This control sets the luma value at which the background shows through the foreground.
VL_MGV_KEYER_RANGE_LUMA	(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_MGV_KEYER_FG_OPACITY	(0,255)	Opacity of the foreground, thus limiting the value of foreground alpha at any point.
VL_MGV_KEYER_DETAIL, VL_MGV_KEYER_SHARPNESS, VL_MGV_KEYER_FUZZ	(-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 7-7 diagrams the relationships between these controls.

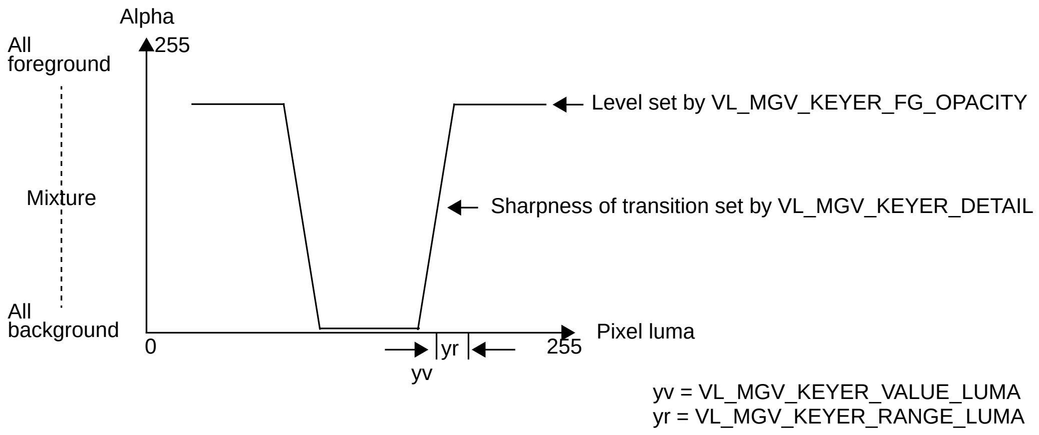

Alpha
All foreground
255
Mixture
All background
0
yr
yv
255
Pixel luma
Level set by VL_MGV_KEYER_FG_OPACITY
Sharpness of transition set by VL_MGV_KEYER_DETAIL
yv = VL_MGV_KEYER_VALUE_LUMA
yr = VL_MGV_KEYER_RANGE_LUMA

Figure 7-7 Relationships Between OCTANE Digital Video Luma Keying Controls

Chroma Keying

Chroma keying overlays one image on another based on the color value. Figure 7-8 illustrates an example of chroma keying.

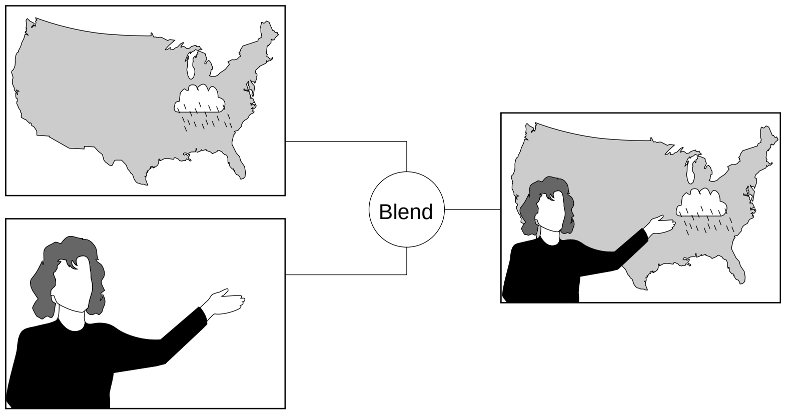


Figure 7-8 Chroma Keying Application: TV Weather Map

Table 7-4 summarizes the controls for OCTANE Digital Video chroma keying and gives their ranges. For each, default is persistent, access is GST, and type is intVal.

Table 7-4 OCTANE Digital Video Chroma Keying Controls

Control	Range	Sets
VL_MGV_KEYER_VALUE_CHROMA_U	(-226,226)	Central U value at which the background shows through the foreground.
VL_MGV_KEYER_RANGE_CHROMA_U	(0,452)	One-sided range of U where the background shows through the foreground.
VL_MGV_KEYER_VALUE_CHROMA_V	(-179,179)	Central V value at which the background shows through the foreground.
VL_MGV_KEYER_RANGE_CHROMA_V	(0,358)	One-sided range of V where the background shows through the foreground.
VL_MGV_KEYER_DETAIL, VL_MGV_KEYER_SHARPNESS, VL_MGV_KEYER_FUZZ	(-8,7)	Sharpness of transition between foreground and background

Note: VL_MGV_KEYER_FG_OPACITY has no effect on OCTANE Digital Video in chroma key mode.

Figure 7-9 diagrams the relationships between these controls.

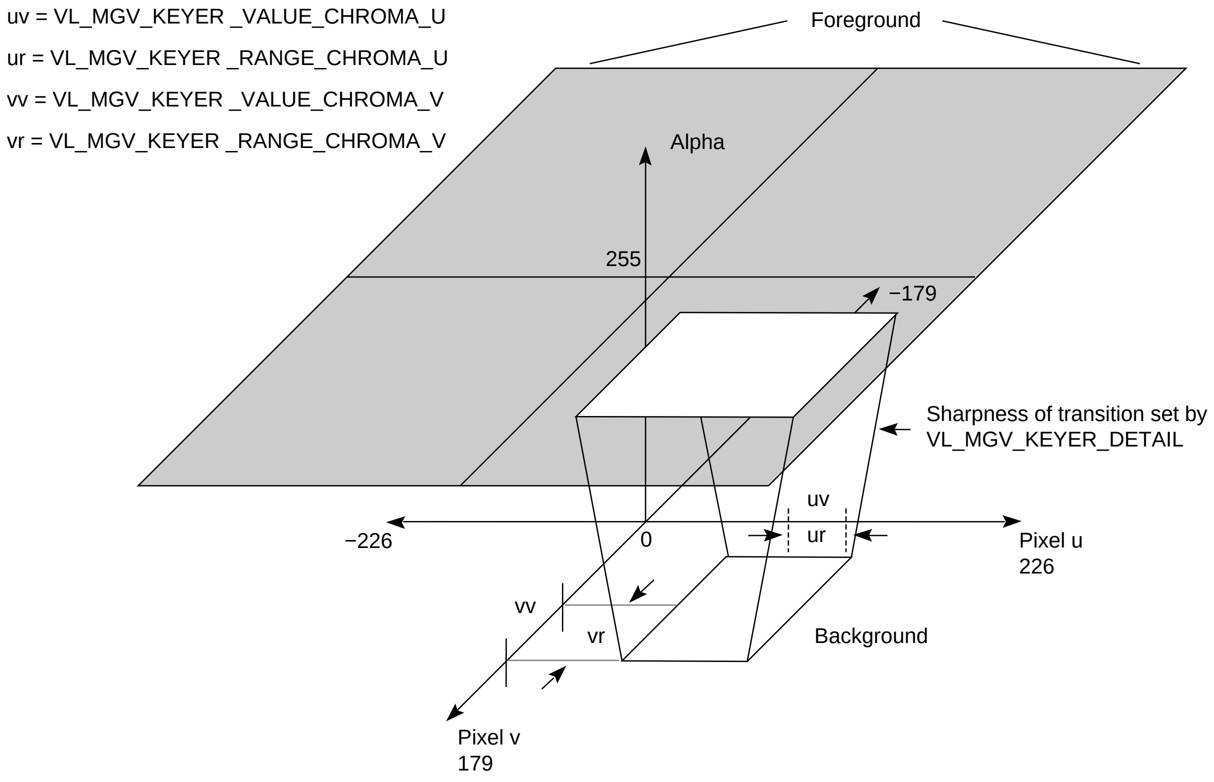

uv = VL_MGV_KEYER _VALUE_CHROMA_U
ur = VL_MGV_KEYER _RANGE_CHROMA_U
vv = VL_MGV_KEYER _VALUE_CHROMA_V
vr = VL_MGV_KEYER _RANGE_CHROMA_V
Foreground
Alpha
255
−179
Sharpness of transition set by VL_MGV_KEYER_DETAIL
uv
ur
−226
0
Pixel u 226
vv
vr
Background
Pixel v 179

Figure 7-9 Relationships Between OCTANE Digital Video Chroma Keying Controls

Fades, Tiles, and Wipes

The values used with the control VL_MGV_WIPE_TYPE determine the type of blending performed:

- from all-foreground to all-background: VL_MGV_WIPETYPE_FADE
- from all-foreground to all-background by randomly tiling screen with rectangles of a specified size: VL_MGV_WIPETYPE_TILE
- wipe to cross the screen as a vertical, diagonal, or horizontal “front,” with a specified angle: VL_MGV_WIPETYPE_SINGLE
- wipe in two orthogonal directions simultaneously (two single wipes at the same time): VL_MGV_WIPETYPE_DOUBLE
- wipe in two orthogonal directions, with the perpendicular position locked to the normal, or in-line position: VL_MGV_WIPETYPE_CORNER

For example, the following fragment specifies that a fade is to be performed:

```
VLControlType val;
val.intVal = VL_MGV_WIPETYPE_FADE;
vlSetControl(vlSvr, vlPath, blend_node, VL_MGV_WIPE_TYPE,
             &val);
```

Fades, tiles, and wipes go from all-foreground (VL_MGV_WIPE_POSN=0) to all-background (VL_MGV_WIPE_POSN=1000), unless VL_MGV_WIPE_INVERT control is set, in which case they go from all-background (VL_MGV_WIPE_POSN = 0) to all-foreground (VL_MGV_WIPE_POSN = 1000).

Table 7-5 summarizes controls common to all wipe types.

Table 7-5 Controls for Fades, Tiles, and Wipes

Control	Values	Sets
VL_MGV_WIPE_POSN type *fractVal*	Numerator (0,1000) Denominator (1000)	Amount of progress of wipe, from none (numerator = 0) to full (numerator = 1000).
VL_MGV_WIPE_REPT type *intVal*	(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 that this control does not apply to fades.
VL_MGV_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 7-6 summarizes the controls specific to wipes or that work differently for wipes. For each, access is GST and the default is persistent, except VL_MGV_WIPE_SYMMETRY and VL_MGV_WIPE_INVERT, for which it is FALSE. Some of these controls work in conjunction with each other.

Table 7-6 OCTANE Digital Video Controls Specific to Wipes

Control	Values	Sets
VL_MGV_WIPE_DIRECTION (VL_MGV_WIPE_ANGLE) type *intVal*	VL_MGV_WIPEANGLE_E VL_MGV_WIPEANGLE_NE VL_MGV_WIPEANGLE_N VL_MGV_WIPEANGLE_NW VL_MGV_WIPEANGLE_W VL_MGV_WIPEANGLE_SW VL_MGV_WIPEANGLE_S VL_MGV_WIPEANGLE_SE	Wipe vector direction, that is, the direction in which the wipe appears to be proceeding as its position increases. Note: VL_MGV_WIPEANGLE_N and VL_MGV_WIPEANGLE_S do not work for the wipe types VL_MGV_WIPETYPE_DOUBLE and VL_MGV_WIPETYPE_CORNER
VL_MGV_WIPE_FUZZ (VL_MGV_WIPE _SHARPNESS) type *intVal*	(-8,7)	Sharpness of wipe transition band. As for VL_MGV_KEYER_DETAIL, -8 is most gradual, +7 is sharpest.

Table 7-6 (continued) OCTANE Digital Video Controls Specific to Wipes

Control	Values	Sets
VL_MGV_WIPE_SYMMETRY type *intVal*	(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_MGV_WIPE_CENT for single, double, and corner wipes; enables VL_MGV_WIPE_CENT_PERP control for double and corner wipes.
VL_MGV_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_MGV_WIPE_POSN.
VL_MGV_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_MGV_WIPE_POSN is 0, and 1000 means center is where VL_MGV_WIPE_POSN is 1000. For this control to work for single, double, and corner wipes, VL_MGV_WIPE_SYMMETRY must be on.
VL_MGV_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_MGV_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.

The Keyer

The role of the keyer is to take a pixel stream and produce an alpha stream. It generates a key for each pixel in each source node:

- If luma keying is set, the keyer assesses the brightness of each pixel.
- If chroma keying is set, the keyer assesses the color of each pixel.
- If spatial, or transition, keying (fade, tile, wipe) is set, the keyer assesses the (x,y) coordinates for each pixel.

The keyer 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. The program *simpleblend.c* illustrates this procedure; it is included in the software and described at the end of this chapter.

The control VL_MGV_BLEND_H_FILT is a horizontal smoothing filter that, if set to TRUE, filters pixel information before the alpha extraction. It smooths the alpha output of the key generator, softening the edges of the image.

Figure 7-10 shows the relationships between value, range, and detail (transition) for a single channel (for example, A).

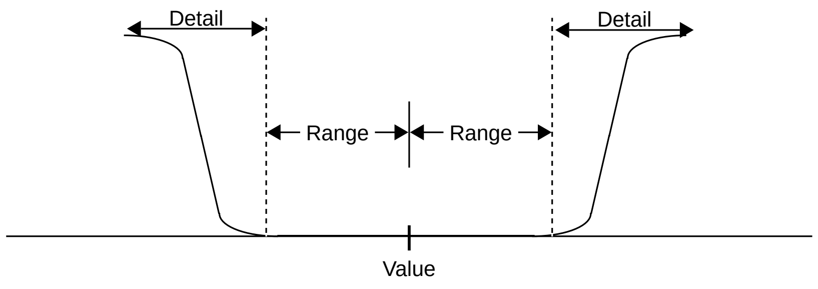


Figure 7-10 Value, Range, and Transition (Keyer Detail) for a Channel

VL Blending Examples

This section explains two example programs from */usr/share/src/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. However, you should always check returns.

Blending Video and Graphics

simpleblend.c 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 `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 or VL_WIPE_DIRECTION, 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()**

Chapter 8

Using Color-Space Conversion

A *color space* is a color component encoding format, for example, RGB and YUV. Because various types of video equipment use different formats, conversion is sometimes required. The Video Library for the OCTANE Digital Video option has a particular node, VL_CSC, to handle color-space conversion.

The OCTANE Digital Video color-space converter (CSC) node controls the color-space converter hardware. The CSC hardware can perform many types of image-processing operations on a video path. This node allows VL control over the conversion process.

The OCTANE Digital Video CSC feature can be used in two ways: for standard conversions between YUV and RGB; or for image processing, for example, posterization[1], solarization[2], or color correction. VL for the OCTANE Digital Video option includes controls for these types of conversions.

This chapter explains

- "Features of the Color-Space Conversion Node"
- "Performing Standard Color-Space Conversions"
- "Using the Color-Space Converter for Image Processing"
- "Examples"

Note: For background information on color-space conversion, see Appendix C, "OCTANE Digital Video Color-Space Conversions," later in this guide.

[1] Quantizing the distortion of color difference signals to obtain special effects, here limiting the number of possible colors in the picture, thus giving the effect of a poster painted with a limited number of colors.

[2] A form of contrast enhancement: when transfer functions become so distorted that the slope reverses, the result is "luminance reversal," where black and white are effectively interchanged.

Features of the Color-Space Conversion Node

Table 8-1 lists the video formats supported for color-space conversion.

Table 8-1 Supported Video Output Formats

Format	**Description**
YUV 4:2:2 (CCIR 601))	Single-link YUV
YUVA 4:2:2:4 (CCIR 601)	Dual-link YUV with alpha
YUVA 4:4:4:4 (CCIR 601)	Dual-link YUV with alpha and all chroma samples
RGBA RP-175	Dual-link RGBA, 8-bit [16-235] or 10-bit [64-940]
RGBA (0-255)	Dual-link RGBA [0-255] (8 bits, full-scale)
RGBA (0-1023)	RGBA full-scale and 10 bits to or from memory only

Figure 8-1 is a software model of a color-space converter.

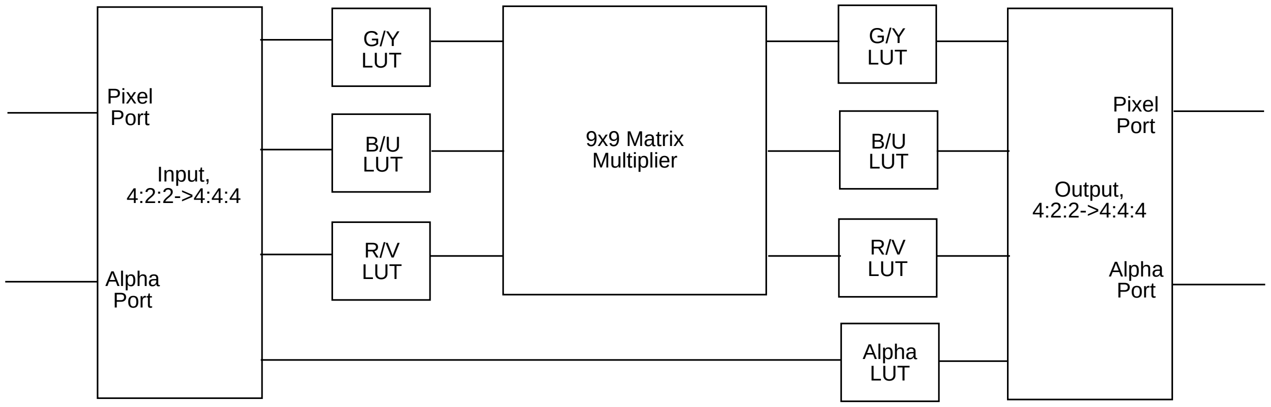


Figure 8-1 Color-Space Conversion Software Model

The high-level processing blocks comprising a color-space converter in Figure 8-1 include

- an input block taking in pixel/alpha, or dual-linked, input and producing component streams; 4:2:2 -> 4:4:4 upsampling is performed for 4:2:2 / 4:2:2:4 inputs
- input lookup tables for each component stream
- a 3 x 3 matrix multiplier
- an output lookup table for each coefficient stream
- an output block producing pixel/alpha, or dual-linked, streams; 4:4:4 to 4:2:2 downsampling is performed for 4:2:2 / 4:2:2:4 output packings

These processing blocks allow the standard RGB-to-YUV and YUV-to-RGB color-space conversions to be implemented, as well as other forms of image processing such as color correction or posterization.

By default, the Video Library’s color-space converter node loads standard conversion values for the lookup tables and matrix multiplier, depending on the source and target color spaces. The application can use the node’s controls to override these values and achieve nonstandard conversions.

Figure 8-2 shows a more detailed model of OCTANE Digital Video’s color-space converter, including delay blocks synchronizing the component paths. This model also shows elements unique to OCTANE Digital Video, such as the constant-hue processing block.

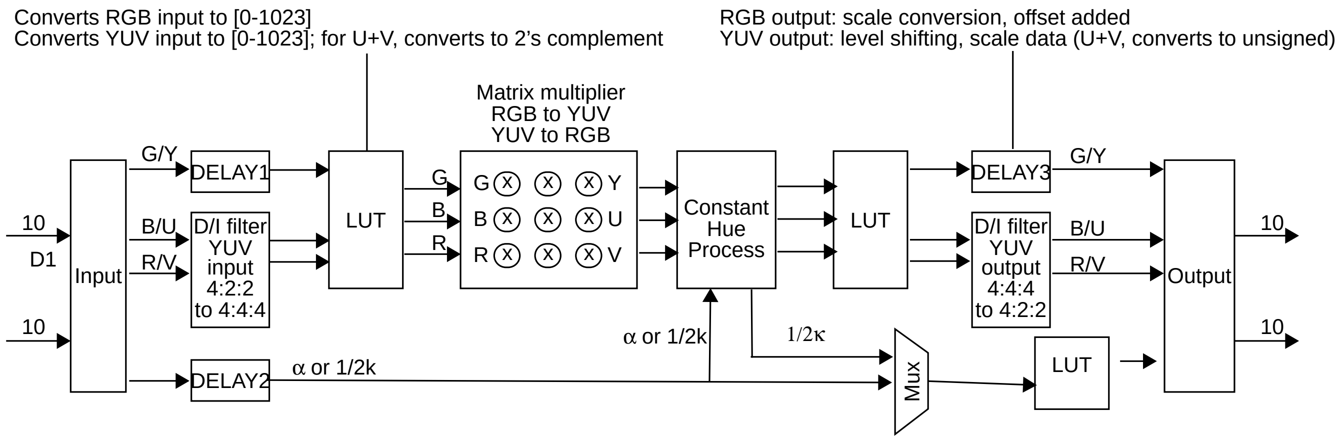

Converts RGB input to [0-1023]
Converts YUV input to [0-1023]; for U+V, converts to 2's complement
RGB output: scale conversion, offset added
YUV output: level shifting, scale data (U+V, converts to unsigned)
Matrix multiplier
RGB to YUV
YUV to RGB
10
D1
10
Input
G/Y
B/U
R/V
DELAY1
D/I filter YUV input 4:2:2 to 4:4:4
DELAY2
α or 1/2k
LUT
G
B
R
G X X X Y
B X X X U
R X X X V
Constant Hue Process
α or 1/2k
1/2κ
Mux
LUT
DELAY3
D/I filter YUV output 4:4:4 to 4:2:2
G/Y
B/U
R/V
LUT
Output
10
10

Figure 8-2 Color-Space Conversion Input to Output Paths: YUV to G'B'R'

Performing Standard Color-Space Conversions

The VL_CSC node converts incoming video from one packing to another, where packing describes color space, component encoding, and bits per component, as shown in Table 8-1. The node operates on full-range or limited-range data.

The kind of the color-space conversion node is VL_CSC; its type is VL_INTERNAL; its number is VL_MGV_NODE_NUMBER_CSC.

The normal usage of the color-space node is to convert from RGB to YUV/YCrCb or from YUV/YCrCb to RGB color spaces. These conversions are specified by setting the input and output packing of the color-space converter using the VL_MGV_CSC_INPUT_PACKING and VL_MGV_CSC_OUTPUT_PACKING controls, respectively, and by specifying the input and output ranges using the VL_MGV_CSC_INPUT_RANGE and VL_MGV_CSC_OUTPUT_RANGE. All other controls on the color-space converter node, such as the input, output, and alpha LUTs and matrix coefficients, default to values supporting the specified conversion.

This section explains

- color-space conversion packings
- color-space conversion ranges
- constant hue
- ports
- full-range and limited-range video
- specifying standard color-space conversion node controls

Color-Space Conversion Packings

The controls that set color-space packing are VL_MGV_CSC_IN_PACKING and VL_MGV_CSC_OUT_PACKING. Table 8-2 summarizes the packing types to use with the color-space converter and their meanings in terms of color space, component ordering, and component size.

Table 8-2 Supported Packing Types for Color-Space Conversion

Packing	Color-Space	Components	Component Size
VL_PACKING_YVYU_422_8	YUV	4:2:2	Pixel size 24: 8 bits for each
VL_PACKING_YVYU_422_10	YUV	4:2:2	10 bits for each
VL_PACKING_YUVA_4444_8	YUV	4:4:4:4	Pixel size 24: 8 for each
VL_PACKING_YUVA_4444_10	YUV	4:4:4:4	10 bits for each
VL_PACKING_AUYV_4444_8	YUV	4:4:4:4	8 bits for each
VL_PACKING_AUYV_4444_10	YUV	4:4:4:4	10 bits for each
VL_PACKING_A_2_UYV_10	YUV	4:4:4:4	Pixel size 32: 10 bits each for UYV and 2 for A
VL_PACKING_AYU_AYV_10	YUV	4:2:2:4	10 bits each for Y, U, V, and A
VL_PACKING_RGBA_8	RGB	4:4:4:4	8 bits for each
VL_PACKING_RGBA_10	RGB	4:4:4:4	10 bits for each
VL_PACKING_ABGR_8	RGB	4:4:4:4	8 bits for each

Table 8-2 (continued) Supported Packing Types for Color-Space Conversion

Packing	Color-Space	Components	Component Size
VL_PACKING_BGR_8_P	RGB	4:4:4	Pixel size 24: 8 bits for each in OpenGL format
VL_PACKING_ABGR_10	RGB	4:4:4:4	Pixel size 32: 10 bits each for BGR, 2 for A
VL_PACKING_A_2_BGR_10	RGB	4:4:4:4	Pixel size 32: 10 each for BGR, 2 for A

Note: Although the OCTANE Digital Video option supports the packing formats VL_PACKING_RGB_332_P, VL_PACKING_Y_8_P, and VL_PACKING_RGB_8, the color-space conversion feature does not.

Besides color space, component encoding, and bits per component, a packing also denotes component order, as stored in memory, which is ignored by the color-space converter node.

Eight-bit video supplied through the video inputs or memory nodes is automatically left-shifted to 10 bits, with bits 0 and 1 set to zero. Consequently, at the color-space converter's input and output, all data is 10-bit, and the 8- and 10-bit packings listed in Table 8-2 are functionally equivalent.

Setting VL_MGV_CSC_IN_PACKING or VL_MGV_CSC_OUT_PACKING loads the input, output, and alpha LUTs and matrix multiplier coefficients with values that reflect the conversion specified by this control and by the range controls.

Color-Space Conversion Ranges

The Video Library formats denote the range of pixel components. Note that formats also convey color-space information; this information is not used by the color-space converter node, although it is highly recommended that the packing and format color spaces specified on the input or output agree.

The controls that set the input and output ranges of the color-space converter are VL_MGV_CSC_IN_RANGE and VL_MGV_CSC_OUT_RANGE. Table 8-3 summarizes the ranges supported by the color-space converter node.

Table 8-3 Supported Ranges

Format	Range
VL_FORMAT_DIGITAL_COMPONENT_SERIAL	CCIR-Range YCrCb 8-bit: 16-235 (Y), 16-240 (Cr, Cb) 10-bit 64-940 (Y), 64-960 (Cr, Cb)
VL_FORMAT_DIGITAL_COMPONENT_SERIAL_RGB	RP-175-Range RGB 8-bit: 16-235 (R, G, B) 10-bit: 64-940 (R, G, B)
VL_FORMAT_SMPTE_YUV	Full-Range YUV 8-bit: 0-255 (Y, U, V) 10-bit 0-1023 (Y, U, V)
VL_FORMAT_RGB	Full-Range RGB 8-bit 0-255 (R, G, B) 10-bit 0-1023 (R, G, B)

When a range control is set, input, output, and alpha LUTs and matrix multiplier coefficients are loaded with values that reflect the conversion specified by the input/output packing and range.

Note: If output packing is VL_PACKING_YVYU_422_8 or VL_PACKING_YVYU_422_10, output range is VL_FORMAT_SMPTE_YUV (full-range YUV), and the converted output is to be saved in memory, you must use VGI1 1, that is, memory drain node 1 (1 or VL_MGV_NODE_NUMBER_MEM_VGI1_2).

Constant Hue

In addition to the standard color-space conversion model, the OCTANE Digital Video color-space feature provides a *constant-hue algorithm*. This algorithm allows illegal YUV values to survive a YUV-to-RGB-to-YUV conversion. In normal conversion, YUV values that cannot be represented in the RGB color space are clamped or otherwise forced into the legal RGB range. Because the YUV (YCrCb) color space is a superset of the RGB color space, illegal RGB values can be generated when YUV is converted to RGB. If the constant-hue block is disabled, then the illegal RGB values are clipped by the output LUT. The lost (clipped) information can result in significantly degraded quality when the image is subsequently transformed back to YUV for video output.

The constant-hue algorithm saves the normally lost information in a correction factor that can be stored in the alpha channel. To restore the original YUV image, this correction factor must be saved with the pixel data.

If the constant-hue algorithm is enabled, the illegal RGB values are converted into legal R′G′B′ values. A constant-hue factor, used to restore R′G′B′ to the original YUV values, can optionally be stored in the alpha channel. If the constant-hue factor is not saved, then the R′G′B′ image appears as if it were range-compressed. The VL_MGV_CSC_ALPHA_CORRECTION control determines whether the alpha channel is replaced by the constant-hue factors, or if the alpha from the color-space converter's input is retained.

Note that because the correction factor computed by the algorithm is directly related to the pixel value, the correction factor is invalidated if the pixel value is recalculated (for example, during compositing).

The controls for constant hue are

- VL_MGV_CSC_CONST_HUE: boolean control to enable (TRUE) or disable (FALSE) the constant-hue algorithm
- VL_MGV_CSC_ALPHA_CORRECTION: boolean control to select the contents of the alpha channel

  If this value is set to TRUE, the constant-hue factor is saved in the alpha channel. If it is set to FALSE, the alpha value from the input is retained.

Note: VL_MGV_CSC_ALPHA_CORRECTION has no effect if VL_MGV_CSC_CONST_HUE is disabled. When both VL_MGV_CSC_CONST_HUE and VL_MGV_CSC_ALPHA_CORRECTION are enabled, it is not advisable to load the alpha LUT.

By default, the constant-hue processing block is enabled, but the constant-hue factor is not stored in the alpha channel (the input alpha is retained).

If the constant-hue factor is not stored in the alpha channel, you might need to range-limit or expand the input alpha value. For example, when full-range RGBA is converted to YCrCbA, the range is limited from [0-255] to CCIRs [16-235]. The range is altered using the output alpha LUT. The default contents of this LUT are determined by the input and output ranges.

VL_CSC Ports

Table 8-4 lists the ports for the color-space conversion node.

Table 8-4 Color-Space Conversion Node Ports

Port	Use
VL_IMPACT_PORT_PIXEL_DRN_A	Pixel input 4:2:2:4
VL_IMPACT_PORT_ALPHA_DRN_A	Alpha input 4:2:2:4
VL_IMPACT_PORT_DUALLINK_DRN_A	Dual link input 4:4:4:4
VL_IMPACT_PORT_PIXEL_SRC_A	Pixel output 4:2:2:4
VL_IMPACT_PORT_ALPHA_SRC_A	Alpha output 4:2:2:4
VL_IMPACT_PORT_DUALLINK_SRC_A	Dual link input 4:4:4:4

The color-space converter node has both single- and dual-linked input and output ports. Only one of these should be used at a time.

- Use the dual-linked ports if the video stream is coming from a dual-linked video or memory node. In almost all cases, the dual-linked stream contains RGBA, YUV 4:4:4:4, or YUV 4:2:2:4 samples. For RGBA, the components are split across the two links according to the RP-175 specification: link 1 (pixels) contains RGB 4:2:2, while link 2 (alpha) contains AGB 4:2:2.

  Note: Use dual-linked connections for RGB or YUV 4:4:4:4 input or output. Because the chrominance samples span both input/output ports, any mistiming between two single-linked connections results in corrupted video.

- Use the single-linked port if the pixel and alpha are coming from two single-linked ports. The single-linked nodes allow pixel values to be specified independently of alpha values for YUV input or output. For example, to convert a YCrCb 4:2:2 stream to RGB, connect only the pixel input port; the alpha input port contains black.

  Note: Exercise caution when both the alpha and pixel ports are connected to different sources; these sources must be perfectly timed or the output pixels can have mismatched alpha.

The software model in Figure 8-1 shows the pixel and alpha ports on the input and output blocks.

Full-Range and Limited-Range Video

The color-space conversion node operates on full-range or limited-range data. Range is specified by a VL format.

Standard D1 VIdeo Range

The standard D1 video range is as follows:

- RGB (three primaries): Y (the luminance signal) and A (the auxiliary channel) each contain the ranges 16-235 for 8-bit data and 64-940 for 10-bit data.
- The U (Cb) and V (Cr) signals contain the ranges 16-240 for 8-bit data and 64-960 for 10-bit data.

Standard RGBA Full Range

The standard full range is 0-255 for 8-bit data and 0-1023 for 10-bit data.

For full-range component values, the resulting video stream might contain false EAV/SAV codes. The VGI1 memory nodes (memory nodes 0, 1, and 2) and the color-space converter can ignore these codes, but the rest of the device cannot. Consequently, full-range YUV or RGB data should be exchanged between the VGI1 and color-space converter only. RP-175-range RGB or CCIR-range YUV video can be exchanged among all OCTANE Digital Video nodes.

Specifying Standard Color-Space Conversion Node Controls

Table 8-5 summarizes controls for standard color-space conversion.

Table 8-5 Controls for Standard Color-Space Conversion

Control	Use	Default	Type
VL_MGV_CSC_IN_PACKING, VL_MGV_CSC_OUT_PACKING	Sets the packing for the color-space converter node's input or output, respectively	VL_PACKING_YVYU _422_10	intVal
VL_MGV_CSC_IN_RANGE, VL_MGV_CSC_OUT_RANGE	Sets the input range (RP-175, CCIR, or full) associated with the input or output video, respectively	VL_FORMAT_DIGITAL _COMPONENT_SERIAL	intVal
VL_MGV_CSC_CONST_HUE	Enables or disables constant-hue algorithm	TRUE	boolVal
VL_MGV_CSC_ALPHA _CORRECTION	When VL_MGV_CSC_CONST_HUE is enabled, this control saves the constant hue correction factor (TRUE) or retains alpha input data (FALSE)	FALSE	boolVal

For all these controls the access is GS:

- G: the value can be retrieved through **vlGetControl()**
- S: the value can be set through **vlSetControl()** while the path is not transferring

When any of these controls are set, the LUTs and matrix coefficients are loaded with values reflecting the input packing/output packing, range conversion, and constant hue, if it is set. Applications can optionally load custom LUTs or coefficients after setting the input and output packing and range.

Using the Color-Space Converter for Image Processing

This section describes the color-space converter in more detail for application developers who wish to use it as an image processing device capable of posterization, solarization, or color correction, or for nonstandard conversions. These applications specify custom values for the lookup tables and matrix coefficients, and require more effort than standard conversions.

Note: See also Appendix C, "OCTANE Digital Video Color-Space Conversions," and "VL_CSC" in Appendix B for information on the CSC controls.

In addition to standard conversions, the color-space converter can be loaded with user-defined input lookup tables, matrix multiplier coefficients and output lookup tables. Applications can manipulate the tables and coefficients to perform color correction, colorization, or other image-processing functions. See Appendix C for a description of the color-space converter model and the relationships between the various internal processing blocks.

Using Custom LUTs and Matrix Multiplier Coefficients

User-defined software-loadable lookup tables support operations such as gamma correction. The input and alpha lookup tables each contain 1024 entries of 12-bit values. For each of these tables, up to four lookup tables can be loaded simultaneously. An application can use the LUT selection controls to select which of the tables is active.

The output lookup tables each contain 4096 entries of 10-bit values. Only one table can be loaded for each of the VR, GY, and UB output LUTs. The matrix multiplier operates with 23-bit internal precision.

Color-space conversion is supported in the active video region only; the vertical and horizontal blanking areas are replaced with black (in the appropriate target color space). Consequently, ancillary data, such as telextext, embedded audio, and embedded timecode, is lost.

Note: Use the standard controls VL_MGV_CSC_IN_PACKING, VL_MGV_CSC_OUT_PACKING, VL_MGV_CSC_IN_RANGE, and VL_MGV_CSC_OUT_RANGE for image processing, even if the input and output packings/ranges are identical. Even if the application subsequently changes all of the lookup tables and coefficients, it should still set the input and output packing to reflect the color space and component encoding. The packing control enables or disables parts of the color-space converter, such as the 4:2:2 to or from 4:4:4 filters.

Standard conversion controls take precedence over those for image processing. For example, if the input or output packing control is set, lookup tables and coefficient values are recalculated to effect the conversion implied by the input/output packing and range.

Using Image-Processing Controls

Table 8-6 summarizes image-processing controls. Access for all these controls is GST:

- G: The value can be retrieved through **vlGetControl()**.
- S: The value can be set through **vlSetControl()** while the path is not transferring.
- T: The value can be set through **vlSetControl()** while the path is transferring.

Table 8-6 Image-Processing Controls

Control	Use	Default	Type
VL_MGV_CSC_COEF	Specifies the matrix multiplier coefficients	Multiplier operates in pass-through mode	extendedVal; data type MGV_CSC_COEF
VL_MGV_CSC_LUT_IN_PAGE VL_MGV_CSC_LUT_ALPHA_PAGE	Selects the active LUT	0	intVal
VL_MGV_CSC_LUT_IN_YG VL_MGV_CSC_LUT_IN_UB VL_MGV_CSC_LUT_IN_VR VL_MGV_CSC_LUT_ALPHA	Specifies the contents of the input or alpha lookup tables	Pass-through (1:1 mapping)	extendedVal; data type MGV_CSC_LUT_INPUT_AND_ALPHA
VL_MGV_CSC_LUT_OUT_YG VL_MGV_CSC_LUT_OUT_UB VL_MGV_CSC_LUT_OUT_VR	Specifies the contents of the output lookup tables	Pass-through (1:1 mapping)	extendedVal data type MGV_CSC_LUT_OUTPUT

Using Coefficients

The control VL_MGV_CSC_COEFF specifies the matrix multiplier coefficients. It has a data pointer pointing to an array of nine integers. The coefficients are stored in the following order:

- data[0] = Y/G 1 data[1] = Y/G 2 data[2] = Y/G 3
- data[3] = U/B 1 data[4] = U/B 2 data[5] = U/B 3
- data[6] = V/R 1 data[7] = V/R 2 data[8] = V/R 3

Each coefficient is a 32-bit fractional two's complement value. The magnitude of each coefficient is from -4 to 3.999. Table 8-7 shows values.

Table 8-7 Coefficient Formats

Bit	Value
31	-2^2(signed bit)
30	2^1
29	2^0
28	2^{-1}
27	2^{-2}
26	2^{-3}
25	2^{-4}
24	2^{-5}
23	2^{-6}
...	...
4	2^{-25}
3	2^{-26}
2	2^{-27}
1	2^{-28}
0	2^{-29}

For OCTANE Digital Video color-space conversion, the valid range for data[0], data[4], and data[8] is from -4 to 3.999; for the other six coefficients, the valid range is from -2 to 1.999. The 31th and 30th bits of the other six coefficients must be either all 0's or all 1's for the range from -2 to 1.999; otherwise they are clamped to the valid range.

Selecting the Active LUT

The OCTANE DIgital Video color-space converter node can store up to four input LUTs (each with YG, UB, and VR), and four alpha LUTs. Use the control VL_MGV_CSC_LUT_IN_PAGE or VL_MGV_CSC_LUT_ALPHA_PAGE in the application to select which of the four LUTs is active.

Using Input and Alpha LUTs

The controls for specifying the contents of the input or alpha lookup tables are VL_MGV_CSC_LUT_IN_YG, VL_MGV_CSC_LUT_IN_UB, VL_MGV_CSC_LUT_IN_VR, and VL_MGV_CSC_LUT_ALPHA.

The data pointer of the extended value points to a VL_MGVInAlphaLutValue structure, as defined in *dev_mgv.h*. This structure contains the page number for the LUT being specified and a lookup table of 1024 integer entries (see VL_MGV_CSC_LUT_IN_PAGE and VL_MGV_CSC_LUT_ALPHA_PAGE) for selecting the LUT active during color-space conversion). The range for each entry in the lookup table is 0-1023 (10 bits).

Using Output LUTs

The controls for specifying output LUTs are VL_MGV_CSC_LUT_OUT_YG, VL_MGV_CSC_LUT_OUT_UB, and VL_MGV_CSC_LUT_OUT_VR.

The data pointer of the extended value points to a VL_MGVOutLutValue structure, as defined in *dev_mgv.h*. This structure contains a lookup table of 4096 integer entries. The OCTANE Digital Video color-space converter can store only one output LUT for each of the YG/UB/VR paths. The range for each entry in the lookup table is 0-1023 (10 bits).

Examples

This section includes two color-space conversion examples.

Example 8-1 Setting CSC Controls for Standard Color-Space Conversion

```
/*
 * The following example is to demostrate how to set CSC controls to perform
 * standard color space conversion. Video input in CCIR 10-bit YUV 422 packing
 * is converted to full-range 10-bit ABGR packing which will be saved in
 * memory. Controls for constant hue and alpha correction are enabled.
 *
 * Video input is source, memory (VGI1) is drain, and CSC is a internal node
 * in a video to memory path. Because this is a CSC example, codes other
 * than CSC are replaced by comments in this examples.
 */

#include <errno.h>
#include <stdio.h>
#include <dmedia/vl.h>
#include <dmedia/vl_mgv.h>

void main(void)
{
        VLTransferDescriptor xferDesc;
        VLControlValue val;
        VLServer svr;
        VLPath path;
        VLNode src, drn;
        VLNode csc_node;

        /* We connect to daemon and set up nodes for source and drain */

        /* Create CSC internal node */
        csc_node = vlGetNode(svr, VL_INTERNAL, VL_CSC, VL_MGV_NODE_NUMBER_CSC);
```

```
/*
 * We create a video to memory path with the source and drain
 * created above.
 */

/* Add the CSC internal node to the path */
if (vlAddNode(svr, path, csc_node)) {
    vlPerror(“Add CSC Device Node”);
    vlDestroyPath(svr, path);
}

/* Set CSC input range to CCIR */
val.intVal = VL_FORMAT_DIGITAL_COMPONENT_SERIAL;
if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_IN_RANGE, &val) < 0)
    vlPerror(“SetControl of CSC input range failed”);

/* Set CSC input packing to 10-bit YUV 422 */
val.intVal = VL_PACKING_YVYU_422_10;
if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_IN_PACKING, &val) < 0)
    vlPerror(“SetControl of CSC input packing failed”);

/* Set CSC output range to full range */
val.intVal = VL_FORMAT_RGB;
if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_OUT_RANGE, &val) < 0)
    vlPerror(“SetControl of CSC output range failed”);

/* Set CSC output packing to 10-bit ABGR */
val.intVal = VL_PACKING_ABGR_10;
if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_OUT_PACKING, &val) < 0)
    vlPerror(“SetControl of CSC output packing failed”);

/* Enable CSC constant hue algorithm, default is TRUE */
val.boolVal = TRUE;
if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_CONST_HUE, &val) < 0)
    vlPerror(“SetControl of CSC constant hue failed\n”);
```

```
        /* Enable alpha correction in the alpha channel, default is FALSE */
        val.boolVal = TRUE;
        if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_ALPHA_CORRECTION, &val)
            < 0)
            vlPerror("SetControl of CSC alpha correction failed\n");

        /* We set VGI1's controls below */

        /* VGI1's VL_FORMAT should be same as CSC's output range */
        val.intVal = VL_FORMAT_RGB;
        if (vlSetControl(svr, path, drn, VL_FORMAT, &val) < 0)
            vlPerror("SetControl of memory format failed");

        /* VGI1's VL_PACKING should be same as CSC's output packing */
        val.intVal = VL_PACKING_ABGR_10;
        if (vlSetControl(svr, path, drn, VL_PACKING, &val) < 0)
            vlPerror("SetControl of memory packing failed");

        /* Begin the data transfer */
        if (vlBeginTransfer(svr, path, 1, &xferDesc)) {
            vlPerror("vlBeginTransfer failed");
        }

        /* Loop until all requested frames are grabbed */
        vlMainLoop();

}
```

Example 8-2 uses the color-space converter node to perform image processing.

Example 8-2 Loading Matrix Coefficients and LUTs

```
/*
 * The following example is to demostrate how to load matrix coefficients,
 * input luts, output luts and alpha lut.  Codes other than loading are
 * replaced by comments in this examples.
 */

#include <errno.h>
#include <stdio.h>
#include <dmedia/vl.h>
#include <dmedia/vl_mgv.h>

void main(void)
{
        VLTransferDescriptor xferDesc;
        VLControlValue val;
        VLServer svr;
        VLPath path;
        VLNode src, drn;
        int i;

        int coef_table[9] =
            {0, 10, 20, 30, 40, 50, 60, 70, 80};

        /* We connect to daemon and set up nodes for source and drain */

        /* Create CSC internal node */
        csc_node = vlGetNode(svr, VL_INTERNAL, VL_CSC, VL_MGV_NODE_NUMBER_CSC);

        /*
         * We create a video to memory path with the source and drain
         * created above.
         */

        /* Add the CSC internal node to the path */
        if (vlAddNode(svr, path, csc_node)) {
            vlPerror(“Add CSC Device Node”);
            vlDestroyPath(svr, path);
        }
```

```
/* Load 9 coefficients */
extVal.dataPointer = coef_table;
extVal.dataSize = sizeof (coef_table);
extVal.dataType = MGV_CSC_COEF;

val.extVal = extVal;

if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_COEF, &val) < 0) {
    vlPerror("SetControl of CSC coefficients failed");
}

/* Generate one page of lut */
for (i = 0; i < 1024; i++)
    inlutVal.lut[i] = 1000;

/* Assign one page of lut we generated to be in 1st page of
   input luts and alpha lut. */
inlutVal.pageNumber = 0;

extVal.dataType = MGV_CSC_LUT_INPUT_AND_ALPHA;
extVal.dataSize = sizeof(VL_MGVInAlphaLutValue);
extVal.dataPointer = &inlutVal;

val.extVal = extVal;

/* Load the page to input Y/G lut */
if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_LUT_IN_YG, &val) < 0) {
    vlPerror("SetControl of CSC YG input lut failed\n");
}

/* Load the page to input U/B lut */
if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_LUT_IN_UB, &val) < 0) {
    vlPerror("SetControl of CSC UB input lut failed\n");
}

/* Load the page to input V/R lut */
if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_LUT_IN_VR, &val) < 0) {
    vlPerror("SetControl of CSC VR input lut failed\n");
}

/* Load the page to alpha lut */
if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_LUT_ALPHA, &val) < 0) {
    vlPerror("SetControl of CSC ALPHA lut failed\n");
}
```

```
        /* Select the 1st page of input and alpha luts to be active during
           normal mode, so they will be used for normal operation. */
        val.intVal = 0;
        if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_LUT_IN_PAGE, &val) < 0)
            vlPerror("SetControl of CSC input lut page failed\n");

        val.intVal = 0;
        if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_LUT_ALPHA_PAGE, &val)
            < 0)
            vlPerror("SetControl of CSC alphalut page failed\n");

        /* Generate 4 pages of lut */
        for (i = 0; i < 4096; i++)
            outlutVal.lut[i] = 1000;

        extVal.dataType = MGV_CSC_LUT_OUTPUT;
        extVal.dataSize = sizeof(VL_MGVOutLutValue);
        extVal.dataPointer = &outlutVal;

        val.extVal = extVal;

        /* Load all 4 pages to output Y/G lut */
        if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_LUT_OUT_YG, &val)
            < 0) {
            vlPerror("SetControl of CSC YG output lut failed\n");
        }

        /* Load all 4 pages to output U/B lut */
        if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_LUT_OUT_UB, &val)
            < 0) {
            vlPerror("SetControl of CSC YG output lut failed\n");
        }

        /* Load all 4 pages to output V/R lut */
        if (vlSetControl(svr, path, csc_node, VL_MGV_CSC_LUT_OUT_VR, &val)
            < 0) {
            vlPerror("SetControl of CSC YG output lut failed\n");
        }

        /* Here we set VGI1 controls, begin transfer and Loop until all
           requested frames are grabbed */

}
```

Chapter 9

Using Video Texture Mapping

The Video Library for the OCTANE Digital Video option contains a texture node for capturing video streams into the texture memory on OCTANE graphics. Once captured, a video field can be used as a texture just as if it were an image loaded into texture memory through function calls to the OpenGL graphics library. [1]

The texture map interface hardware has two inputs, link A and link B.

- For single-link transfers, you can specify which input to send to TRAM.
- For dual-link transfers, the RGB data is extracted from link A, and the alpha data is extracted from link B.

The texture-map interface hardware also has a scaler chip for extracting a rectangular subregion of the input video field and shrinking it either horizontally or vertically before it is transferred to TRAM.

Finally, the texture-map interface hardware also contains a real-time mipmap generator, which can create and transfer the levels of detail necessary to display the video field in mipmap mode on the graphics subsystem (see the *OpenGL Programming Guide* for a discussion of mipmapping).

Double-buffered video texture applications require 4 MB of TRAM, which is available with OCTANE graphics.

Figure 9-1 diagrams the hardware configuration that supports the texture node.

[1] For a more detailed explanation of the OpenGL routines mentioned in this chapter, see the *OpenGL Programming Guide*.

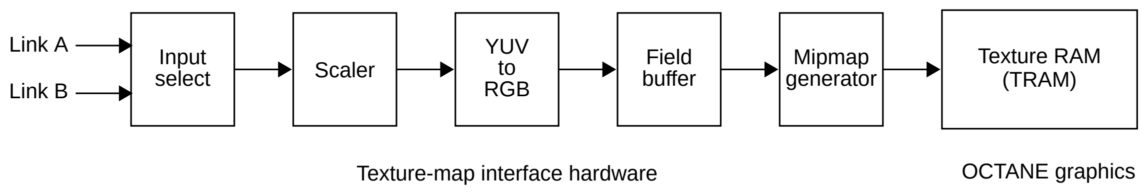


Figure 9-1 Video Texture-Mapping Hardware Configuration Block Diagram

This chapter covers the following topics:

- "Performing Video Texture Mapping"
- "Controls for Video Texture Mapping"
- "OpenGL Functions for Video Texture Mapping"
- "Example Program: vidtotex.c"

Performing Video Texture Mapping

Initially, you use VL calls to create a path between a source node and the texture drain node. Typically, the source node is a video input node. The texture drain node supports both single-link and dual-link transfers.

- For single-link transfers, use the VL_MGV_TEXTURE_INPUT_LINK control to specify the input link that supplies the data sent to TRAM. In addition, you can use the VL_MGV_TEXTURE_AUTOSWAP control to make the input link selection toggle automatically after each capture into TRAM. This feature is useful for applications that require two live video textures simultaneously.
- For dual-link transfers, an RGBA stream is sent to TRAM. In this mode, the RGB data is extracted from link A and the alpha data is extracted from link B.

Once the path is set up, use the **vlBeginTransfer()** routine. At this point, data is being transferred from the video source to the texture drain. However, to capture this data into the TRAM on the OCTANE graphics board, you must explicitly tell OpenGL to do it; otherwise the data is simply lost.

The OpenGL function **glCopyTexSubImage2DEXT()** captures into TRAM the data that the VL is transferring. To specify to this function that a video transfer is occurring, the GLX read drawable must be set to a video input stream. The function **glXCreateGLXVideoSource()** creates a handle for a video input stream. This handle can then be passed as the read-drawable parameter to the function **glXMakeCurrentReadSGI()**. Once this handle is passed, each call to **glCopyTexSubImage2DEXT()** begins a capture into TRAM of the data that is being sent to the VL texture drain node. It is important to realize that this function begins a capture into TRAM and then returns to the caller. Thus, the caller can continue executing while the video field is being captured.

To use the simplest example, you create a loop that calls **glCopyTexSubImage2DEXT()**, and then draw with the captured video field. To display completely drawn fields, use double-buffered drawing. Even if the drawing routine is very fast, however, this implementation can capture only every other video field that is being sent to the texture node. Although the framebuffer used for drawing is double buffered, the texture buffer itself is not. Any attempt to draw with the single-buffered texture causes the OpenGL draw routine to wait for the video field to finish loading into TRAM. When this load finishes, the drawing begins. However, the video field being sent to the texture node while the drawing is occurring is not captured.

The solution to this problem is to double-buffer the TRAM loads. Then, while a video field is being captured into one of the texture buffers, the other one can be used for drawing. This overlapping of loading one video field into TRAM at the same time that another one is being used for drawing enables you to do real-time video texturing. You can use the OpenGL function **glHint()** to enable double-buffer texture loads on an OCTANE graphics system that has enough TRAM to support it.

Note: Double-buffered video texture applications require 4 MB of TRAM, which you can obtain by installing the Texture Upgrade to 4 MB of Texture Memory for OCTANE graphics.

Real-Time Mipmap Generation

If you want your application to use the texture node's ability to do real-time mipmap generation, several issues should be considered. Before a texture is created, the OpenGL minifying filter must be set to a filter that enables mipmapping. You set it by calling the OpenGL function **glTexParameteri(**GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, *filter***)** and passing one of the following values for *filter*:

- GL_NEAREST_MIPMAP_NEAREST
- GL_LINEAR_MIPMAP_NEAREST
- GL_NEAREST_MIPMAP_LINEAR
- GL_LINEAR_MIPMAP_LINEAR

When a texture is created using the OpenGL function **glTexImage2D()**, ten Level-Of-Details (LODs) must be defined (rather than just one for non-mipmap applications). These LODs must be sized so that LOD 0 is 512 horizontally by 256 vertically, and each subsequent LOD (except LOD 9) is half the size, both horizontally and vertically, of the preceding one. LOD 9 must be sized as 1 × 1.

Once the VL texture node control VL_MGV_TEXTURE_MIPMAP_MODE is set to VL_MGV_TEXTURE_MIPMAP_ON, each subsequent call to the OpenGL function **glCopyTexSubImage2DEXT()** generates all ten LODs for the current video field and loads them into TRAM.

To scale the entire video input field in mipmap mode, the zoom and aspect controls must be set correctly. Normally, you set the VL_ZOOM control to 1. For the VL_MGV_HASPECT control, the numerator would be set to 512 and the denominator set to the width of the active video field in pixels. For the VL_MGV_VASPECT control, the numerator would be set to 256 and the denominator set to the larger of 256 and the height of the active video field in lines.

Timing Issues

For real-time video texturing on an OCTANE workstation with the OCTANE Digital Video option, these two subsystems must be frame-locked. If they are not, you can still perform video texturing, but real-time performance cannot be guaranteed.

To frame-lock the graphics and video hardware, use *setmon*. Before running a video texture application, enter one of these commands:

- NTSC/525-line systems: **`/usr/gfx/setmon -Fe 1280x1024_59`**
- PAL/625-line systems: **`/usr/gfx/setmon -Fe 1280x1024_49`**

The texture map interface hardware on the OCTANE Digital Video board contains a field buffer that introduces a one-field delay between the input source and the texture memory on the graphics board. This one-field delay is hidden by the fact that whenever you call the **glCopyTexSubImage2DEXT()** function to capture a video field, the actual capture does not occur until the beginning of the next video field. This delay occurs regardless of whether the call to **glCopyTexSubImage2DEXT()** occurs during the video vertical blanking interval or during the active video interval.

For example, the */usr/share/src/dmedia/video/impact/vidtotex.c* program is a simple application of video texturing. It captures each video input field into TRAM, and draws a flat polygon in an X window with it. In this example, the following actions occur:

1. The call to **glCopyTexSubImage2DEXT()** occurs when video field *n* is being sent into the texture-map interface input select hardware (see Figure 9-1).
2. When the next video field, *n* + 1, is being sent to the texture map interface hardware, field number *n* is being transferred out of the field buffer and captured into TRAM on the graphics board, as diagrammed in Figure 9-2.

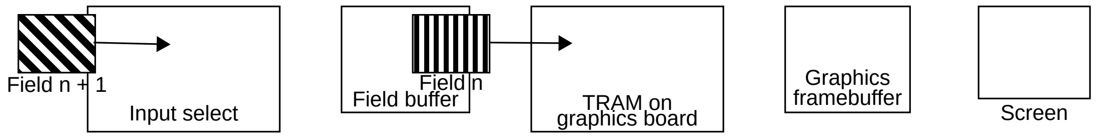


Figure 9-2 Video Delay Through Texture-Map Interface Field Buffer

3. During the video field time for field *n* + 2, field *n* is being transferred out of TRAM and into the graphics framebuffer, as diagrammed in Figure 9-3.

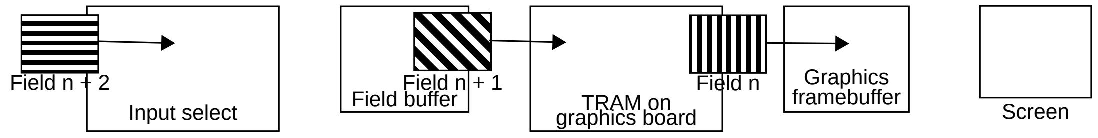


Figure 9-3 Video Delay Through Graphics TRAM

4. During the video field time for field *n* + 3, field *n* is displayed on the screen, as diagrammed in Figure 9-4.

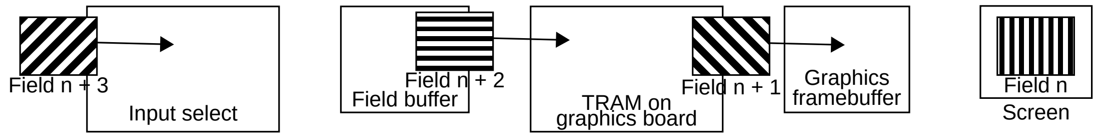


Figure 9-4 Video Delay Through Graphics Framebuffer

Thus, for this application, the call to **glCopyTexSubImage2DEXT()** results in the capture of the video field that is entering the texture map interface input select hardware at the time of the call. This field is displayed on the screen three video fields later.

Controls for Video Texture Mapping

This section describes the controls available on the VL texture drain node. Changing these controls is unrestricted; for example, they can be changed before or after **vlBeginTransfer()** has been called. However, they must match those that are set up through calls to OpenGL; otherwise, the call to **glCopyTexSubImage2DEXT()** returns an error.

VL_CAP_TYPE

VL_CAP_TYPE specifies the type of field to capture even fields, odd fields, all fields, and non-interleaved frames. Specifying even or odd fields causes each call to OpenGL to initiate a capture to block until the correct field type is present.

If you specify the non-interleaved frame capture type, use the field dominance control (see VL_MGV_DOMINANCE_FIELD, below) to determine the field of the frame to capture first.

VL_OFFSET

VL_OFFSET specifies the upper left corner of a subregion of the active video region used to generate the texture. The *x* value is the horizontal offset from the left-hand edge in pixels, and the *y* value is the vertical offset from the top edge in lines. Along with VL_SIZE, this control can be used to extract any subregion of the input video field.

VL_OFFSET operates on the unzoomed (undecimated) image; it does not change if the zoom factor is changed.

The minimum horizontal offset is 0. The minimum vertical offset depends on the video standard:

- NTSC/525-line: -13
- PAL/625-line: -18 if VL_CAP_TYPE is VL_CAPTURE_ODD_FIELDS; otherwise -19

The maximum horizontal and vertical offsets are restricted: unzoomed size plus offset must not extend past the end of the active video region. The horizontal offset must be an even number.

VL_PACKING

VL_PACKING specifies the current packing format. For single-link transfers, this control can be set to either VL_PACKING_RGB_8 or VL_PACKING_RGBA_8. For dual-link transfers, this control is always set to VL_PACKING_RGBA_8.

VL_RATE

VL_RATE is a read-only control that specifies the number of textures per second to capture into TRAM. Since each capture is initiated by a call to OpenGL, you should make at least the same number of calls per second as is specified by this control.

The VLTransferComplete event is generated at the end of a capture into TRAM. The VLSequenceLost event is generated each time a capture into TRAM is missed. However, these events are reported only if they are enabled via the **vlSelectEvents()** routine.

VL_SIZE

VL_SIZE specifies the amount of the image to send to texture RAM; that is, how much clipping takes place. The *x* value is interpreted as the subregion width in pixels, and the *y* value is interpreted as the subregion height in lines.

This control operates on the decimated image. For example, when the image is decimated to half size using VL_ZOOM, the limits of the VL_SIZE control change by a factor of 2.

Along with the VL_OFFSET control, VL_SIZE can be used to extract any subregion of the input video field. Controlling size is useful for cropping bad data at the edges of the video region.

The minimum width is 4 pixels; the minimum height is 3 lines. The maximum width is equal to the active line length multiplied by the product of the zoom and the horizontal aspect values. The maximum height is equal to the total field height minus 4, multiplied by the product of the zoom and the vertical aspect values. The width must be an even number.

```
maxwidth = active_line_length x (zoom x horiz_aspect)
maxheight = (total_field_height - 4) x (zoom x vert_aspect)
```

VL_ZOOM

VL_ZOOM specifies the amount of scaling applied to the video subregion used as a texture source. Because only decimation is allowed on the texture node, the zoom value is always less than or equal to 1. For different zoom factors along the horizontal and vertical axes, use VL_MGV_HASPECT and VL_MGV_VASPECT controls. The zoom factor along each axis is equal to the product of the VL_ZOOM value and the aspect value for that axis.

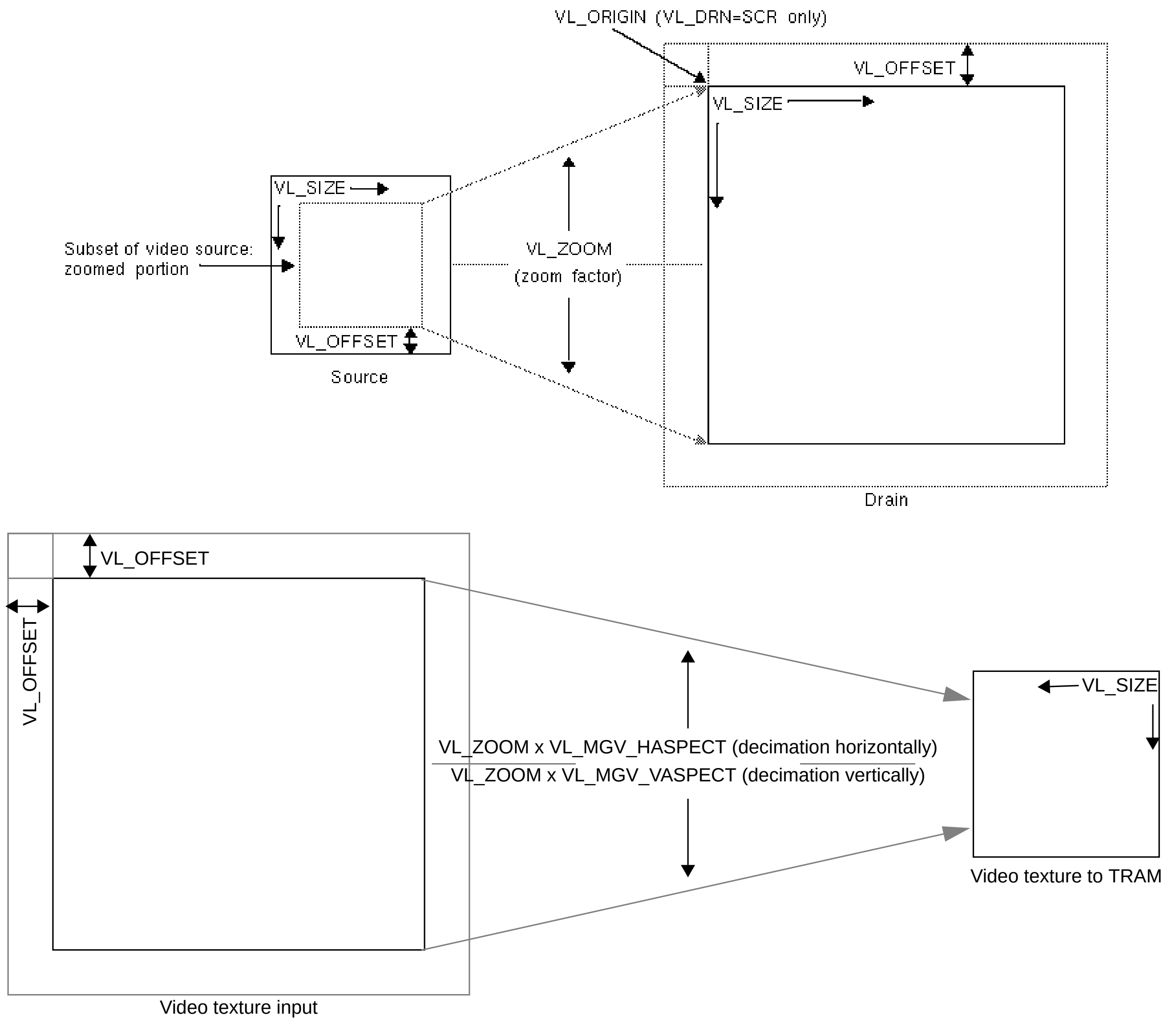

VL_ORIGIN (VL_DRN=SCR only)
VL_OFFSET
VL_SIZE
VL_SIZE
Subset of video source: zoomed portion
VL_ZOOM (zoom factor)
VL_OFFSET
Source
Drain
VL_OFFSET
VL_OFFSET
VL_ZOOM x VL_MGV_HASPECT (decimation horizontally)
VL_ZOOM x VL_MGV_VASPECT (decimation vertically)
VL_SIZE
Video texture to TRAM
Video texture input

Figure 9-5 Zoom, Size, and Offset for Video Texture Mapping

VL_MGV_DOMINANCE_FIELD

VL_MGV_DOMINANCE_FIELD specifies the field dominance when VL_CAP_TYPE is set to non-interleaved frames. In that case, VL_MGV_DOMINANCE_FIELD specifies the field of the frame to be captured into TRAM first.

VL_MGV_HASPECT

VL_MGV_HASPECT specifies the scaling along the horizontal axis that is applied to the video subregion used as a texture source. The overall scale factor horizontally is the product of this value and the VL_ZOOM value. This control enables you to scale the horizontal axis independently of the vertical axis, which is useful when mipmapping is enabled.

To scale the entire video input field horizontally when mipmapping is on, set the zoom to 1, and then set the numerator for VL_MGV_HASPECT to 512 and the denominator to the width of the video field in pixels.

VL_MGV_VASPECT

VL_MGV_VASPECT specifies the scaling along the vertical axis that is applied to the video subregion used as a texture source. The overall scale factor vertically is the product of this value and the VL_ZOOM value. This control enables you to scale the vertical axis independently of the horizontal axis, which is useful when mipmapping is enabled.

To scale the entire video input field vertically when mipmapping is on, set the zoom to 1, and then set the numerator for VL_MGV_VASPECT to 256 and the denominator to the larger of 256 and the height of the video field in lines.

VL_MGV_TEXTURE_ROUND_MODE

VL_MGV_TEXTURE_ROUND_MODE specifies the type of rounding used to convert from 10-bit to 8-bit input. You can either round

- normally: VL_MGV_TEXTURE_ROUND_8BIT
- with a pseudonumber generator: VL_MGV_TEXTURE_ROUND_RNG
- with a pseudonumber generator that resets each field: VL_MGV_TEXTURE_ROUND_RNGFRM

VL_MGV_TEXTURE_MIPMAP_MODE

The VL_MGV_TEXTURE_MIPMAP_MODE control turns mipmap mode off or on. When mipmap mode is on, the texture node generates mipmaps sized as follows:

- 512×256
- 256×128
- 128×64
- 64×32
- 32×16
- 16×8
- 8×4
- 4×2
- 2×1
- 1×1

VL_MGV_TEXTURE_INPUT_LINK

VL_MGV_TEXTURE_INPUT_LINK specifies the input link, A or B, that sends pixel data to TRAM. This control is available only in single-link mode.

VL_MGV_TEXTURE_AUTOSWAP

VL_MGV_TEXTURE_AUTOSWAP turns autoswap mode off or on. When autoswap mode is on, the input link toggles automatically after each capture into TRAM.

If VL_CAP_TYPE is set to non-interleaved frame, toggling occurs after frame captures. VL_MGV_TEXTURE_AUTOSWAP is useful for applications that require two live video textures simultaneously, and is available only in single-link mode.

OpenGL Functions for Video Texture Mapping

This section explains how to use OpenGL functions for video texture mapping. For complete information on these functions, see the *OpenGL Programming Guide*.

- **glGenTexturesEXT**(GLsizei *count*, GLuint **texnames*)

  This function generates texture names, which are unsigned integers, and puts count texture names in *texnames*. The generated texture names have no dimensionality, but assume that of the texture target to which they are first bound (see **glBindTexureEXT()**, below).

  For applications that use more than one video texture at a time (for example, double-buffering the TRAM loads), use this function to get the names of textures so that they can later be passed to **glBindTexturesEXT()** in order to identify them.

- **glBindTextureEXT**(GLenum *target*, GLuint *texname*)

  This function makes it possible to use named textures besides the usual OpenGL texture targets. While a texture is bound, GL operations on the target to which it is bound affect the bound texture. For video texturing applications, *target* should always be GL_TEXTURE_2D.

- **glHint**(GLenum *target*, GLenum *mode*)

  You can control TRAM double buffering with hints (like other aspects of OpenGL behavior). To enable double buffering, call this function with *target* set to GL_TEXTURE_MULTI_BUFFER_HINT_SGIX and *mode* set to GL_FASTEST. To disable double buffering, use the same target, but set *mode* to GL_NICEST.

- **glEnable**(GLenum *capability*)

  Use this function to enable texturing. Setting *capability* to GL_TEXTURE_2D enables two-dimensional texturing.

- **glXCreateGLXVideoSourceSGIX**(Display **dpy*, int *screen*, VLServer *svr*, VLPath *path*, int *nodeClass*, VLNode *node*)

  This function creates a GLX handle for a video input stream that can be used as the read parameter on a call to **glXMakeCurrentReadSGI()**. Thereafter, the GL transfers video data into texture memory when **glCopyTexSubImage2DEXT()** is called.

  If any control that affects the transfer of video data is changed on the video transfer path for which a particular GLX video source was created, destroy the GLX video source and create a new one. Otherwise, the data read from the source will be undefined. The configuration of a GLX video source is static, and is fixed when the GLX video source is created.

- **glXMakeCurrentReadSGI**(Display **dpy*, GLXDrawable *draw*, GLXDrawable *read*, GLXContext *gc*)

  This function attaches a GLX context to separate read and write drawables. The handle returned by **glXCreateGLXVideoSourceSGIX()** can be passed as the read parameter. Thereafter, the GL transfers video data into texture memory when **glCopyTexSubImage2DEXT()** is called.

- **glTexParameteri**(GLenum *target*, GLenum *pname*, GLint *param*)

  This function assigns the value in *param* to the texture parameter specified by *pname* for the target given by *target*. To enable or disable mipmapping, set *target* to GL_TEXTURE_2D and *pname* to GL_TEXTURE_MIN_FILTER. To turn mipmapping off, set *param* to either GL_NEAREST or GL_LINEAR. To turn mipmapping on, set *param* to one of the following: GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, or GL_LINEAR_MIPMAP_LINEAR. See the *OpenGL Programming Guide* for a description of these parameters.

- **glTexImage2D**(GLenum *target*, GLint *level*, GLint *components*, GLsizei *width*, GLsizei *height*, GLint *border*, GLenum *format*, GLenum *type*, const GLvoid **pixels*)

  This function defines a texture image. The arguments describe the parameters of the texture image, such as height, width, level-of-detail number, and the internal resolution and format used to store the image.

  The height and width must be powers of two. When mipmapping is not used, only level-of-detail number 0 need be defined; otherwise all ten LODs must be defined. For video textures, *pixels* can be set to NULL, because the texture data itself is loaded through calls to **glCopyTexSubImage2DEXT()**.

- **glCopyTexSubImage2DEXT**(GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffset*, GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*)

  This function replaces a rectangular portion of a two-dimensional texture image with pixels from the current GL_READ_BUFFER. The screen-aligned pixel rectangle with lower left corner at (*x*,*y*) having width *width* and height *height* replaces the portion of the texture array with *x* indices *xoffset* through *xoffset* + *width* - 1, inclusive, and *y* indices *yoffset* through *yoffset* + *height* - 1, inclusive, at the mipmap level specified by *level*.

  However, for video texture loads, to load all 10 mipmap levels, enable mipmapping using **glTexParameteri()** and call this function with *level* set to 0. For video textures, this setting initiates a load into texture memory from the video port.

  If any OpenGL texture parameters do not match the corresponding parameters set up through the Video Library, no load occurs and the GL error flag (see **glGetError()**), is set accordingly.

- **glGetError()**

  This function returns the value of the error flag. Each detectable error is assigned a numeric code and a symbolic constant. When an error occurs, the error flag is set to the appropriate error code value. No other errors are recorded until **glGetError()** is called, the error code is returned, and the flag is reset to GL_NO_ERROR.

  For video texture loads, a mismatch between the OpenGL parameters and the Video Library parameters generates an error that can be detected with this function.

Example Program: vidtotex.c

The example program */usr/share/src/dmedia/video/impact/vidtotex.c* continuously captures a stream of live video from a video source and repeatedly loads it into an OpenGL texture map. Those texture maps are then used to texture a polygon in an X window on the graphics display.

The video source can be selected via a command-line option, or via the default input device control on *vcp*. The output of *vlinfo* shows valid video source numbers and their mappings.

Return Codes

This appendix explains the return codes that are used with the Video Library for the OCTANE Digital Video option. The return code is accessible through the **vlGetErrno()** routine; see also **vlPerror()** and **vlStrError()**.

`VLAPIConflict`

> You have called an API routine that is not supported on this platform.

`VLSuccess`

> The Video Library routine completed without error.

`VLBadAccess`

> The client attempted to perform an operation that is illegal given the state of the client, the node, or the path. This error is returned, for example, if the client attempts to add a node to a path that has been set up, or call **vlSetControl()** on a path with control usage est to VL_READ_ONLY.

`VLBadAlloc`

> The Video Library could not allocate the system resources required for the requested operation, for example, memory and semaphores. If the source of the error is not evident (that is, sufficient physical memory and paging space was present), report this error to technical support.

`VLBadAtom`

> The server does not recognize the value specified by the atom parameter in the request as a valid atom ID.

`VLBadBuffer`

> The value of the buffer parameter is not a DMbuffer ID recognized by the Video Library.

`VLBadControl`

The value specified by the control parameter is not recognized by the node the request was made to.
The node to which the request was made does not recognize the value specified by the control parameter for **vlSetControl()** or **vlGetControl()**.

`VLBadDevice`

The server does not recognize the value specified by the device parameter in the request as a valid device ID. See also `VLBadMatch`.

`VLBadIDChoice`

The requested resource ID is not in range; report this error to Customer Support.

`VLBadImplementation`

An internal processing error occurred. Report the error and the context in which it occurred to Customer Service.

`VLBadIoctl`

An error occurred between the video daemon and the device driver associated with the video device. This error can result from an invalid parameter setting in **vlSetControl()**, although it can also represent an internal processing error. This error should be reported to technical support.

`VLBadLength`

The video daemon received a request with an invalid length. Report this internal processing error to Customer Support.

`VLBadMatch`

The arguments specified for the node, path, or device parameters are not consistent. The node may not reside on the path, or the path may not reside on the device.

`VLBadName`

An error took place when the DMparams list was to be retrieved; see VLDMGETPARAMS(3dm).

`VLBadNode`

The server does not recognize the value specified by the node parameter in the request as a valid node ID. See also `VLBadMatch`.

`VLBadPath`

The server does not recognize the value specified by the path parameter in the request as a valid path. See also `VLBadMatch`.

`VLBadPort`

The value specified by the port parameter is not a recognized port on the associated node.

`VLBadRequest`

The daemon has received a bad request code. Report this internal processing error to technical support.

`VLBadServer`

The value of the server parameter is not a server ID recognized by the Video Library.

`VLBadSize`

The size of the DMbuffer elements associated with a memory node are not compatible with the size of a video unit (field or frame), given the node's control settings.

`VLBadValue`

The value of a parameter is invalid. When generated by **vlGetControl()**, VLBadValue can indicate that the incorrect control value type was used, that the value is not within the range for the control, or that the node cannot accept the specified value due to a conflict with other node settings.

`VLBadWinAlloc`

vlSetupPaths() can return this code if there is insufficient screen space to place a screen source or drain. See Appendix B, "OCTANE Digital Video Nodes and Their Controls," for placement constraints.

`VLBufferTooSmall`

The size of the DMbuffer elements associated with a memory node are smaller than the size of a video unit (field or frame) given the node's control settings.

`VLInputsNotLocked`

The processing element associated with a node cannot lock to the input signal. This code may indicate that no signal is present or that the supplied video signal uses a different timing standard than that expected by the node (see VL_TIMING on the input or device node).

`VLNoRoute`

vlSetConnection() can return this error if no route could be found from the source (node, port) to the drain (node, port). The probable reason is that all connector resources are in use.

`VLNotEnoughSpace`

The supplied data region did not contain enough space to hold the information returned by the server.

`VLNotSupported`

vlSetConnection() or **vlGetConnection()** can return this code if the video device does not support explicit connections.

`VLPathInUse`

This error is generated if a required resource, for example a node (**vlSetupPaths()**) or a connector (**vlSetConnection()**) cannot be acquired.

- In the case of **vlSetupPaths()**, the node cannot be acquired because the path has requested VL_SHARE stream usage while another path has the required nodes with a stream of VL_LOCK, or the path has requested VL_SHARE control usage while another path has the required nodes with a control usage of VL_LOCK.
- For **vlSetConnection()**, the paths using the required connector could not be preempted because the application has requested that no preemption occur, or because a path using the connector has stream usage set to VL_LOCK.

`VLSetupFailed`

A general failure occurred during a **vlSetupPaths()** request. If multiple paths were specified for **vlSetupPaths()**, some or none of the paths may have been set up. In addition, some paths may have been preempted in order to set up those paths.

It is recommended that the application set up the paths again to stream usage VL_READ_ONLY and control usage VL_READ_ONLY or VL_SHARE in order to reset the state of all paths. This combination of control and stream usage is guaranteed to succeed.

`VLValueOutOfRange`

The control value specified for a **vlSetControl()** operation is not within the range accepted by the node. The value was adjusted before being set. (Compare with `VLBadValue`, where the control's value is not changed at all.) Use the **vlGetControlInfo()** routine to retrieve the valid ranges for the control.

Appendix B

OCTANE Digital Video Nodes and Their Controls

This appendix describes the nodes available to the OCTANE Digital Video option. It lists the ports and controls associated with each node, as well as special considerations involved in node usage.

In the tables that summarize the control set for a node, the columns are as follows:

Default	The default value for the control. If the value is Dynamic, the default value depends on the value of other controls. For example, frame size is dependent on device timing. The default value is described in the verbose description of the control.

If the value is Persistent, the default value is initially obtained from the defaults file, but is never reset. Many controls available through the video control panel *vcp* (for example, the default video input) fall into this category. For this value, changes made by **vlSetControl()** are persistent across paths, even if the node goes into an unused state.

If the default is a specific value or is Dynamic, the control is reinitialized to the default value when the node is no longer in use, that is, when all application paths have been destroyed and the only applications remaining are supervisory. At present, the *vcp* is the only supervisory application.

Some controls, such as VL_WINDOW, have a default value of None. This value means that the control *must* be set before a transfer can be started on a path containing the node.

Type	The member of the VLControlValue union used to set or get the value of the control.
Access	Access is one or more of the following: • G: The value can be retrieved through **vlGetControl().** • S: The value can be set through **vlSetControl()** while the path is not transferring. • T: The value can be set through **vlSetControl()** while the path is transferring.

The nodes are as follows:

- VL_DEVICE
- VL_BLENDER
- VL_CSC: color-space conversion node
- VL_FB: internal framebuffer node for freezing video
- VL_MEM: region of workstation memory
- VL_SCREEN: workstation screen
- VL_TEXTURE: texture node
- VL_VIDEO: connection to a video device; for example, a video tape deck or camera

Chapters in this guide explain these specific nodes:

- VL_BLENDER: Chapter 7, “Blending, Keying, and Transitions”
- VL_CSC: Chapter 8, “Using Color-Space Conversion”
- VL_TEX: Chapter 9, “Using Video Texture Mapping”

VL_DEVICE

The device node (digital video source node) provides controls that affect the operation of the OCTANE Digital Video device as a whole. These controls include global parameters such as timing, as well as default information such as the default source or drain.

For device nodes:

- type is VL_DEVICE
- kind is 0
- number is 0
- port is none

Table B-1 lists device node controls. For all these controls, access is GST, except VL_MGV_TRIGGER_WAIT, which is G only.

Table B-1 Device Node Controls

Control	Default	Type	Use
VL_DEFAULT_DRAIN	Persistent	intVal	The VL_DEFAULT_DRAIN control determines the drain node the Video Library selects when a node is acquired with **vlGetNode**(VL_DRAIN, VL_VIDEO, VL_ANY). The value of the control is a video drain node number, as reported by **vlGetDeviceList()**. Once a path is set up, the node number is fixed for the lifetime of the path. Consequently, changing this control does not change paths previously set up using a default drain node. Paths can register for the VLDefaultDrain event to be notified when this control's value is changed.
VL_DEFAULT_SOURCE	Persistent	intVal	The VL_DEFAULT_SOURCE control determines the source node the Video Library selects when a node is acquired with **vlGetNode**(VL_SRC, VL_VIDEO, VL_ANY). The value of the control is a video source node number, as reported by **vlGetDeviceList()**. Once a path is set up, the node number is fixed for the lifetime of the path. Consequently, changing this control does not change paths previously set up using a default source node. Paths can register for the VLDefaultSource event to be notified when this control's value is changed using **vlSelectEvents()**.

Table B-1 (continued) Device Node Controls

Control	Default	Type	Use
VL_SYNC	Persistent	intVal	The OCTANE Digital Video device can derive timing from an external source or use an internal free-running clock. If VL_SYNC is set to VL_SYNC_INTERNAL, the internal timing source is used. When VL_SYNC is set to VL_SYNC_GENLOCK, timing is derived from an external clock selected by the VL_SYNC_SOURCE control.
VL_SYNC_SOURCE	Persistent	intVal	When the VL_SYNC control is set to VL_SYNC_GENLOCK, this control selects the source of synchronization for the OCTANE Digital Video device. The device can accept external timing from GEN_PORTanalog reference input GEN_DIN1serial digital input 1 GEN_DIN2serial digital input 2
VL_TIMING	Persistent	intVal	This control selects the input timing for the OCTANE Digital Video device and affects the timing for all nodes. The device supports the following modes: VL_TIMING_525_SQ_PIX: NTSC, 525-line square pixel timing VL_TIMING_525_CCIR601: CCIR 601, 525-line non-square pixel timing VL_TIMING_625_SQ_PIX: PAL, 625-line square pixel timing VL_TIMING_625_CCIR601: CCIR 601, 625-line non-square pixel timing Square pixel modes are used with the OCTANE Compression option only.

Table B-1 (continued) Device Node Controls

Control	Default	Type	Use
VL_MGV_AUTOPHASE	Persistent	intVal	Autophasing allows the video inputs to be locked to each other if, on input, their phase differences are not too great. This control selects whether input autophasing is enabled and, if enabled, the type of autophasing performed. The options for this control are as follows: VL_MGV_AUTOPHASE_NORMAL: This mode synchronizes the video inputs to the reference input. It can accommodate a vertical interval switch with a maximum deviation of +/- 1/2 line from the reference. VL_MGV_AUTOPHASE_EXTENDED: This extended autophase mode allows frequency-locked inputs to be +/- 4 lines relative to the reference input. VL_MGV_AUTOPHASE_VARIABLE: The variable mode allows the two inputs to be nonsynchronous with the reference and can accommodate inputs that are offset by up to +/- 4 lines. Variable autophasing uses the clock from the "last" input as the clock for both video channels. The last input is determined by monitoring the field (F) bit of channel 1 and 2. If channel 2 is already in the odd field when channel 1 enters the odd field, channel 1 is assumed to be the latest input. VL_MGV_AUTOPHASE_OFF: Autophasing is disabled; signals are passed through with their original timing.
VL_MGV_INPUT _ALPHA_LUT_SELECT	VL_MGV _ALPHA_LUT _CCIR601	intVal	This control selects the type of LUT to be used when video data is routed into the crosspoint mux (see "Getting Connections" in Chapter 5) to a blender alpha input. The LUT is generally used to expand limited-range data to full-range value for use as alpha. The following LUTs are available: VL_MGV_ALPHA_LUT_PASS: A pass-through LUT for use when the input is video (not alpha). Values 2-253 are passed unmodified. Input values 254 and 1 are mapped to 255 and 0, respectively. VL_MGV_ALPHA_LUT_CCIR601: Used to expand CCIR-range input to full-range input. Values >=235 are mapped to 255, values <=16 are mapped to 0, and values 17-234 are mapped to 1-254. VL_MGV_ALPHA_LUT_SUPERBLACK: Used for inputs that have been quantized with setup and the key extends into the blanking level. Values >=235 are mapped to 255, value 1 to 0, and values 2-234 to 1-254. VL_MGV_ALPHA_LUT_REDUCED_RANGE: Used for input keys that do not extend the full range. Values >=224 are mapped to 255, values <=32 to 0, and values 32-221 to 1-254. This control takes effect only when video is fed from the VBAR mux to the blender alpha input; when video is fed into a pixel input, then VL_MGV_ALPHA_LUT_PASS is always selected.

Table B-1 (continued) Device Node Controls

Control	Default	Type	Use
VL_MGV_OUTPUT _ALPHA_LUT_SELECT	VL_MGV _ALPHA_LUT _CCIR601	intVal	This control selects the LUT to use when alpha data is routed from the blender out of the crosspoint mux. Its function is to map the full-range alpha values to CCIR or other range values. The values for this control are the same as for VL_MGV_INPUT_ALPHA_LUT_SELECT, with the inverse mappings applied.
VL_MGV_TRIGGER _LINE	Persistent	intVal	This control reports the line on which the GPI trigger input is sampled. The OCTANE Digital Video device samples the trigger on a per-field basis at lines 4 and 266 for NTSC (reported as line 4 by **vlGetControl()**) and lines 1 and 313 for PAL (reported as 1 by **vlGetControl()**).
VL_MGV_TRIGGER _POLARITY	Persistent	intVal	This control selects the polarity, indicating that the GPI trigger has gone off. Valid values are as follows: POLAR_NEG: The GPI trigger input has become negative. POLAR_POS: The GPI trigger input has become positive.
VL_MGV_TRIGGER _WAIT Default: none	None	boolVal	Allows a client to “sniff” the GPI trigger. A **vlGetControl()** call with VL_MGV_TRIGGER_WAIT blocks the client application until the trigger goes off. While the application is blocked, events received from the video daemon are queued on the client’s connection to the daemon. For an application to be notified when the trigger fires, it must be registered for VLDeviceEvent. Using the event mechanism has the advantage that the client application can continue to interact with the Video Library or perform local processing while waiting for the event. VL_MGV_TRIGGER_WAIT is provided for applications that require a faster response to the trigger than can be provided with VLDeviceEvent.

VL_BLENDER

The blender node provides two-layer blending and keying. The foreground pixel input (PIXEL_DRN_A) can be used as an input to the keyer to provide for chroma and luma key generation. The blender can also be used with external pixel and alpha sources to perform user-defined blend operations. For the blender node:

- type is VL_INTERNAL
- kind is VL_BLENDER
- number is VL_MGV_NODE_NUMBER_BLENDER
- ports are
  - VL_IMPACT_PORT_PIXEL_DRN_A, foreground pixel input
  - VL_IMPACT_PORT_ALPHA_DRN_A, foreground pixel input
  - VL_IMPACT_PORT_PIXEL_DRN_B, background pixel input
  - VL_IMPACT_PORT_ALPHA_DRN_B, background pixel input
  - VL_IMPACT_PORT_PIXEL_SRC_A, pixel output
  - VL_IMPACT_PORT_ALPHA_SRC_B, alpha output

The blender is the only node that can use the screen node alpha information. To use the screen alpha you must route the screen pixel data to blender alpha inputs. The blender alpha output can then be sent to the video outputs or memory.

Note: The blender operates only on YUV:4:2:2 8 bit video. When sending the blender output to video, it is best to blank the chroma.

The OCTANE Digital Video blender is a Porter-Duff style blender; see “Setting Normalization” in Chapter 7 for more information.

The blender does not stop you from doing special-effects blends. The output is clipped into the standard range. For a nice effect, try looping the output of the blender through a framebuffer back to the input of the blender in various modes.

Note: For more information on blending and keying, see Chapter 7.

Table B-2 lists blender controls. Access for all controls is GST. For more information on blender node controls, see Chapter 7.

Table B-2 Blender Node Controls

Control	Default	Type	Use
VL_BLEND_A_FCN	VL_BLDFCN_MINUS _A_ALPHA	intVal	Sets blend function that controls mixing of foreground signals.
VL_BLEND_B_FCN	VL_BLDFCN_ONE	intVal	Sets blend function that controls mixing of background signals.
VL_BLEND_A	Dynamic	intVal	Sets input source for foreground image.
VL_BLEND_B	Dynamic	intVal	Sets input source for background image.
VL_BLEND_A_ALPHA	Dynamic	intVal	Sets input source for foreground alpha.
VL_BLEND_B_ALPHA	Dynamic	intVal	Sets input source for background alpha.
VL_BLEND_A_NORMALIZE	TRUE	boolVal	Sets normalization; off is not supported by the OCTANE Digital Video option.
VL_BLEND_B_NORMALIZE	TRUE	boolVal	Sets normalization, following Porter-Duff model (background pixels premultiplied by their corresponding alphas before blending).
VL_MGV_KEYER_MODE	Persistent	intVal	Selects “master” keyer control that determines the type of keying performed (luma, chroma, or spatial).
VL_MGV_KEYER_DETAIL	Persistent	intVal	Sets sharpness of transition between foreground and background allowing blurring of edges. The value -8 yields the most gradual transition, +7 the sharpest.
VL_MGV_KEYER_FG_OPACITY	Persistent	intVal	Sets opacity of the foreground, thus limiting the value of foreground alpha at any point.
VL_MGV_KEYER_VALUE_LUMA	Persistent	intVal	Sets central luma value. This control sets the luma value at which the background shows through the foreground.
VL_MGV_KEYER_RANGE _LUMA	Persistent	intVal	Sets one-sided range of the center value. This control determines the range of luma values where the background shows through the foreground.
VL_MGV_KEYER_VALUE _CHROMA_U	Persistent	intVal	Sets central U value at which the background shows through the foreground.

Table B-2 (continued) Blender Node Controls

Control	Default	Type	Use
VL_MGV_KEYER_RANGE _CHROMA_U	Persistent	intVal	Sets one-sided range of U where the background shows through the foreground.
VL_MGV_KEYER_VALUE _CHROMA_V	Persistent	intVal	Sets central V value at which the background shows through the foreground.
VL_MGV_KEYER_RANGE _CHROMA_V	Persistent	intVal	Sets one-sided range of V where the background shows through the foreground.
VL_MGV_WIPE_SYMMETRY	FALSE	intVal	Sets 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_MGV_WIPE_CENT for single, double, and corner wipes; enables VL_MGV_WIPE_CENT_PERP control for double and corner wipes.
VL_MGV_WIPE_INVERT	FALSE	intVal	Reverses 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.
VL_MGV_WIPE	Persistent	intVal	Sets autowiper on.
VL_MGV_WIPE_TYPE	Persistent	intVal	Selects type of blending (wipe) performed.
VL_MGV_WIPE_ANGLE VL_MGV_WIPE_DIRECTION	Persistent	intVal	Sets wipe vector direction, that is, the direction in which the wipe appears to be proceeding as its position increases. Note that VL_MGV_WIPEANGLE_N and VL_MGV_WIPEANGLE_S do not work for the wipe types VL_MGV_WIPETYPE_DOUBLE and VL_MGV_WIPETYPE_CORNER.
VL_MGV_WIPE_SHARPNESS	Persistent	intVal	Sets sharpness of wipe transition band. As for VL_MGV_KEYER_DETAIL, -8 is most gradual, +7 is sharpest.
VL_MGV_WIPE_FUZZ	Persistent	intVal	Same as VL_MGV_WIPE_SHARPNESS.

Table B-2 (continued) Blender Node Controls

Control	Default	Type	Use
VL_MGV_WIPE_SPEED	Persistent	intVal	Sets speed at which the autowiper sweeps the wipe. The value is the speed of the wipe in units of number of fields for each wipe position change.
VL_MGV_WIPE_POSN	Persistent	fractVal	Sets amount of progress of wipe, from none (numerator = 0) to full (numerator = 1000).
VL_MGV_WIPE_POSN_PERP	Persistent	fractVal	Sets amount of progress of wipe, from none (numerator = 0) to full (numerator = 1000), along a direction perpendicular to normal wipe position VL_MGV_WIPE_POSN.
VL_MGV_WIPE_CENT	Persistent	intVal	Sets offset that is center of a symmetrical wipe along wipe position. 0 means center is where VL_MGV_WIPE_POSN is 0, and 1000 means center is where VL_MGV_WIPE_POSN is 1000. For this control to work for single, double, and corner wipes, VL_MGV_WIPE_SYMMETRY must be on.
VL_MGV_WIPE_CENT_PERP	Persistent	intVal	Sets 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_MGV_WIPE_REPT	Persistent	intVal	Sets 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. This control does not apply to fades.
VL_MGV_WIPE_REPT_PERP	Persistent	intVal	Sets number of repetitions perpendicular to wipe direction for single, double, and corner wipes, and length of other side of rectangles for tile wipe.
VL_MGV_WIPE_EXT_TRIG	FALSE	boolVal	If set to TRUE, causes the trigger to initiate an automatic wipe (autowipe).
VL_MGV_WIPE_SPEED	10	intVal	Sets duration of an autowipe in tenths of a second.
VL_MGV_BLEND_B_FLAT	Persistent	intVal	Sets flat-background generator on, so that background pixel source is used for pixel timing only and live video from pixel source B goes to the blender.

Table B-2 (continued) Blender Node Controls

Control	Default	Type	Use
VL_MGV_BLEND_B_Y	Persistent	intVal	Sets value of background Y.
VL_MGV_BLEND_B_U	Persistent	intVal	Sets value of background U.
VL_MGV_BLEND_B_V	Persistent	intVal	Sets value of background V.
VL_MGV_BLEND_SHADOW_ON	Persistent	boolVal	Activates shadow hardware. See “Adding Shadows” in Chapter 7 for information.
VL_MGV_BLEND_SHADOW _GAIN	Persistent	intVal	Sets value shift for shadow.
VL_MGV_BLEND_SHADOW _OFFSET	Persistent	intVal	Adds to shadow value. Note that darkening a very light shadow can result in noise.
VL_MGV_BLEND_H_FILT	Persistent	boolVal	Sets horizontal smoothing filter that filters pixel information before the alpha extraction and smooths the alpha output of the key generator.

VL_CSC

The color-space converter (CSC) node controls the color-space converter hardware. The CSC hardware can perform many image-processing operations on a video path, allowing VL control over the conversion process.

The CSC hardware can be used in two ways: standard conversions, or nonstandard conversions with fine control over the color-space hardware. In many cases, the standard conversions are sufficient. For example, the standard controls can set up a conversion from YCrCbA (CCIR range) to RGBA in a few calls.

For application developers who wish to use VL_CSC as an image processing device capable of posterization, solarization, or color correction, or for nonstandard conversions, nonstandard control commands are provided. To use these controls, you must understand how the hardware works and how color-space conversion is performed. See Appendix C, “OCTANE Digital Video Color-Space Conversions.”

Chapter 8, “Using Color-Space Conversion” provides information on using this node. For the color-space conversion node VL_CSC:

- type is VL_INTERNAL
- kind is VL_CSC
- number is VL_MGV_NODE_NUMBER_CSC

See Chapter 8 for video formats, ports, ranges, and packings for VL_CSC.

Table B-3 summarizes controls for standard color-space conversion. Access for all controls is GS.

Table B-3 Controls for Standard Color-Space Conversion

Control	Default	Type	Use
VL_MGV_CSC_IN_PACKING VL_MGV_CSC_OUT_PACKING	VL_PACKING_YVYU _422_10	intVal	Sets the packing for the color-space converter node’s input or output, respectively.
VL_MGV_CSC_IN_RANGE VL_MGV_CSC_OUT_RANGE	VL_FORMAT_DIGITAL _COMPONENT_SERIAL	intVal	Sets the input range (RP-175, CCIR, or full) associated with the input or output video, respectively.
VL_MGV_CSC_CONST_HUE	TRUE	boolVal	Enables or disables constant-hue algorithm.
VL_MGV_CSC_ALPHA _CORRECTION	FALSE	boolVal	When VL_MGV_CSC_CONST_HUE is enabled, this control saves the constant hue correction factor (TRUE) or retains alpha input data (FALSE).

Table B-4 summarizes image-processing (nonstandard) controls. Access for all these controls is GST.

Table B-4 Image-Processing Controls

Control	Default	Type	Use
VL_MGV_CSC_COEF	Multiplier operates in pass-through mode	extendedVal; data type MGV_CSC_COEF	Specifies the matrix multiplier coefficients
VL_MGV_CSC_LUT_IN_PAGE VL_MGV_CSC_LUT_ALPHA_PAGE	0	intVal	Selects the active LUT
VL_MGV_CSC_LUT_IN_YG VL_MGV_CSC_LUT_IN_UB VL_MGV_CSC_LUT_IN_VR VL_MGV_CSC_LUT_ALPHA	Pass-through (1:1 mapping)	extendedVal; data type MGV_CSC_LUT_INPUT _AND_ALPHA	Specifies the contents of the input or alpha lookup tables
VL_MGV_CSC_LUT_OUT_YG VL_MGV_CSC_LUT_OUT_UB VL_MGV_CSC_LUT_OUT_VR	Pass-through (1:1 mapping)	extendedVal data type MGV_CSC_LUT_OUTPUT	Specifies the contents of the output lookup tables

VL_FB

The framebuffer node provides a mechanism for freezing a video stream. This node is most useful when it is used with the video source nodes, which lack freeze capability. It is also suitable when a snapshot of a video stream is required and the application cannot freeze the input because the live feed is used elsewhere.

Note that the memory and screen source nodes have inherent freeze capability. For the framebuffer node:

- type is VL_INTERNAL
- kind is VL_FB
- number is VL_MGV_NODE_NUMBER_FB
- ports are
  - VL_IMPACT_PORT_PIXEL_SRC_A, 8-bit single-link output
  - VL_IMPACT_PORT_PIXEL_DRN_A, 8-bit single-link input

The framebuffer node imposes a one-frame delay on the video stream.

The framebuffer element is shared between this node and the CC1 memory source node. Consequently, only one of the two can be in use at a time. Attempts to set up both on a path with stream usage VL_SHARE or VL_LOCK result in the first path being preempted. If the framebuffer node and the CC1 memory source node are set up on the same path, an error is returned.

The framebuffer node is internal to the crosspoint mux. Consequently, to avoid consuming the (scarce) VBAR-crosspoint connectors, ensure that its use is required in the path. For example, this node is usually not needed to freeze the video output, since the video drain nodes have freeze capability.

The default control for this node is VL_FREEZE. If set to TRUE, this control freezes the video stream passing through the framebuffer. If set to FALSE, live video resumes. For this control, the default is FALSE, type is boolVal, access is GST.

VL_MEM

This discussion divides the VL_MEM nodes into their manifestations as source and drain.

VL_MEM Source

The OCTANE Digital Video option supports four memory source nodes: VGI1 1, VGI1 2, VGI1 DL, and CC1. The VGI1 memory sources provide real-time single- and dual-link paths from main memory to the OCTANE Digital Video option. For the memory source node:

- type for all four memory source nodes is VL_SRC
- kind for all four memory source nodes is VL_MEM
- number is VL_MGV_NODE_NUMBER_VGI_1, VL_MGV_NODE_NUMBER__VGI_2, and VL_MGV_NODE_NUMBER__VGI_DL, and VL_MGV_NODE_NUMBER__CC, respectively
- ports are
  - memory source nodes VGI1 1 and VGI1 2: VL_PORT_PIXEL_SRC_A: single-link 8- or 10-bit video stream capable of real-time operation
  - dual-link video source node: VL_PORT_DUALLINK_SRC_A: dual-link 8- or 10-bit video stream capable of real-time operation
  - memory source node CC1:
    VL_PORT_PIXEL_SRC_A: single-link 8-bit video stream, no 10-bit support, does not guarantee real-time operation, but has attached framebuffer

    The CC1 node is used mostly to support alpha for the blender. If the blender uses an image from this framebuffer, the Y value is interpreted as alpha and the blender uses it accurately. Data in this buffer allows you to put a nonrectangular shape (for example, a heart or an irregularly shaped logo) as a matte around an image.

    The CC1 memory source node is also useful for slide shows or other static-image situations in which video input changes only every 10 or 20 seconds and real-time performance is not critical. Because the CC1 node has its own framebuffer, there is no CPU overhead.

Table B-5 lists memory source node controls. For all these controls, access is GS.

Table B-5 Memory Source Node Controls

Control	Default	Type	Use
VL_CAP_TYPE	CAP_TYPE _INTERLEAVED	intVal	Specifies the type of video units—fields or frames—that the application obtains from the ring buffer. Valid capture types are VL_CAPTURE_NONINTERLEAVED, VL_CAPTURE_INTERLEAVED, VL_CAPTURE_EVEN_FIELDS, VL_CAPTURE_ODD_FIELDS, and VL_CAPTURE_FIELDS. See "Using VL_CAP_TYPE and VL_RATE" in Chapter 2 for information on capture types.
VL_FORMAT	VL_FORMAT_DIGITAL _COMPONENT_SERIAL	intVal	Specifies the type of video format to be produced. See "Using VL_FORMAT" in Chapter 2 for formats and explanations.
VL_PACKING	Single link: VL_PACKING_YVYU _422_8 Dual link: VL_PACKING_YUVA _4444_8	intVal	Specifies the bit order in which the video components are stored in memory. The native packings supported by VL_MGV_NODE_NUMBER_VGI_[1,2] are VL_PACKING_YVYU_422_8 and VL_PACKING_YVYU_422_10. Supported non-native single link packings (implemented automatically in software) are VL_PACKING_Y_8_P, VL_PACKING_RGB_332_P, and VL_PACKING_RGB_8. The native packings supported by VL_MGV_NODE_NUMBER_VGI_DL are VL_PACKING_YUVA_4444_8, VL_PACKING_YUVA_4444_10, VL_PACKING_AUYV_4444_8, VL_PACKING_AUYV_4444_10, VL_PACKING_RGBA_8, VL_PACKING_RGBA_10, VL_PACKING_ABGR_8, VL_PACKING_ABGR_10, and VL_PACKING_AYU_AYV_10. See "Using VL_PACKING" in Chapter 2 for the specifications of each packing.
VL_OFFSET	*(0,0)*	xyVal	Specifies the upper left corner of a video region to be output. The coordinates are offsets of the upper left corner of the active video and take precedence over the size. Therefore, in order to accommodate the given offset, the size may be changed. A VLControlChanged event is generated to inform interested parties of any change in size.

Table B-5 (continued) Memory Source Node Controls

Control	Default	Type	Use
VL_RATE	Dynamic; depends on timing and capture type	fractVal	Specifies the rate at which the hardware extracts video units (fields or frames, depending on the capture type) from the ring buffer. The video unit is repeated, or black is output, to achieve the video output rate of 60 fields per second (NTSC) or 50 fields per second (PAL). The VGI1 memory source nodes can consume video units from system memory at any rate up to the video standard rate. For VL_CAPTURE_NONINTERLEAVED and VL_CAPTURE FIELDS, valid ranges are as follows: NTSC: 1 through 60 units per second (must be multiple of fields per frame for noninterleaved) PAL: 1 through 50 units per second (must be multiple of fields per frame for noninterleaved) For VL_CAPTURE_INTERLEAVED, VL_CAPTURE_EVEN_FIELDS, or VL_CAPTURE_ODD_FIELDS, valid ranges are 1 through 30 units per second for NTSC and 1 through 25 units per second for PAL.
VL_SIZE	Dynamic; depends on timing and capture type	xyVal	Specifies the width (pixels) and height (lines) of the video data contained within each ring buffer entry. These values, along with VL_PACKING, determine the size in bytes of each ring buffer entry and thus the transfer size. The width must be a multiple of four pixels. The length must be a minimum of one line for field capture types and two lines for frames. The specified size is constrained by the maximum allowable (as dictated by the device timing) and by the current offset position (VL_OFFSET). If the size is too large, it is reduced. The offset is not changed. It is recommended that VL_OFFSET be set before VL_SIZE.
VL_TIMING	Dynamic; from device node	intVal	Retrieves the current device-wide video timing value. See "VL_DEVICE" in this chapter for more details. Setting this control on any other node type has no effect.
VL_ZOOM	1.0	fractVal	Specifies the amount of scaling to be applied to the video before it is transferred to memory. The VGI1 memory source nodes have no scaling ability. The only legal value is 1.0.

Table B-5 (continued) Memory Source Node Controls

Control	Default	Type	Use
VL_MGV _DOMINANCE _FIELD	VL_MGV_DOMINANCE _F1	intVal	Sets the field dominance mode, which determines the order in which the fields are read from memory. This control applies only to the frame-oriented capture types (VL_CAPTURE_INTERLEAVED and VL_CAPTURE_NONINTERLEAVED). For VL_CAPTURE_INTERLEAVED, values are as follows: VL_MGV_DOMINANCE_F1: For video timings VL_TIMING_525_CCIR601 and VL_TIMING_525_SQ_PIX, F1 (also known as odd) dominance dictates that data for the F1 field resides in memory *after* that for F2. For VL_TIMING_625_CCIR601 and VL_TIMING_625_SQ_PIX, the data for F1 resides in memory *before* that of F2. VL_MGV_DOMINANCE_F2: For VL_TIMING_525_CCIR601 and VL_TIMING_525_SQ_PIX, F2 (also known as even timings), dominance dictates that data for the F1 field resides in memory *before* that for F2. For VL_TIMING_625_CCIR601 and VL_TIMING_625_SQ_PIX, the data for F1 resides in memory *after* that of F2. The meaning of *before* and *after* depends on the capture type. For interleaved frames, *before* indicates that the data that compose the first line of the designated field begins at the first byte of the buffer. In this format, the lines of F1 and F2 are interleaved within the one ring buffer; thus the second line of the buffer belongs to the other field, and so forth. For noninterleaved frames, *before* indicates that the dominant field is in a buffer preceding the buffer(s) containing nondominant fields. For VL_CAPTURE_NONINTERLEAVED, values are as follows: VL_MGV_DOMINANCE_F1: The F1 field is in the first buffer of the pair, and the F2 field in the second. VL_MGV_DOMINANCE_F2: The F2 field is in the first buffer of the pair, the F1 field in the second.

Table B-5 (continued) Memory Source Node Controls

Control	Default	Type	Use
VL_MGV_BUFFER _QUANTUM	1	intVal	The granularity, or quantum, of data transfer required by the application. The video data is padded at the end so that the size of a field/frame is a multiple of VL_MGV_BUFFER_QUANTUM. This control is intended for applications that do I/O directly from the ring buffer, and may consequently require the frame or field size to be a multiple of the device block size. Direct I/O, for example, usually requires that 512 bytes of data be transferred at a time.
VL_MGV_DMA _ERROR_RESTART	VL_MGV_DMA_RESTART _ON	intVal	If enabled (VL_MGV_DMA_RESTART_ON), a video transfer continues when an error is encountered. Otherwise (VL_MGV_DMA_RESTART_OFF), the video transfer is aborted. This control covers three types of errors: The reference video timing is not clean, resulting in short/long lines, fields, or both. These errors are with respect to the programmed size and offset. The system GIO bus bandwidth was insufficient to transfer video from system memory at video rates. The video clock was interrupted.
VL_MGV_DMA _VOUT_EXPAND	VL_MGV_DMA_EXPD _OFF	intVal	Specifies whether or not 8-bit data read from memory is expanded to 10-bit data before being output by the DMA channel to the VBAR mux. If enabled (VL_MGV_DMA_EXPD_ON), then zeroes are inserted into the least significant two bits; otherwise (VL_MGV_DMA_EXPD_OFF), all 10 bits are output unmodified.

Table B-5 (continued) Memory Source Node Controls

Control	Default	Type	Use
VL_MGV_DMA _VOUT _STARVATION	VL_MGV_DMA_VO _STARV_RPT	intVal	Sets the video output policy to use when the memory node underflows the ring buffer (that is, the application has not filled the ring buffer at the rate that the memory node consumes it). An application can choose between two starvation policies. In each case, video output from system memory resumes when the application places the next field/frame in the ring buffer via **vlPutValid()**. VL_MGV_DMA_VO_STARV_BLK: Outputs black fields or frames. This choice does not involve further access to memory until a new buffer becomes available. VL_MGV_DMA_VO_STARV_FLD: Causes the last field output to be repeated. VL_MGV_DMA_VO_STARV_RPT: Repeats the last unit (field or frame) that was transferred from main memory. The repetition is performed by continuing to transfer the same field/frame from memory to video until a new buffer becomes available or the transfer is ended. This results in system bus bandwidth continuing to be consumed. Caution: In order to maintain compatibility with the behavior of the CC1 memory source node as well as the earlier Galileo Video™ products, where a framebuffer is incorporated, the default value for this control is VL_MGV_DMA_VO_STARV_RPT. Therefore the ring buffer used in the transfer must contain a minimum of two buffer entries (four for VL_CAPTURE_NONINTERLEAVED), so that one buffer can be repeated by the system while the application is filling the second. If only one buffer is used, then the first buffer output is repeated indefinitely and **vlGetNextFree()** never returns a free buffer.

VL_MEM Drain

The OCTANE Digital Video option supports three memory drain nodes: VGI1 1, VGI1 2, and VGI1 DL. The VGI1 memory drains provide real-time single- and dual-link paths from the OCTANE Digital Video device to ring buffers. For the memory drain:

- type for all three memory drain nodes is VL_DRN
- kind for all three memory drain nodes is VL_MEM
- number is VL_MGV_NODE_NUMBER_VGI1_1, VL_MGV_NODE_NUMBER__VGI1_2, and VL_MGV_NODE_NUMBER__VGI1_DL, respectively
- ports are as follows:
  - memory drain nodes VGI1 1 and VGI 2: VL_PORT_PIXEL_DRN_A: single-link source for 8- or 10-bit video stream capable of real-time operation
  - dual-link video drain node: VL_PORT_DUALLINK_DRN_A: dual-link source for 8- or 10-bit video stream capable of real-time operation

With the VL_MGV_DMA_VIN_ROUND control enabled, the components of the 10-bit video signal applied to the VGI1 memory drain are rounded to 8 bits; otherwise all 10 bits are passed through and written to memory.

With rounding disabled, setting an 8-bit packing while capturing 10-bit data truncates the data to 8 bits.

Figure B-1 shows the bit relationships for the CCIR 601 8- and 10-bit video format components

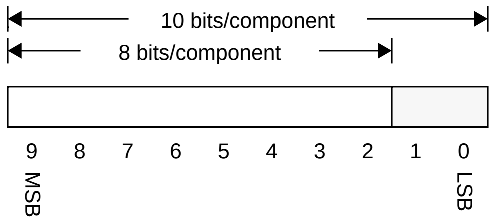


Figure B-1 Rounding for Memory Drain

When rounding is enabled, 10-bit data is converted to 8-bit data depending on the rounding type and the randomized rounding mode. In simple rounding, if bit 1 is set, then the value is rounded up (one is added to bit 2), otherwise it is rounded down.

Randomized rounding involves using a 22-bit shift register to generate two pseudo-random bits to be added to bits 1 and 0 of the 10-bit component, which may or may not result in a carry to bit 2.

The behavior of the shift register is dictated by the randomized rounding mode. With repeated randomized rounding, the shift register is initialized to the same value at the start of each odd (F1) field. Thus, the same pseudo-random sequence will be used for each frame. However, in free-wheel mode, the shift register is never reset and the sequence becomes totally random. The shift register is guaranteed never to become stuck at zero.

Rounding occurs only on active lines and during the digital active line between, and not including, SAV and EAV. The digital blanking data is not modified.

Table B-6 lists memory drain node controls. For all these controls, access is GS.

Table B-6 Memory Drain Node Controls

Control	Default	Type	Use
VL_CAP_TYPE	CAP_TYPE_ INTERLEAVED	intVal	Specifies the type of video units—fields or frames—that the application obtains from the ring buffer by the application. Valid capture types are VL_CAPTURE_NONINTERLEAVED, VL_CAPTURE_INTERLEAVED, VL_CAPTURE_EVEN_FIELDS, VL_CAPTURE_ODD_FIELDS, and VL_CAPTURE_FIELDS. (See “Using VL_CAP_TYPE and VL_RATE” in Chapter 2 for information on capture types.)
VL_FORMAT	Dynamic	intVal	Specifies the type of video format to be produced. (See “Using VL_FORMAT” in Chapter 2 for formats and explanations.)

Table B-6 (continued) Memory Drain Node Controls

Control	Default	Type	Use
VL_PACKING	Single link: VL_PACKING_YVYU _422_8 Dual link: VL_PACKING_YUVA _4444_8	intVal	Specifies the bit order in which the video components are stored in memory. The native packings supported by VL_MGV_NODE_NUMBER_VGI_[1,2] are VL_PACKING_YVYU_422_8 and VL_PACKING_YVYU_422_10. Supported non-native single link packings (implemented automatically in software) are VL_PACKING_Y_8_P, VL_PACKING_RGB_332_P, and VL_PACKING_RGB_8. The native packings supported by VL_MGV_NODE_NUMBER_VGI_DL are VL_PACKING_YUVA_4444_8, VL_PACKING_YUVA_4444_10, VL_PACKING_AUYV_4444_8, VL_PACKING_AUYV_4444_10, VL_PACKING_RGBA_8, VL_PACKING_RGBA_10, VL_PACKING_ABGR_8, VL_PACKING_ABGR_10, and VL_PACKING_AYU_AYV_10. See "Using VL_PACKING" in Chapter 2 for the specifications of each packing.
VL_OFFSET	*(0,0)*	xyVal	Specifies the upper left corner of a video region to be output. The coordinates are offsets of the upper left corner of the active video and take precedence over the size. Therefore, in order to accommodate the given offset, the size may be changed. A VLControlChanged event is generated to inform interested parties of any change in size.

Table B-6 (continued) Memory Drain Node Controls

Control	Default	Type	Use
VL_RATE	Dynamic; depends on timing and capture type	fractVal	Specifies the rate at which video units (fields or frames depending on capture type) are extracted from the ring buffer. The video unit is repeated, or black is output, to achieve the video output rate of 60 fields per second (NTSC) or 50 fields per second (PAL). The VGI1 memory source nodes can consume video units from system memory at any rate up to the video standard rate. For VL_CAPTURE_NONINTERLEAVED and VL_CAPTURE FIELDS, valid rates are as follows: NTSC: 1 through 60 units per second (must be multiple of fields per frame for noninterleaved) PAL: 1 through 50 units per second (must be multiple of fields per frame for noninterleaved) For VL_CAPTURE_INTERLEAVED, VL_CAPTURE_EVEN_FIELDS, or VL_CAPTURE_ODD_FIELDS, valid ranges are 1 through 30 units per second for NTSC and 1 through 25 units per second for PAL.
VL_SIZE	Dynamic; depends on timing and capture type	xyVal	Specifies the width (pixels) and height (lines) of the video data contained within each ring buffer entry which—along with VL_PACKING—determines the size in bytes of each ring buffer entry and thus the transfer size. The width must be a multiple of four pixels. The length must be a minimum of one line for field capture types, and two lines for frames. The specified size is constrained by both the maximum allowable (as dictated by the device timing and capture type) as well as the current offset position (VL_OFFSET). If the size is too large, it is reduced. The offset is not changed. It is recommended that VL_OFFSET be set before VL_SIZE.
VL_TIMING	Dynamic; from device node	intVal	Retrieves the current device-wide video timing value. See "VL_DEVICE" in this chapter for more details. Setting this control on any other node type has no effect.
VL_ZOOM	1.0	fractVal	Specifies the amount of scaling to be applied to the video before it is transferred to memory. The VGI1 memory drain nodes have no scaling ability. The only legal value is 1.0.

Table B-6 (continued) Memory Drain Node Controls

Control	Default	Type	Use
VL_MGV_BUFFER _QUANTUM	1	intVal	The granularity, or quantum, of data transfer required by the application. The video data is padded at the end so that the size of a field/frame is a multiple of VL_MGV_BUFFER_QUANTUM. This control is intended for applications that do I/O directly from the ring buffer, and may consequently require the frame or field size to be a multiple of the device block size. Direct I/O, for example, usually requires that 512 bytes of data be transferred at a time.
VL_MGV _DOMINANCE _FIELD	VL_MGV_DOMINANCE _F1	intVal	Sets the field dominance mode, determining the order in which the fields are read from memory. This control applies only to the frame-oriented capture types (VL_CAPTURE_INTERLEAVED and VL_CAPTURE_NONINTERLEAVED). See the discussion of VL_MGV_DOMINANCE_FIELD in Table B-5 earlier in this appendix for more details.
VL_MGV_DMA _ERROR_RESTART	VL_MGV_DMA _ERROR_RESTART_OFF	intVal	If enabled (VL_MGV_DMA_RESTART_ON), a video transfer continues when an error is encountered. Otherwise (VL_MGV_DMA_RESTART_OFF), the video transfer is aborted. This control covers three types of errors: The reference video timing is not clean, resulting in short/long lines, fields, or both. These errors are with respect to the programmed size and offset. The system GIO bus bandwidth was insufficient to transfer video from system memory at video rates. The video clock was interrupted.
VL_MGV_DMA _ROUND_TYPE	VL_MGV_DMA_RND _SMPLE	intVal	Specifies type of rounding algorithm to be used. The simple rounding method (VL_MGV_DMA_RND_SMPLE) rounds up if bit 1 is one or rounds down if bit 1 is zero. The randomized rounding method (VL_MGV_DMA_RND_RAND) makes the decision whether or not to round up based on comparing the two least-significant bits to a random sequence.

Table B-6 (continued) Memory Drain Node Controls

Control	Default	Type	Use
VL_MGV_DMA _RAND_ROUND _MODE	VL_MGV_DMA _RND_RAND _RPT	intVal	Determines whether or not the random sequence used for randomized rounding is repeated. If the sequence is to be repeated (VL_MGV_DMA_RND_RAND_RPT), then a shift register is seeded to a fixed value at the start of each odd field. Otherwise, the shift register free-wheels.
VL_MGV_DMA _VIN_ROUND	VL_MGV_DMA_RND_OFF	intVal	Enables (VL_MGV_DMA_RND_ON) or disables (VL_MGV_DMA_RND_OFF) rounding of 10-bit video data to 8 bits per component. Only the active area data is rounded.

VL_SCREEN

This discussion divides the VL_SCREEN nodes into their manifestations as source and drain.

VL_SCREEN Source

The OCTANE Digital Video option supports two screen source nodes: A and B.

The screen source nodes provide a means of using the graphics screen as a source of video data. Both pixel and alpha information can be extracted from the screen source area (although note that the alpha data can be sent only to the blender node’s alpha inputs). For screen nodes:

- type for both screen drain nodes is VL_SRC
- kind for both screen drain nodes is VL_SCREEN
- number is VL_MGV_NODE_NUMBER_SCREEN_A andVL_MGV_NODE_NUMBER_SCREEN_B, respectively
- ports are as follows:
  - VL_IMPACT_PORT_ALPHA_SRC_[A,B], single-link 8-bit CCIR pixel stream derived from a graphics window’s alpha contents
  - VL_IMPACT_PORT_PIXEL_SRC_[A,B], single-link 8-bit CCIR pixel stream derived from a graphics window’s pixel contents

Certain constraints apply to window positioning. If an application attempts to place the window in an illegal location, the node attempts to place the window at a valid location. The size of the window can also be changed. If either the position or size is changed, the application is notified by a VLValueChanged event. If the window cannot be placed anywhere on the screen, **vlSetControl()** returns VLBadValue.

Make sure your application meets these constraints:

- Windows A and B must not overlap vertically.
- The vertical distance between windows A and B must be greater than 12 pixels.

Table B-7 lists screen source node controls. For all these controls, access is GST.

Table B-7 Screen Source Node Controls

Control	Default	Type	Use
VL_FREEZE	FALSE	boolVal	When set to TRUE, this control freezes the contents of the screen drain. Updates to the graphics framebuffer continue to be displayed on the graphics display but are not reflected on the video output. If set to FALSE, live output resumes.
VL_OFFSET	(0, 0)	xyVal	Specifies the upper left corner of a subregion of the graphics area used to produce the video output. The offset is relative to VL_ORIGIN. See also VL_SIZE, which defines the size of the subregion.
VL_ORIGIN	(0, 0)	xyVal	Specifies the upper left corner of a frame-size graphics area used to produce the video. The origin is specified in X Window root-window coordinates. VL_OFFSET and VL_SIZE can be used to specify a subregion of this area.
VL_SIZE	CCIR 601 525: 720x486 CCIR 601 625: 768x576 NTSC: 640x486 PAL: 768x576	xyVal	Specifies the size of a subregion of the graphics area used to produce the video output. See also VL_OFFSET, which specifies the location of the subregion, and VL_ORIGIN, which maps a graphics window area to a frame. This control is applied before VL_ZOOM.

Table B-7 (continued) Screen Source Node Controls

Control	Default	Type	Use
VL_ZOOM	1.0	fractVal	Sets the amount of zoom that is applied to the graphics area before it is converted to video. Valid values are 1 and 1/2. Note: The 1/2 zoom value selects full-screen video output and should be used only when the OCTANE Digital Video device is operating on square-pixel timing (used with the OCTANE Compression option only).
VL_MGV_ DEINTERLAC E	TRUE	boolVal	Specifies how the video fields are generated from the frame-size graphics area with origin VL_ORIGIN. If this control is set to TRUE, a video field line is produced by averaging the corresponding graphics frame lines. If set to FALSE, the corresponding graphics line is selected depending on the field dominance and is output verbatim.

VL_SCREEN Drain

The OCTANE Digital Video option supports three screen drain nodes: A, B, and C.

The screen drain nodes provide a means of displaying video data in a graphics window. The OCTANE Digital Video device displays the video over a specified window, obscuring any graphics contents that may have been there. Note that the OCTANE Digital Video device does not place the video data into the framebuffer, but instead injects data directly into the raster. Consequently, a **glReadPixels()** (OpenGL) or **lrectwrite()** (IRIS GL) operation returns the contents that were drawn into the window, not the video data. For the screen drain:

- type for all three screen drain nodes is VL_DRN
- kind for all three screen drain nodes is VL_SCREEN
- number is VL_MGV_NODE_NUMBER_SCREEN_A, VL_MGV_NODE_NUMBER_SCREEN_B, and VL_MGV_NODE_NUMBER_SCREEN_C, respectively
- port is VL_IMPACT_PORT_PIXEL_DRN_[A,B,C], single-link 8-bit drain of CCIR-601 range video for display in a graphics window

To display live video using the graphics framebuffer, video frames should be captured using one of the VGI1 memory drain nodes, and then drawn using GL or OpenGL functions.

Screen drains B and C share the same physical framebuffer. When only drain B is used, the framebuffer is 24 bits deep. When drains B and C are used, or when only C is used, the framebuffer is split into two 12-bit logical framebuffers.

A 12- to 24-bit dithering is applied to produce the output of each window. While windows B and C can accept data from different sources, the following controls affect both nodes when applied to either:

- VL_SIZE
- VL_ZOOM
- VL_FREEZE

Window positioning has certain constraints. If an application attempts to place the window in an illegal location, the node attempts to place the window at a valid location. The size of the window can also be changed. If either the position or size is changed, the application is notified by a VLValueChanged event. If the window cannot be placed anywhere on the screen, **vlSetControl()** returns VLBadValue.

Make sure your application meets these constraints:

- Window A must not overlap B or C vertically, although it can overlap them horizontally.
- Windows B and C must not overlap horizontally, although they can overlap vertically.
- Window C must be to the right of window B by at least 45 pixels.
- The top of window C must be level with or below window B.
- The bottom of window B must be level with or higher than the bottom of window C.
- The vertical distance between window A and window B or C must be greater than 12 pixels.

Table B-8 lists screen drain node controls. For all these controls, access is GST, except VL_MGV_DEINTERLACE and VL_WINDOW, which are GS.

Table B-8 Screen Drain Node Controls

Control	Default	Type	Use
VL_FREEZE	FALSE	boolVal	If set to TRUE, this control freezes the contents of the screen drain. If set to FALSE, live video display resumes.
VL_OFFSET	(0, 0)	xyVal	Specifies the upper left corner of a subregion of the video frame to be displayed. See also VL_SIZE, which specifies the size of the subregion. When used with VL_ZOOM, VL_OFFSET is applied after VL_ZOOM.
VL_ORIGIN	(0, 0)	xyVal	Specifies the location on the screen where the video is displayed. The window coordinates are X-Server window coordinates.
VL_SIZE	CCIR 601 525: 720 x 486 CCIR 601 625: 768 x 576 NTSC: 640 x 486 PAL: 768 x 576	xyVal	Specifies the size of a subregion of the video frame to be displayed. See also VL_OFFSET, which specifies the location of the subregion. When used with VL_ZOOM, VL_SIZE is applied after VL_ZOOM.
VL_WINDOW	None	intVal	Specifies the window in which the video is displayed. The value set is the X Server window ID, as returned by **XtWindow()**, for example. The window ID cannot be changed while a screen drain is transferring.
VL_ZOOM	1.0	fractVal	Sets the amount of zoom applied to the video data before it is displayed. Valid values are 7/1, 6/1, 5/1, 4/1, 3/1, 2/1, 1/1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, and 1/8. Zoom is applied before offset (pan) and size.

Table B-8 (continued) Screen Drain Node Controls

Control	Default	Type	Use
VL_MGV_ALPHA _NOT_PIXEL	TRUE	boolVal	Specifies whether the screen drain should take pixel or alpha data from the source node. The preferred mechanism for specifying this information is to use **vlSetConnection()**; this function overrides the value of this control. If VL_MGV_ALPHA_NOT_PIXEL is set to TRUE, the screen drain takes input from the source's alpha port. Otherwise, it takes input from the source's pixel port. If the source node has no alpha port, then the pixel port is used.
VL_MGV _DEINTERLACE	FALSE	boolVal	Sets deinterlacing method. Converting video (which is interlaced) to graphics (which is progressive scan) requires deinterlacing the image: that is, replacing the missing lines with something. Two types of deinterlace methods are available: replacing the missing lines with black lines (the simplest method) or interpolating missing lines by simple filtering of adjacent lines. If set to TRUE, then the average of the adjacent lines is used to produce a full-brightness deinterlaced screen. If set to FALSE, then the lines contain black, producing a half-brightness window but with the same interlacing as video.

VL_TEXTURE

The texture drain node enables you to use video as a graphics texture source. The texture drain node can automatically generate mipmaps for use by the texture engine.

The texture node has two forms:

- single-link: VL_TEXTURE__NODE_NUMBER_TEXTURE
- dual-link: VL_TEXTURE_NODE_NUMBER_TEXTURE_DL

Chapter 9 provides information on using this node. For the texture drain node:

- type is VL_TEXTURE
- kind is VL_DRN
- number is VL_MGV_NODE_NUMBER_TEXTURE or VL_MGV_NODE_NUMBER_TEXTURE_DL

For the texture drain node VL_TEXTURE__NODE_NUMBER_TEXTURE, ports are VL_IMPACT_PORT_PIXEL_DRN_A and VL_IMPACT_PORT_PIXEL_DRN_B, which are single-link pixel drains feeding the graphics texture engine.

For the texture drain node VL_TEXTURE__NODE_NUMBER_TEXTURE_DL, ports are

- VL_IMPACT_PORT_DUALLINK_DRN_A: dual-link pixel drain feeding the graphics texture engine
- VL_IMPACT_PORT_PIXEL_DRN_A: single-link pixel drain feeding the graphics texture engine
- VL_IMPACT_PORT_ALPHA_DRN_A: single-link alpha drain feeding the graphics texture engine

Table B-9 summarizes controls for texture mapping. Access for all controls is GST.

Table B-9 Controls for Video Texture Mapping

Control	Default	Type	Use
VL_CAP_TYPE	VL_CAPTURE_FIELDS	intVal	Specifies type of field to capture. Valid capture types are VL_CAPTURE_NONINTERLEAVED, VL_CAPTURE_EVEN_FIELDS, VL_CAPTURE_ODD_FIELDS, and VL_CAPTURE_FIELDS. Note that texture nodes do not use VL_CAPTURE_INTERLEAVED. See “Using VL_CAP_TYPE and VL_RATE” in Chapter 2 for information on capture types.
VL_OFFSET	(0, 0)	xyVal	Specifies upper left corner of a subregion of the active video region used to generate the texture. Offset is relative to the upper left corner of the active video region. The subregion is defined by VL_OFFSET and VL_SIZE.
VL_PACKING	Single-link: VL_PACKING_RGB_8 Dual-link: VL_PACKING_RGBA_8	intVal	Specifies packing format for texture node. Valid values for single-link transfers are VL_PACKING_RGB_8 and VL_PACKING_RGBA_8. The only valid value for dual-link transfers is the default value
VL_RATE	NTSC: 60 fields/second PAL: 50 fields/second	fractVal	Specifies number of textures generated per second. This value is always equal to the maximum number of textures that can be generated per second based on the current value of VL_CAP_TYPE.

Table B-9 (continued) Controls for Video Texture Mapping

Control	Default	Type	Use
VL_SIZE	Dynamic	xyVal	Specifies size of subregion of active video region used to generate the texture. The location of the subregion is specified by VL_OFFSET.
VL_ZOOM	1.0	fractVal	Specifies scaling (for texture nodes, decimation only) applied to the video subregion used as a texture source. The video subregion is defined by VL_OFFSET and VL_SIZE. The scale factor along each axis of the subregion is a product of the zoom value and the aspect value for that axis.
VL_MGV_DOMINANCE _FIELD	VL_MGV _DOMINANCE_F1	intVal	Specifies field dominance when VL_CAP_TYPE is VL_CAPTURE_NONINTERLEAVED. Valid values are VL_MGV_DOMINANCE_F1 and VL_MGV_DOMINANCE_F2.
VL_MGV_HASPECT	1.0	fractVal	Specifies scaling applied to the video subregion along the horizontal axis. The overall scale factor horizontally is the product of this value and the VL_ZOOM value.
VL_MGV_VASPECT	1.0	fractVal	Specifies scaling applied to the video subregion along the vertical axis. The overall scale factor vertically is the product of this value and the VL_ZOOM value.
VL_MGV_TEXTURE _ROUND_MODE	VL_MGV_TEXTURE _ROUND_8BIT	intVal	Specifies the type of rounding to use to convert from 10- to 8-bit input. Valid values are VL_MGV_TEXTURE_ROUND_8BIT, VL_MGV_TEXTURE_ROUND_RNG, and VL_MGV_TEXTURE_ROUND_RNGFRM.
VL_MGV_TEXTURE _MIPMAP_MODE	VL_MGV_TEXTURE _MIPMAP_OFF	intVal	Specifies whether mipmap mode is enabled or not. Valid values are VL_MGV_TEXTURE_MIPMAP_OFF and VL_MGV_TEXTURE_MIPMAP_ON.

Table B-9 (continued) Controls for Video Texture Mapping

Control	Default	Type	Use
VL_MGV_TEXTURE_INPUT_LINK	VL_MGV_TEXTURE_INPUT_LINK_A	intVal	Specifies which input link sends pixel data to the texture engine. This command is available only in single-link mode when autoswapping is off. Valid values are VL_MGV_TEXTURE_INPUT_LINK_A and VL_MGV_TEXTURE_INPUT_LINK_B.
VL_MGV_TEXTURE_AUTOSWAP	VL_MGV_TEXTURE_AUTOSWAP_OFF	intVal	Specifies whether to swap the input automatically to the texture engine. When autoswapping is enabled, the input to the texture engine alternates between the two texture drain ports after each capture into texture memory. If VL_CAP_TYPE is set to VL_CAPTURE_NONINTERLEAVED, swapping occurs after frame transfers. This control is available only in single-link mode. Valid values are VL_MGV_TEXTURE_AUTOSWAP_OFF and VL_MGV_TEXTURE_AUTOSWAP_ON.

VL_VIDEO

This discussion divides the VL_VIDEO node into its manifestations as source and drain.

VL_VIDEO Source

The OCTANE Digital Video option supports three digital video source nodes: 1, 2, and dual-link.

The video source nodes correspond to two video input connectors available on the OCTANE Digital Video device. These connectors can be used separately to feed CCIR-601 video to the OCTANE Digital Video option, or as a dual link to supply RP-175 RGB or YUV 4:4:4:4 or 4:2:2:4 video. When used in dual-link mode, the connector labelled 1 is used for pixel input while the connector labelled 2 is used for alpha input.

The uses of the connectors in single- or dual-link modes are mutually exclusive. When either of the single-linked nodes are in use, the dual-linked node is unavailable. Similarly, if the dual-link node is in use, both single-linked nodes are considered to be in use. Mutual exclusion takes place when paths are set up with stream usage VL_SHARE or VL_LOCK. Mutual exclusion conditions are not applied to paths with stream usage VL_READ_ONLY. For the video source:

- type for all three screen drain nodes is VL_SRC
- kind for all three screen drain nodes is VL_VIDEO
- number is VL_MGV_NODE_NUMBER_VIDEO_1 VL_MGV_NODE_NUMBER_VIDEO_2, and VL_MGV_NODE_NUMBER_VIDEO_DL, respectively
- ports are as follows:
  - video source nodes 1 and 2: VL_IMPACT_PORT_PIXEL_SRC_A. single-link serial digital video input
  - dual-link video source node: VL_IMPACT_PORT_DUALLINK_SRC_A - RP-175 style RGB or YUV dual-link serial digital video

Table B-10 lists video source node controls. For all these controls, access is GST, except VL_FORMAT, which is GS.

Table B-10 Video Source Node Controls

Control	Default	Type	Use
VL_FREEZE	FALSE	boolVal	Freezes the input video stream. Because the OCTANE Digital Video device does not support frozen inputs, this control can be set only to FALSE.
VL_FORMAT	VL_FORMAT_DIGITAL_ COMPONENT_SERIAL (single-link) VL_FORMAT_DIGITAL_ COMPONENT_DUAL_ SERIAL (dual-link)	intVal	Specifies the format of the incoming video. Valid values for the single-link nodes are VL_FORMAT_DIGITAL_COMPONENT_SERIAL and VL_FORMAT_RAW_DATA. Valid values for the dual-link node are VL_FORMAT_DIGITAL_COMPONENT_DUAL_SERIAL and VL_FORMAT_DIGITAL_COMPONENT_RGB_SERIAL. (See "Using VL_FORMAT" in Chapter 2 for format explanations.)
VL_OFFSET	(0, 0)	xyVal	Pans within the video. The OCTANE Digital Video source nodes support an offset of (0, 0) only.

Table B-10 (continued) Video Source Node Controls

Control	Default	Type	Use
VL_SIZE	Dynamic	xyVal	Reports the width and height of the active video region. The values are fixed for each timing mode: CCIR 525: 720 x 486 CCIR 625: 720 x 576 NTSC square pixel: 640 x 486 PAL square pixel: 768 x 576 Square pixel modes are used with the OCTANE Compression option only. Timing is specified through the VL_TIMING control on the device node.
VL_MGV_INPUT_8BIT	FALSE	boolVal	Enables 10-bit to 8-bit truncating when 10-bit video input is supplied. Internally, the OCTANE Digital Video option treats all video streams as 10-bit. As a result, when this control is set to TRUE, the lower two bits are forced to zero. If set to FALSE, the input stream passes unmodified. If the input video contains 8-bit data, then it is left-shifted two bits to produce a 10-bit value. The lower two bits contain zeros.

VL_VIDEO Drain

The OCTANE Digital Video option supports three digital video drain nodes: 0, 1, and 2.

The video drain nodes correspond to two video output connectors available on the OCTANE Digital Video device. These connectors can be used separately to output CCIR-601 video from the OCTANE Digital Video option, or as a dual link to output RP-175 RGB or YUV 4:4:4:4 or 4:2:2:4 video. When used in dual-link mode, the connector labelled 1 is used for pixel output while the connector labelled 2 is used for alpha output.

The uses of the connectors in single or dual-link modes are mutually exclusive. When either of the single-linked nodes are in use, the dual-linked node is unavailable. Similarly, if the dual-link node is in use, both single-linked nodes are considered to be in use. Mutual exclusion takes place when paths are set up with stream usage VL_SHARE or VL_LOCK. Mutual exclusion conditions are not applied to paths with stream usage VL_READ_ONLY. For the video drain:

- type for all three video drain nodes: VL_DRN
- kind for all three video drain nodes: VL_VIDEO
- number: VL_MGV_NODE_NUMBER_VIDEO_1 VL_MGV_NODE_NUMBER_VIDEO_2, and VL_MGV_NODE_NUMBER_VIDEO_DL, respectively
- ports:
  - video source nodes 1 and 2: VL_IMPACT_PORT_PIXEL_SRC_A: single-link serial digital video input
  - dual-link video source node: VL_IMPACT_PORT_DUALLINK_SRC_A: RP-175 style RGB or YUV dual-link serial digital video

Table B-11 lists video drain node controls. For all these controls, access is GST, except VL_FORMAT, which is GS.

Table B-11 Video Drain Node Controls

Control	Default	Type	Use
VL_FREEZE	FALSE	boolVal	If set to TRUE, the output of the drain node is frozen. For dual-link, both outputs are frozen simultaneously. Use the VL_MGV_OUTPUT_DL_SELECT_FREEZE control to freeze specific links of the dual-link nodes. Note that VL_MGV_OUTPUT_FSYNC must be set to TRUE in order for the video to freeze.
VL_FORMAT	VL_FORMAT_DIGITAL_ COMPONENT_SERIAL (single-link) VL_FORMAT_DIGITAL_ COMPONENT_DUAL_ SERIAL (dual-link)	intVal	Specifies the format of the incoming video. Valid values for the single-link nodes are VL_FORMAT_DIGITAL_COMPONENT_SERIAL and VL_FORMAT_RAW_DATA (arbitrary 8-bit data). Valid values for the dual-link node are VL_FORMAT_DIGITAL_COMPONENT_DUAL_SERIAL and VL_FORMAT_DIGITAL_COMPONENT_RGB_SERIAL.

Table B-11 (continued) Video Drain Node Controls

Control	Default	Type	Use
VL_OFFSET	(0, 0)	xyVal	Sets offset; the OCTANE Digital Video drain nodes support an offset of (0, 0) only.
VL_SIZE	Dynamic	xyVal	Reports the width and height of the active video region. The values are fixed for each timing mode: CCIR 525: 720 x 486 CCIR 625: 72 0x 576 NTSC square pixel: 640 x 486 PAL square pixel: 768 x 576 Square pixel modes are used with the OCTANE Compression option only. Timing is specified through the VL_TIMING control on the device node.
VL_MGV_ALPHA _NOT_PIXEL	FALSE	boolVal	If the node supplying the video drain node has both pixel and alpha outputs, this control selects whether the alpha (TRUE) or pixel (FALSE) channel is selected. This control is provided only for compatibility with Galileo Video applications. It is recommended that the application use **vlSetConnection()** to specify the output port.
VL_MGV_OUTPUT _BLANK	Persistent	boolVal	If this control is set to TRUE, the video output is blanked; that is, video black is output on the serial digital port. If set to FALSE, live video is displayed. VL_MGV_OUTPUT_FSYNC must be set to TRUE for this control to have any effect.
VL_MGV_OUTPUT _CHROMA	Persistent	boolVal	If set to TRUE, the chroma portion of a video stream is passed through. If set to FALSE, chroma is blanked. VL_MGV_OUTPUT_FSYNC must be set to TRUE for this control to have any effect.
VL_MGV_OUTPUT _FSYNC	Persistent	boolVal	If set to TRUE, this control enables the output synchronization hardware. The output synchronizer must be enabled for the VL_FREEZE, VL_MGV_OUTPUT_BLANK, VL_MGV_OUTPUT_HPHASE, or VL_MGV_OUTPUT_CHROMA controls to have any effect.

Table B-11 (continued) Video Drain Node Controls

Control	Default	Type	Use
VL_MGV_OUTPUT _HPHASE	0xC00	intVal	Specifies the horizontal phase of the video output with respect to the video input. It is a 12-bit unsigned integer that increments in steps of the pixel clock (typically 74 nsec). The output occurs later in time as the value of this control increases. This control has a range of 1 to 0xFFF, which can advance the output by slightly more than three lines or delay the output by slightly more than one line. The default value 0xC00 makes the output match the timing of the video input. The value 0 is illegal. VL_MGV_OUTPUT_FSYNC must be set to TRUE for this control to have any effect.
VL_MGV_DL _SELECT_BLANK	VL_MGV_DL_SELECT _ALL	intVal	Selects the channels (pixel, alpha, both) on the dual-link node to blank. Valid values are VL_MGV_DL_SELECT ALPHAL (alpha channel only), VL_MGV_DL_SELECT_PIXEL (pixel channel only), and VL_MGV_DL_SELECT_AL (pixel and alpha channels).
VL_MGV_OUTPUT _DL_SELECT _CHROMA	VL_MGV_DL_SELECT _ALL	intVal	Selects the channels (pixel, alpha, both) on the dual-link node on which chroma should be blanked. Valid values are the same as for VL_MGV_OUTPUT_DL_SELECT_BLANK.
VL_MGV_OUTPUT _DL_SELECT _FREEZE	VL_MGV_DL_SELECT _ALL	intVal	Selects the channels (pixel, alpha, both) on the dual-link node to freeze. If both channels are selected, the freeze is performed atomically on both channels. Valid values are the same as for VL_MGV_OUTPUT_DL_SELECT_BLANK.
VL_MGV_OUTPUT _DL_SELECT _FSYNC	VL_MGV_DL_SELECT _ALL	intVal	Selects the output on the dual-link node synchronizers to enable or disable. Valid values are the same as for VL_MGV_OUTPUT_DL_SELECT_BLANK.

Appendix C

OCTANE Digital Video Color-Space Conversions

The OCTANE Digital Video option supports three native color spaces—RGB, YUV, and CCIR. The choice of color space is determined by the external equipment for video I/O connections, by the system for connections to the graphics subsystem, and by application software for transfers to and from system memory. Application software can avoid all color-space conversions during video I/O. The OCTANE Digital Video option can translate between YUV and RGB with high accuracy in real time.

Understanding the capabilities of the OCTANE Digital Video option to perform color-space conversions and the results of these conversions allows developers and end users to maximize the quality of their output. This appendix explains

- "OCTANE Digital Video Color Spaces"
- "Mathematical Operations Performed During Conversions"
- "Implications of Color-Space Conversions"
- "Example Color Conversions"

OCTANE Digital Video Color Spaces

The OCTANE Digital Video option uses a minimum of ten bits of precision for each color component at all steps of its internal pipeline. Representations for the three native internal color representations are explained separately in this section.

RGB

RGB is the color space used by the graphics subsystem; screen sources and drains and some memory transfers use this color space. RGB has the most accurate representation of visible colors, because all possible combinations are valid. This color space does not support superblack or other nonvisible color values. Each component is represented by a 10-bit value between 0 and 1023. Black has the value [0,0,0], and white is [1023,1023,1023].

When converting to RGB, each resulting RGB component is clamped to the range [0..1023]. It is possible to overflow the clamping mechanism when dramatically illegal colors are input. Overflows occur only when the resulting red, green, or blue value is greater than 2047 or less than -2048.

Note: Do not use 4:2:2 coding with RGB data.

YUV

The YUV color space is obtained from RGB by the matrix transformation in equation 1.

Equation 1
$$\begin{bmatrix} 0.500 & -0.419 & -0.081 \\ 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 512 \\ 0 \\ 512 \end{bmatrix} = \begin{bmatrix} V \\ Y \\ U \end{bmatrix}$$

The V, Y, and U values range from [0..1023]. Black has the VYU value [512,0,512]. White has the value [512,1023,512].

This color space is used by the Betacam, M-II, and YUV formats. With proper filtering, 4:2:2 coding can be used.

CCIR

The CCIR color space is obtained from RGB by the matrix transformation in equation 2.

Equation 2

$$\begin{bmatrix} 0.500 & -0.419 & -0.081 \\ 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix} \times \begin{bmatrix} \frac{896}{1023} \\ \frac{876}{1023} \\ \frac{896}{1023} \end{bmatrix} + \begin{bmatrix} 512 \\ 64 \\ 512 \end{bmatrix} = \begin{bmatrix} Cr \\ Y \\ Cb \end{bmatrix}$$

The Cr, Y, and Cb values are clamped to the range [4..1019]. Black has the CrYCb value [512,64,512]. White has the value [512,940,512].

This color space is used by the component digital formats. With proper filtering, 4:2:2 coding can be used.

Mathematical Operations Performed During Conversions

The OCTANE Digital Video option can process and store each color space explained in the previous section. For best precision, the input color space should be maintained through the processing path. For example, an application that implements DDR functionality could choose to store data in the native representation of the input signal: Betacam data could be stored as YUV, input from an RGB camera as RGB, and data from a D1 deck as CCIR. If the application works in this way, no conversions are performed and the data is passed directly through the system. In particular, CCIR601 data coming from a D1 deck is bit-accurate in this case.

However, it might not be desirable for the application to work this way. If that is the case, the application can use all of the conversion, decimation and interpolation capabilities of the OCTANE Digital Video option to perform real-time color space and 4:2:2 ⇔ 4:4:4 conversions.

Conversions are performed only when absolutely required. Each incoming stream can be converted from its current color space to any other color space. Conversions can also be performed when going to graphics and digital video outputs.

The output color space controls conversions. For example, if you blend a CCIR stream from a digital video input with an RGB stream from graphics and send the result to the digital video output, the RGB signal is converted to CCIR before the blend occurs. The CCIR stream is not converted. If you sent the same blend to a Betacam output, both streams are converted to YUV before the blend.

Implications of Color-Space Conversions

The two major concerns when performing conversions from one color space to another are *precision* and *range*.

Precision of Color Conversions Done by the OCTANE Digital Video Option

The OCTANE Digital Video option stores colors at all steps in its pipeline with a minimum of 10 bits of precision. When performing color-space conversions, the data is converted to 12-bit signed values before it is passed to the matrix multipliers. The matrix multipliers have 15-bit coefficients and 26-bit accumulators. The most significant 16 bits of the matrix-multiplication result are passed on to additional hardware, which applies any needed offsets and then clamps to the proper range.

Silicon Graphics, Inc., has verified both through simulation and hardware testing that the maximal error for two conversions (RGB to CCIR to RGB) is four units out of 1024. The matrix coefficients have been biased to round slightly high rather than slightly low to avoid the type of problems that can otherwise easily occur in the blue component.

Conversions between RGB and YUV are more accurate (a maximum error of 3 in 1024 after two conversions), since data is not as compressed in the YUV representation.

Range Issues For Color Conversions Done by Any Means

Different color spaces allocate the available bits of precision in different ways. The RGB space is designed to maximize the accuracy of color representations. The YUV and CCIR color spaces are designed to strongly uncouple chrominance and luminance information.

Since RGB represents visible colors, it is contained inside the YUV and CCIR spaces. The CCIR color space also has a slight amount of additional headroom that was intended to prevent aliasing artifacts when Finite Impulse Response filtering operations are performed on the digital data.

Whenever a conversion operation is performed between CCIR and RGB or between CCIR and YUV, the colors that are not representable in the destination color space must be somehow mapped into colors that are representable. The usual way to do this is to clamp each component to the available range in the destination color space. Other methods, such as projecting towards the center of the representable space, might produce results that appear to be better in some cases, but are not feasible to implement in hardware.

When converting from CCIR to YUV, the axes of the two spaces are parallel, so the result of this clamping operation is very predictable. Superblack and superwhite are clipped to black and white, respectively, and oversaturated colors might also be clipped.

When converting from RGB to YUV or CCIR, clamping never occurs, because all RGB colors are representable in those color spaces.

When converting from CCIR or YUV to RGB, the results of clamping are much less intuitive, because these conversions involve rotation and scaling operations, with the result that the component axes in one color space do not align with those in the other.

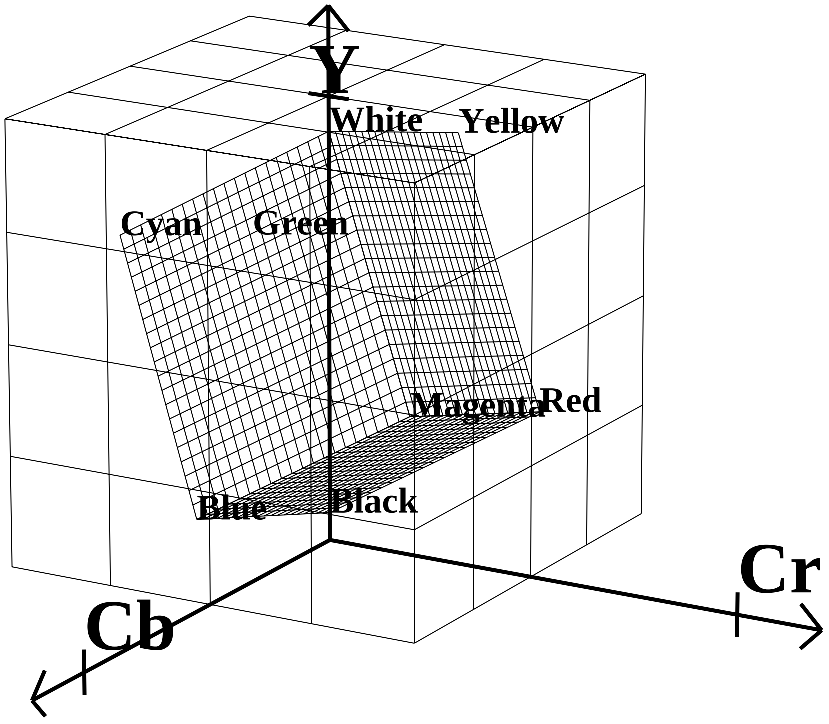


Figure C-1 RGB Cube in CCIR Space

Figure C-1 shows the RGB color cube inside the CCIR color space. The volume contained within the outer (CCIR) cube, but outside the inner (RGB) cube, represents “illegal” colors that cannot be displayed.

As shown in the figure, the CCIR color space allocates almost three quarters of its available bit combinations to illegal colors. When any of these color values are converted to RGB, the result is clamped to the edge of the RGB cube. Since the inner cube contains the displayable colors, this clamping operation has no impact on them.

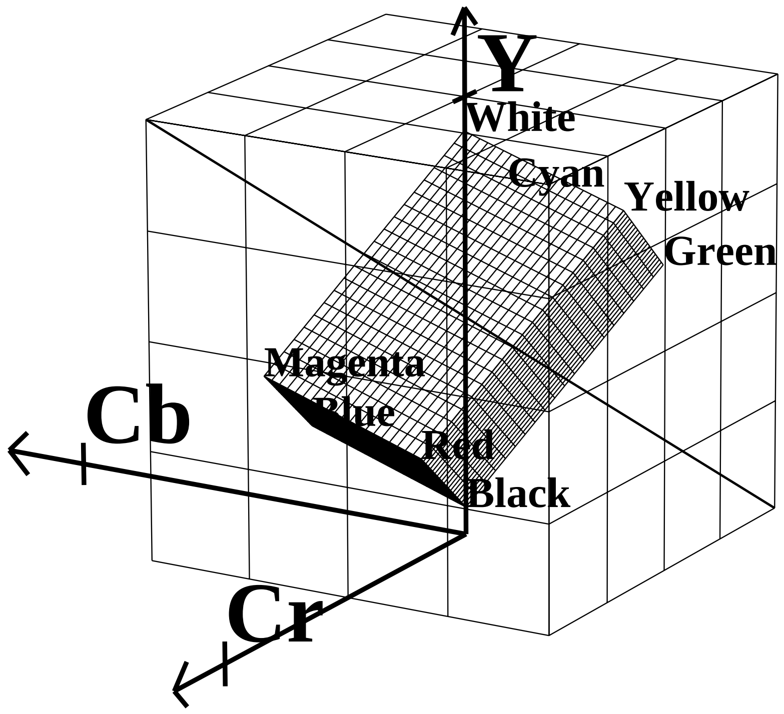


Figure C-2 Color Cube With Luminance/Chrominance Ramp Vector

If CCIR is converted to RGB and back to CCIR using certain types of test signals, the output can appear to be vastly wrong. A common and extreme version of this is the signal that simultaneously ramps Cr, Y, and Cb from the minimum to maximum possible values.

In Figure C-2, the heavy diagonal line passing through the figure is the set of colors in the luma/chroma ramp test signal. As shown in the figure, a large portion of this pattern is outside the RGB cube. In fact, over two thirds of this pattern is outside the displayable range.

Example Color Conversions

This section includes example graphs that display the results of converting from CCIR to RGB and back. They show the same type of result you would see if you brought a digital signal into the OCTANE Digital Video option, passed it through a screen or memory node using RGB format, and sent it back out to the digital output.

These effects do not occur if you simply pass digital data through the OCTANE Digital Video board using the CCIR format. In these cases, the output matches the input on a bit-by-bit basis.

Note: These examples show conversion from CCIR to full-range RGB, without use of the constant-hue algorithm.

Example 1: 100% Color Bars

This example, like the other two in this section, consists of three graphs. Each graph displays the input CCIR pattern, intermediate RGB pattern, and output CCIR pattern for a given color component. Figure C-3 shows the red and Cr components, Figure C-4 the green and Y components, and Figure C-5 the blue and Cb components. In this example and the others, if the input and output CCIR values are identical, only two lines are shown.

In this example, conversion to RGB and back has no effect on the image. The 100% amplitude color bar signal lies within the visible range and therefore is perfectly represented in RGB.

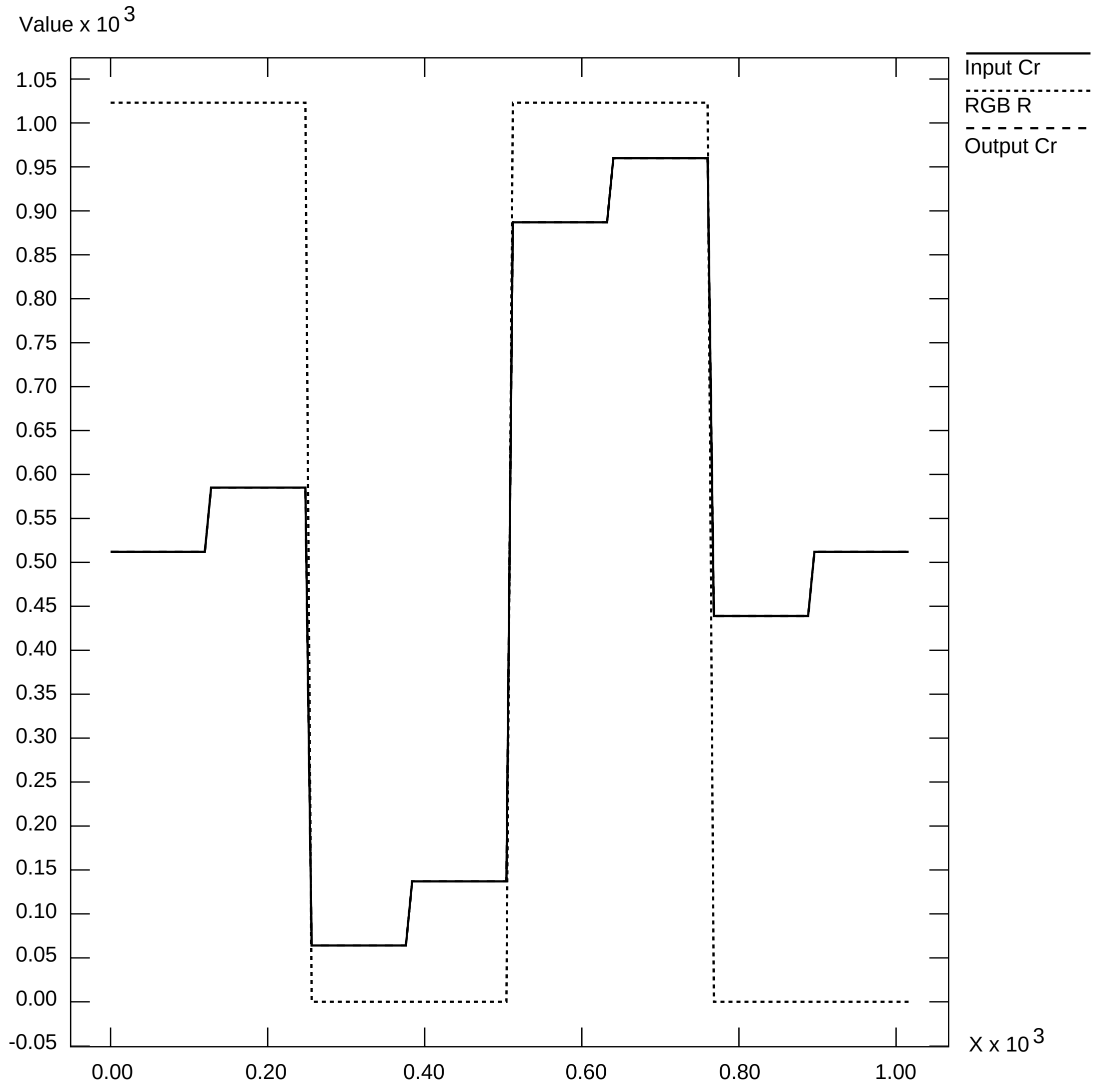


Figure C-3 100% Color Bars: Cr/R

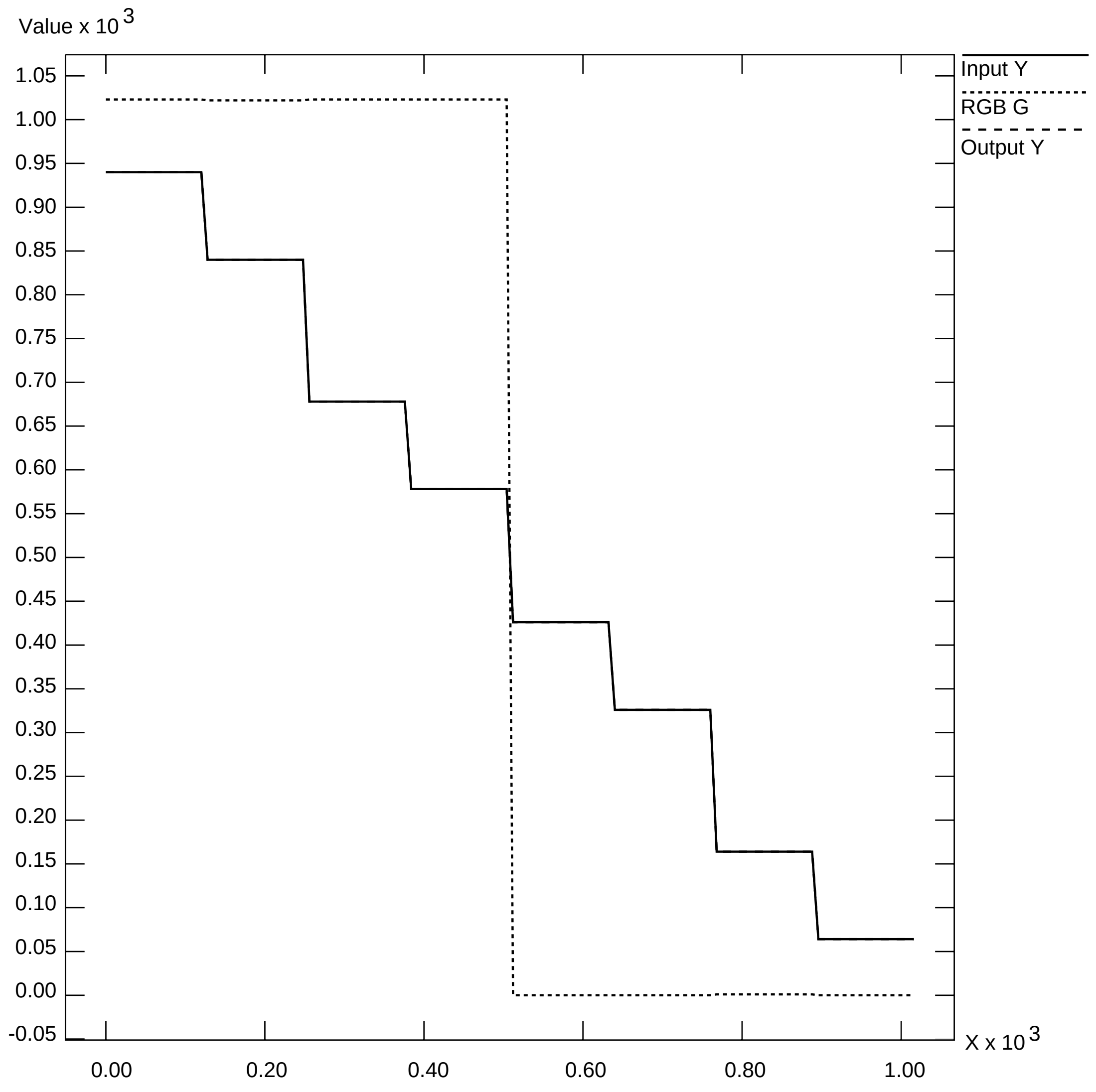


Figure C-4 100% Color Bars: Y/G

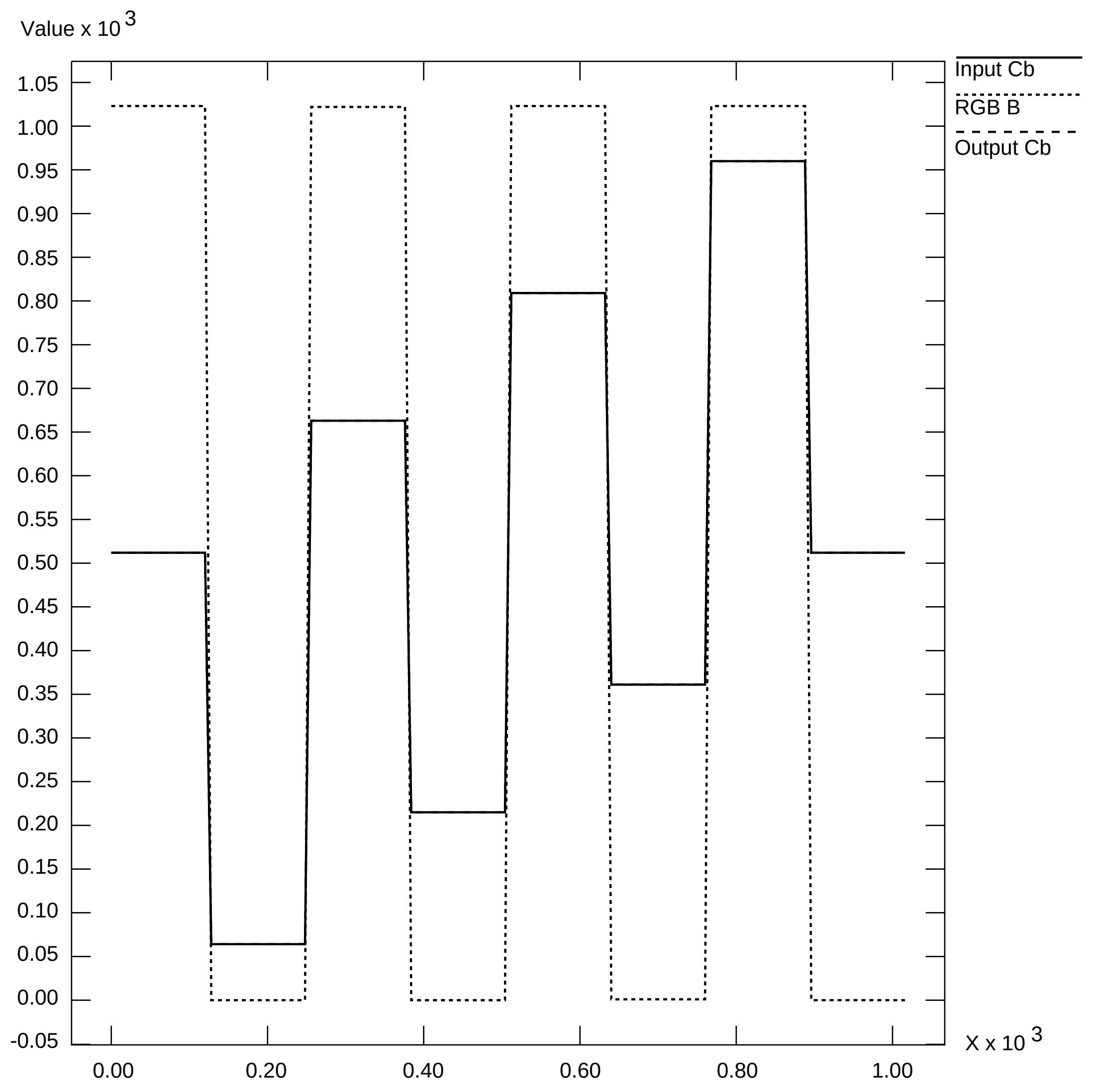


Figure C-5 100% Color Bars: Cb/B

Example 2: Luminance Ramp

In this example, the conversion to RGB and back affects only the superblack and superwhite regions. All luminance values that are blacker than black are clamped to black; all values whiter than white are clamped to white.

In the RGB color space, each component ramps from 0 to 1023 as the input luminance ramps from 64 (black) to 940 (white). This test pattern lies along the Y axis of the color cubes.

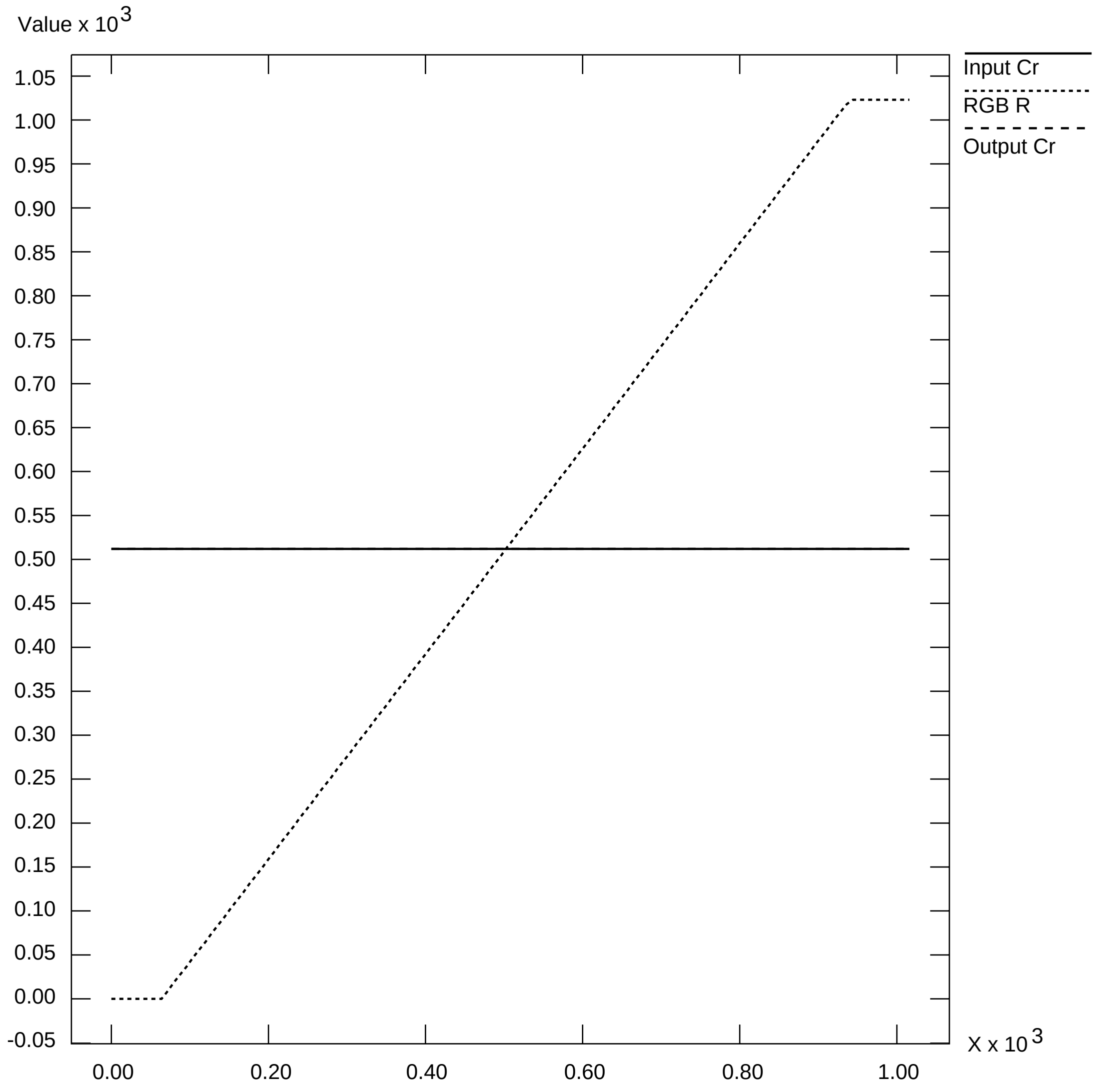


Figure C-6 Luminance Ramp: Cr/R

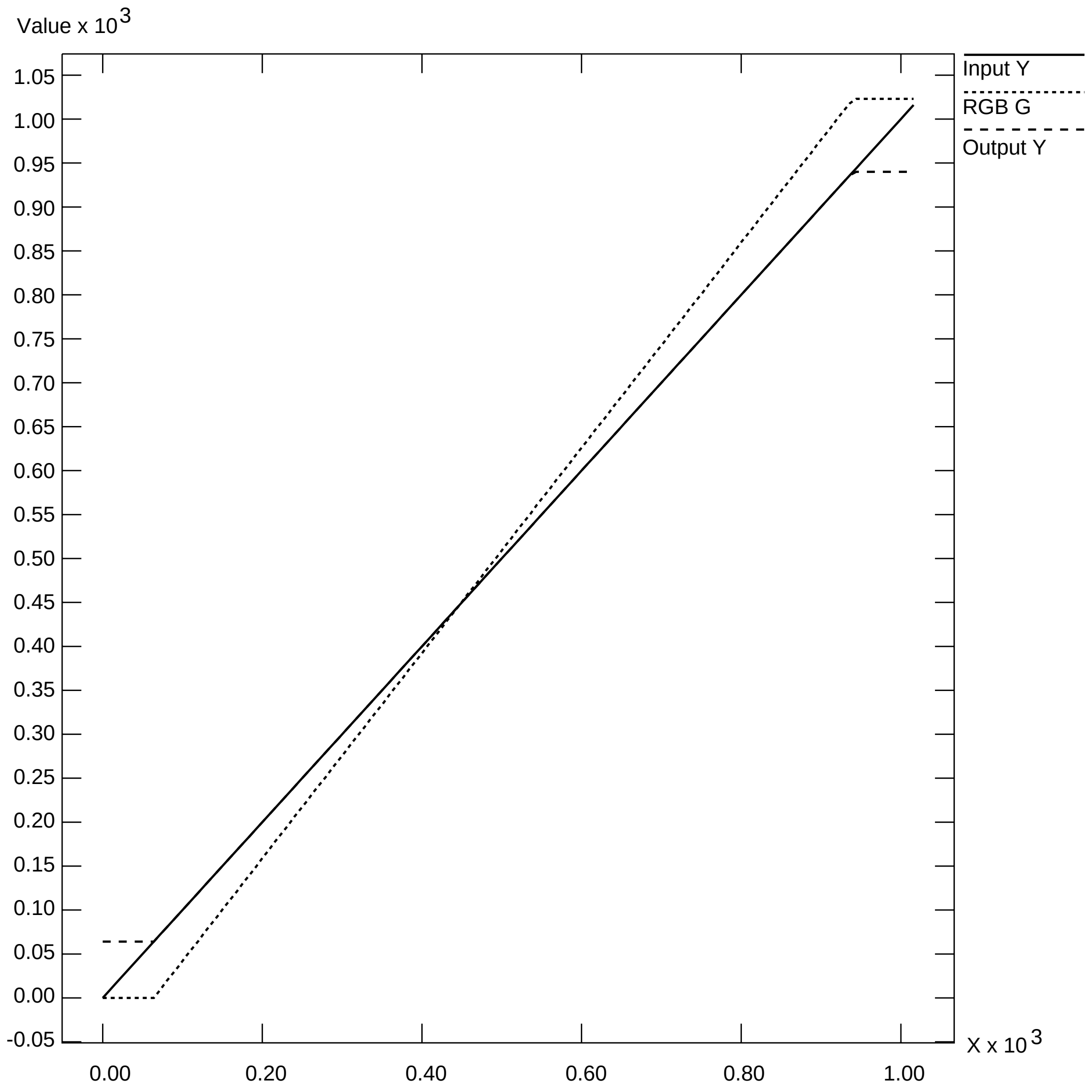


Figure C-7 Luminance Ramp: Y/G

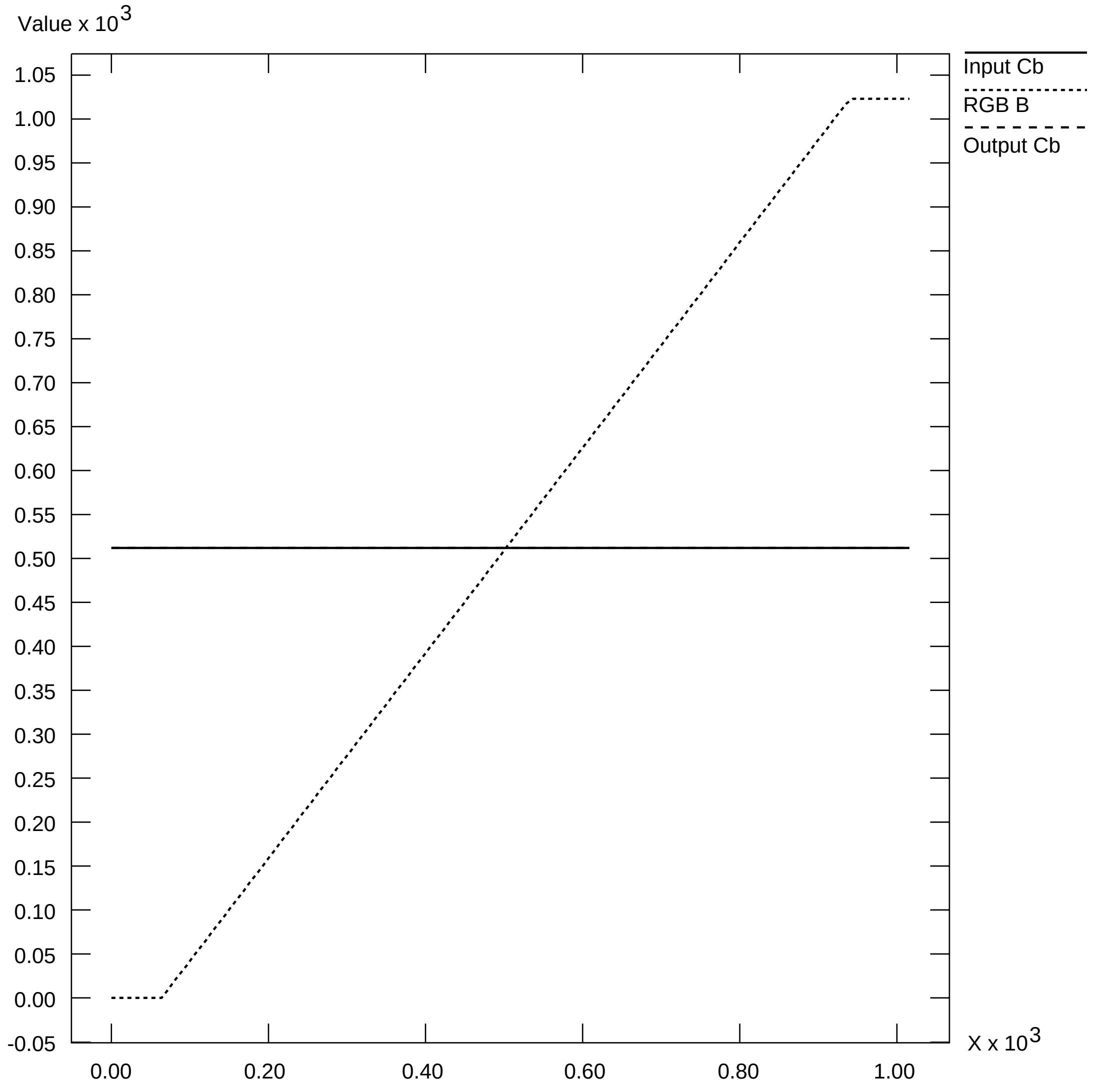


Figure C-8 Luminance Ramp: Cb/B

Example 3: Simultaneous Chroma/Luma Ramp

This example is the most extreme of the three, and shows how surprising the results of color conversions can be when arbitrary synthetic CCIR inputs are used.

Each CCIR input signal ramps from 0 to 1023 simultaneously. As mentioned in the first example, over two thirds of this pattern lies outside the legal range. The portion within the legal range is represented exactly, but the region outside is clamped to the RGB cube surface.

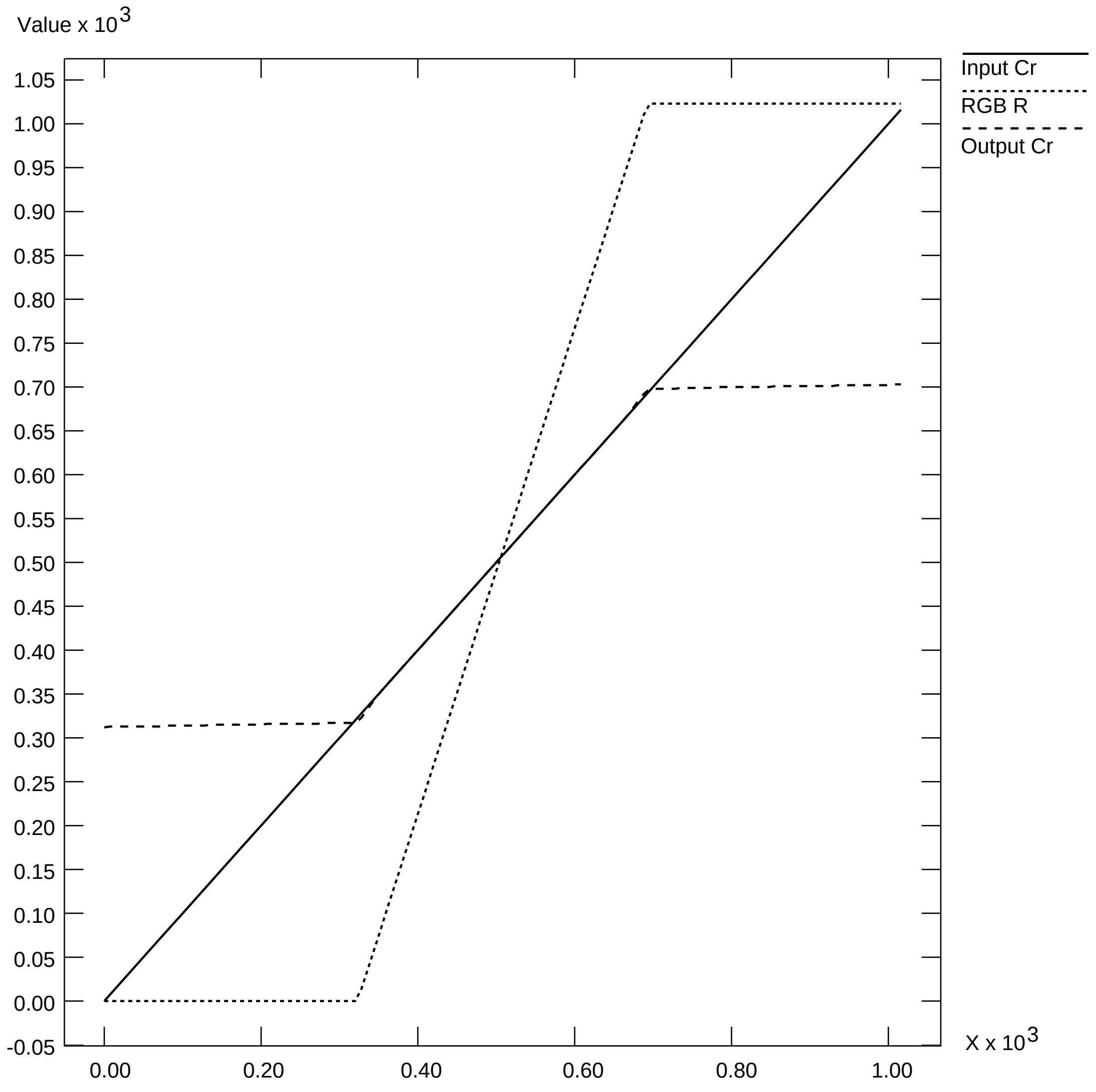


Figure C-9 Chroma/Luma Ramp: Cr/R

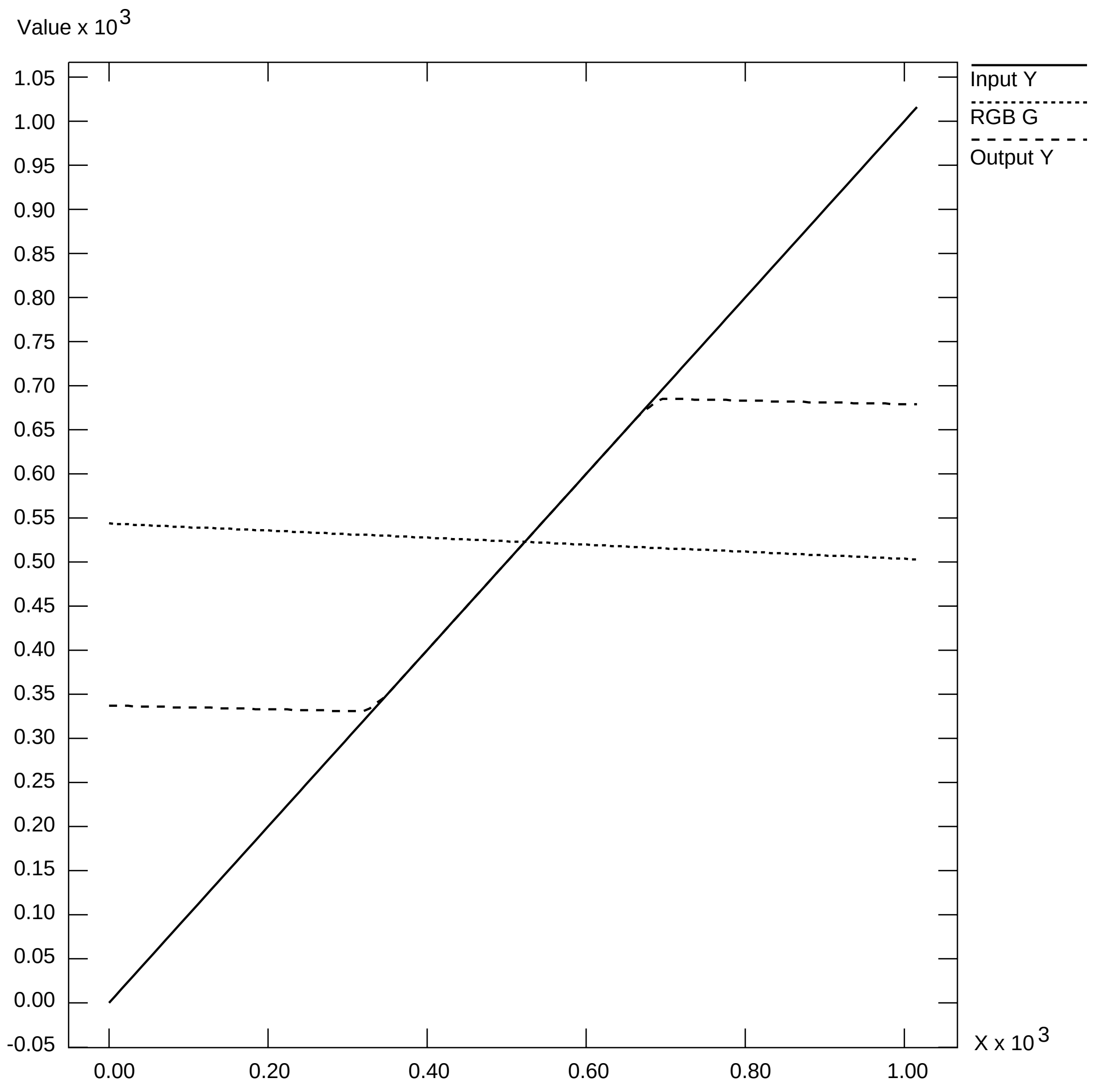


Figure C-10 Chroma/Luma Ramp: Y/G

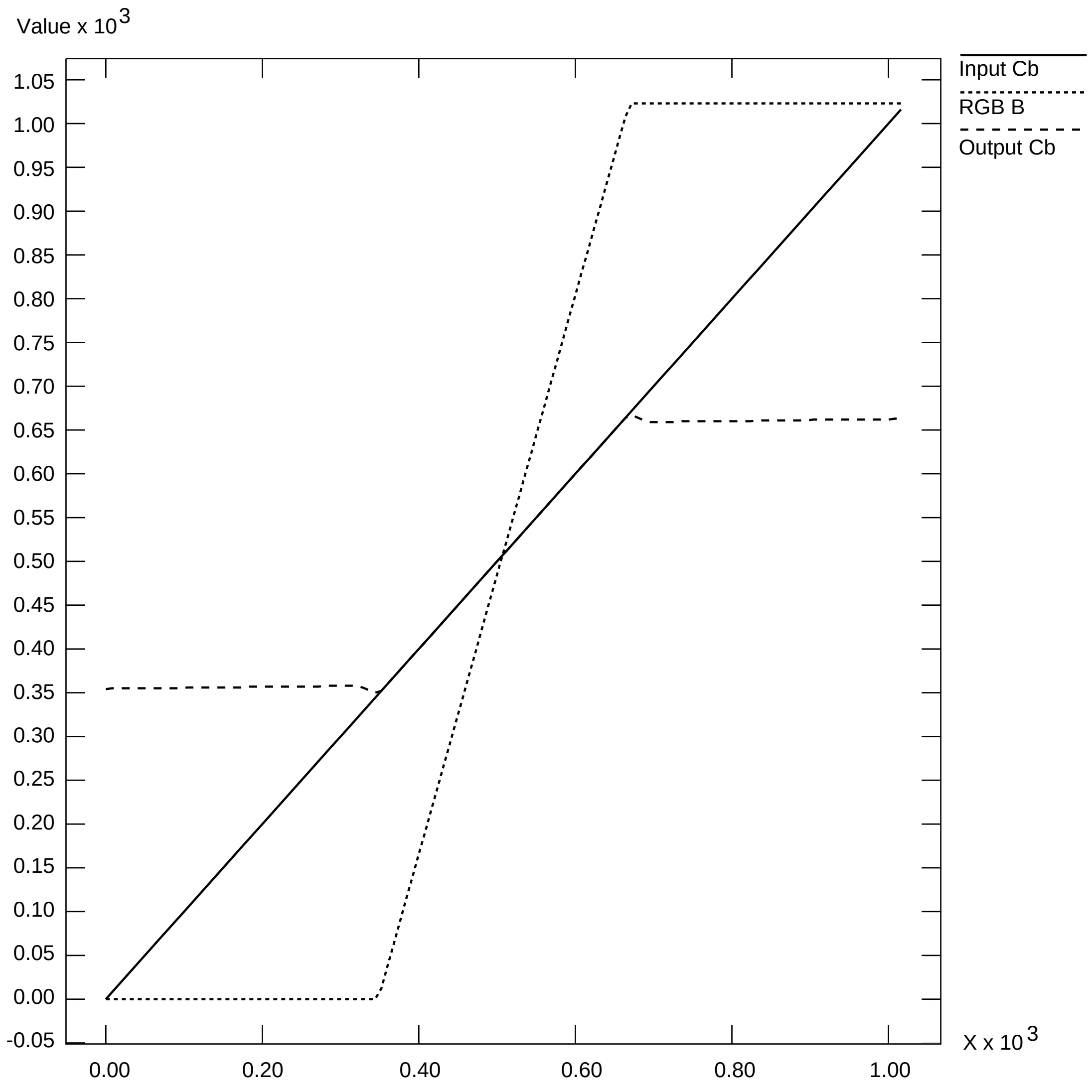


Figure C-11 Chroma/Luma Ramp: Cb/B

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® 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 0.5 MHz. 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_B 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 3.58 MHz (NTSC) or 4.43 MHz (PAL) subcarrier signal for color in television. SECAM uses two frequency-modulated color subcarriers transmitted on alternate horizontal lines; SC_R is 4.406 MHz and SC_B is 4.250 MHz.

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 (3.58 MHz for NTSC, 4.43 MHz 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.

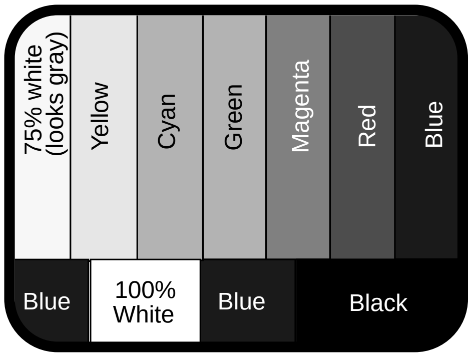


Figure Gl-1 SMPTE Color Bars (75%)

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 3.58 MHz 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 3.58 MHz color subcarrier transmitted on the back porch of every horizontal pulse; the hue of the color sync phase is yellow-green.

Figure Gl-2 diagrams the relationship of the color burst and the chrominance signal. See also *color subcarrier* and *video waveform*.

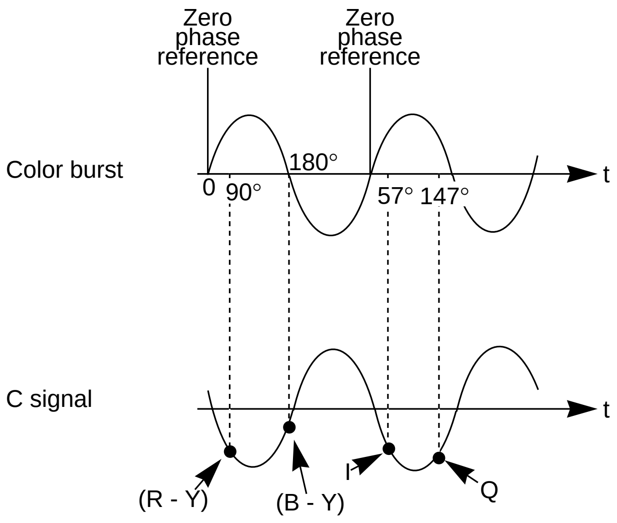


Figure Gl-2 Color Burst and Chrominance Signal

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 color component encoding format defined by three color components, such as R, G, and B or Y, U, and V.

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 3.58 MHz in NTSC and 4.43 MHz 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*.

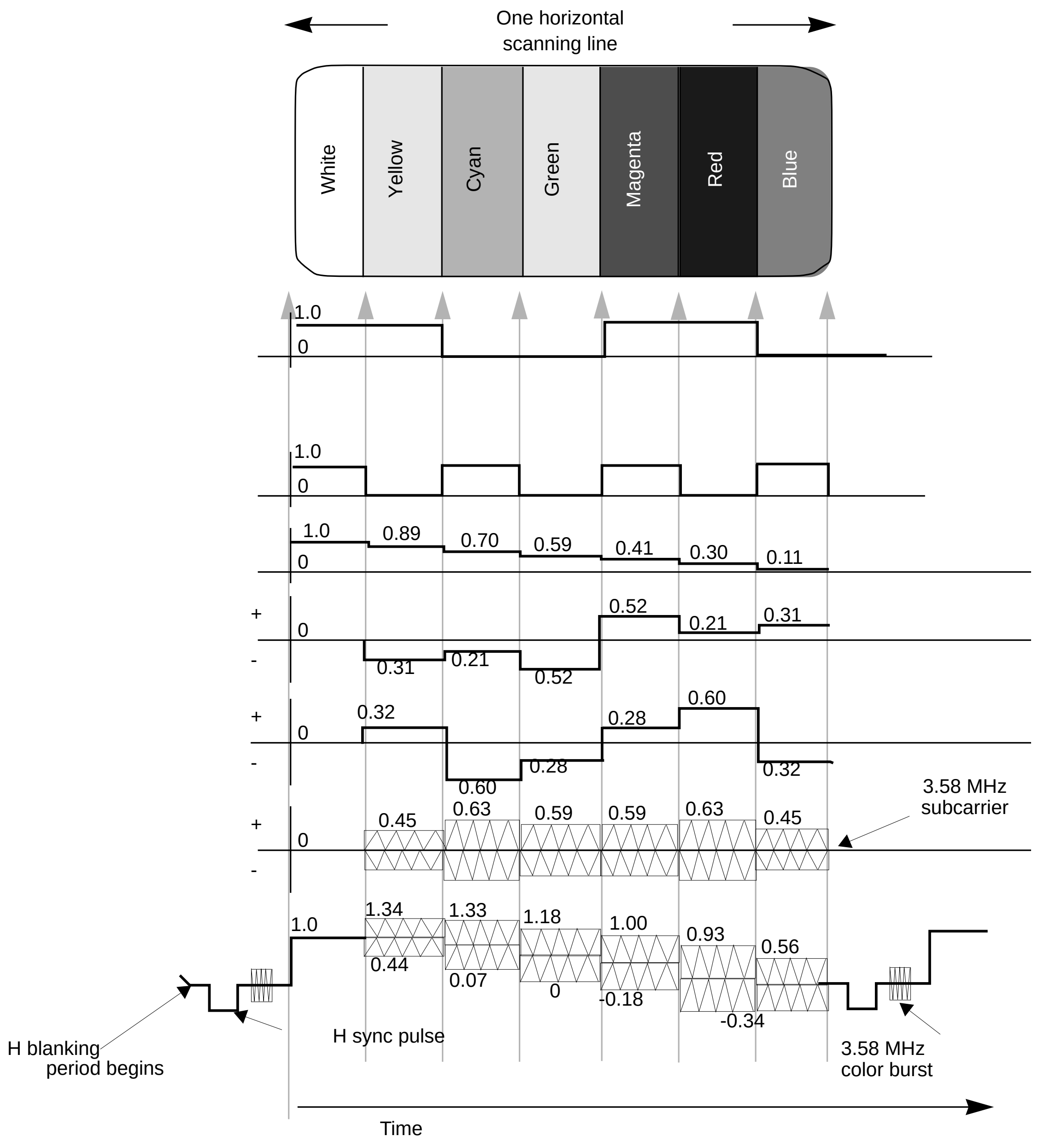


Figure GI-3 Component Video Signals

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 0.0 MHz to 0.5 MHz. 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 x 1035 viewable resolution), trilevel sync, and 30 MHz 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).

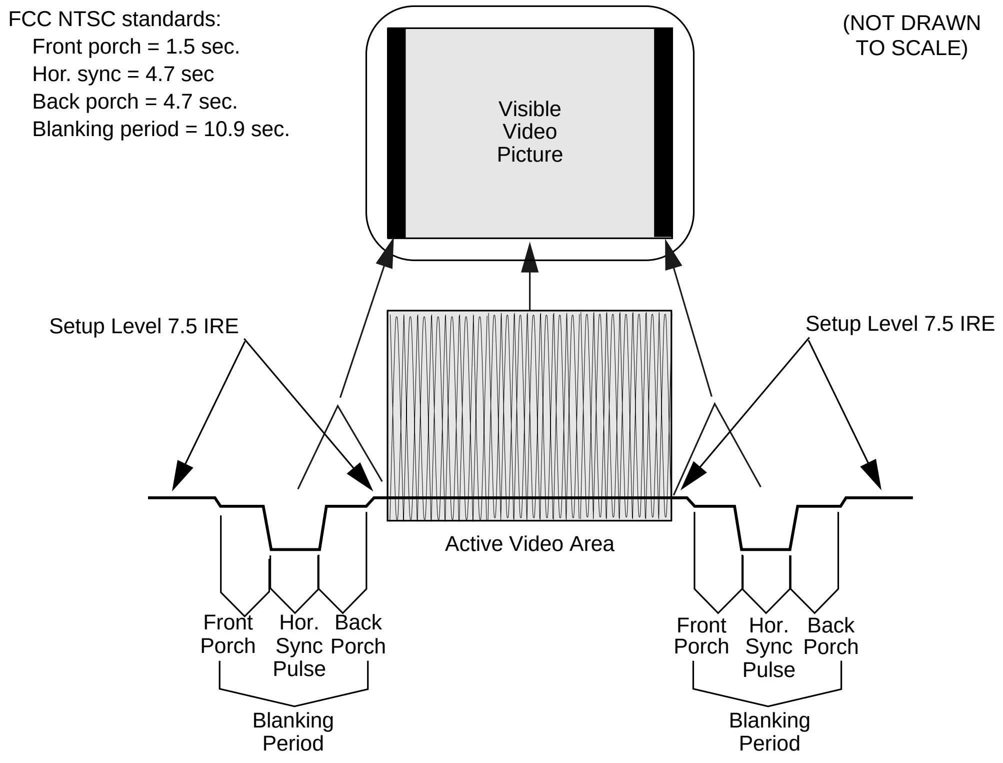


Figure Gl-4 Horizontal Blanking

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*.

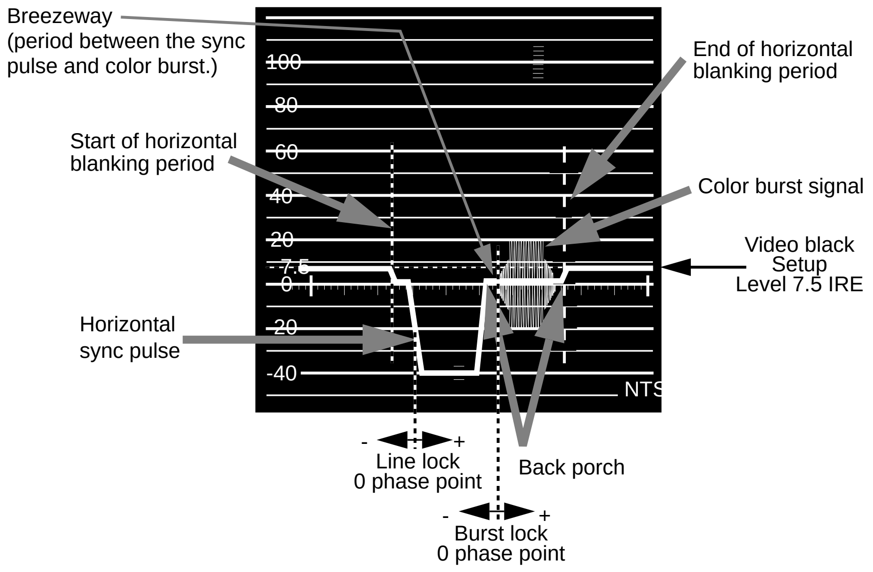


Figure Gl-5 Horizontal Blanking Interval

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 3.58 MHz (NTSC) or 4.43 MHz (PAL) C signal indicate the different hues in the picture information.

hue-saturation-intensity

A tristimulus 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 3.58 MHz C signal in quadrature with the I signal. Hues are green and magenta. Bandwidth is 0.0 MHz to 0.5 MHz. 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 0.0 MHz to 0.5 MHz. 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 3.58 MHz 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

Sequentiel 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*.

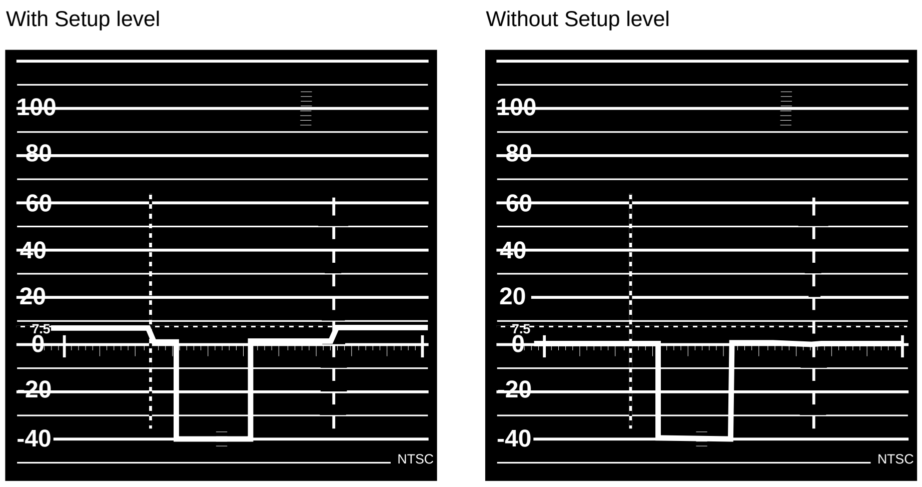


Figure Gl-6 Waveform Monitor Readings With and Without Setup

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).

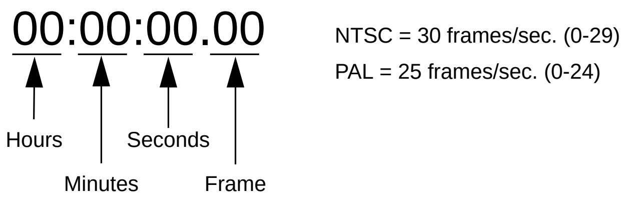


Figure Gl-7 SMPTE Time Code

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 75Ohm system; therefore a 75Ohm 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-12 lists major videotape formats.

Table Gl-12 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.

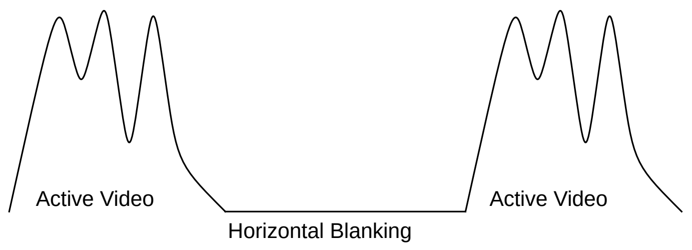


Figure Gl-8 Red or Blue Signal

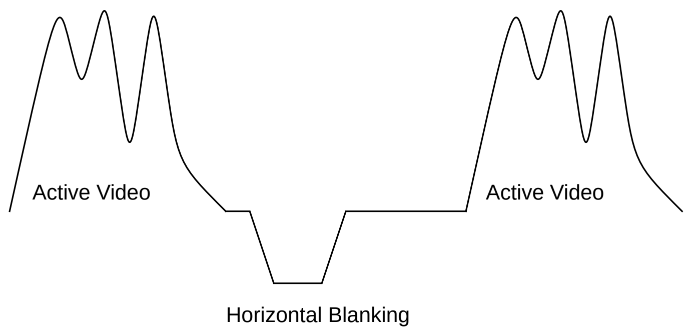


Figure Gl-9 Y or Green Plus Sync Signal

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.

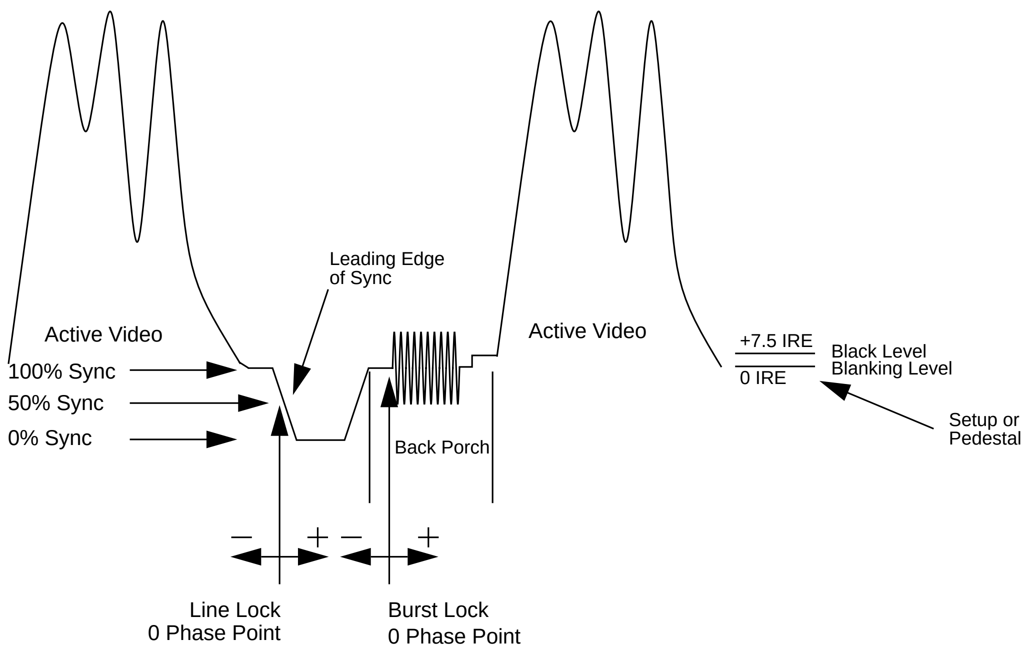


Figure Gl-10 Video Waveform: Composite Video Signal With Setup (Typical NTSC)

Figure Gl-11 shows a composite video signal without setup.

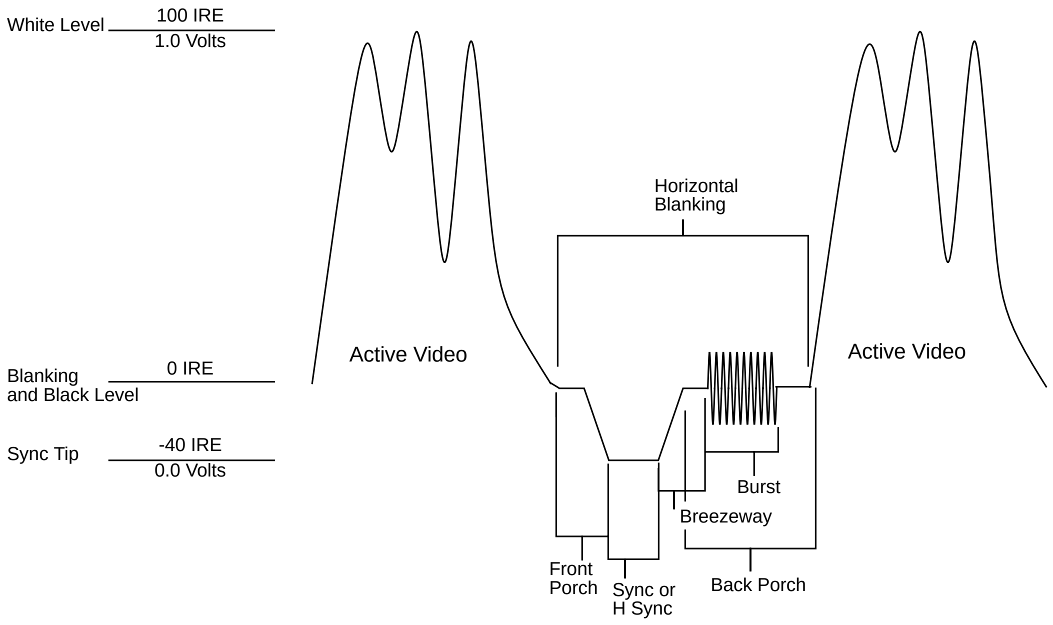


Figure Gl-11 Video Waveform: Composite Video Signal (Typical PAL)

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 fsct) + Q sin(2 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 - 0.321B
Q = 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

A

application
  creating, 13-59
  sample, location, 3, 7
asynchronous I/O, 101
autophase
  and crosspoint mux, 90
  and timing glitch, 91
  control, 173
autoswap, for texture node, 148, 159, 202
autowipe, 178

B

blender node, 104-112, 175-179
  controls, 176-179
blending, 103-124
  before or after zooming, 29
  node, 104-112
    setting up, 106-108
buffer, 8
  alignment, 99
  and data transfer, 43-59, 97
  creating for video data, 52-53
  getting DMediaInfo and image data from, 57
  reading data from, 54-57
  reading frames to memory from, 56
  registering, 54

C

caching, 98
capture, 97-102
CC1 memory source node, 182, 183
chroma
  blanking when sending blender output to video, 108
  keying, 103, 116-118, 122
client, 5
color space
  conversion, 209-227
    and image processing, 136-139
    controls
      image-processing, 137-139
      standard, 135
    custom LUTs, coefficients, 136-137
    full-range, limited-range data, 134
    math operations, 211-212
    packings, 129-130
    performing, 125-145
    port, 133-134
    precision, 212
    range, 130-131, 213-215
    saved in memory, 131
    software model, 126-128
    video formats, 126

connection, 11, 82-89
connectivity. *See* connection
constant hue, 132-133
contcapt.c (OpenGL), 60
control, 8, 22-37, 61-67, 169-207
  access, 107, 170
  and nodes, 169-207
  blender, 107-108
  blending, 104, 106
  classes, 65-66
  default values explained, 169
  fraction ranges, 65
  groupings, 66-67
  in header file, 61
  keying, 112-121
    listed, 113
  type and values, 64-65
  type explained, 170
  VL, listed, 62-63
conventions, xv
crosspoint mux, 182
  and LUT, 173
  timing restrictions, 90-91
ctrlusage, 19

D

daemon, video, 5-6
  opening connection to, 15
data transfer, 14
  ending, 58-59
  starting, 49-50, 54
  to and from memory, 22-32
decimation, 28-30, 32
  texture node, 155-156
deinterlacing control, 199
dev_mgv.h, 7, 15, 61
device, 8
  ID, getting, 18
  management, 5-6
  node, 171-174
    controls, 171-174
  video, transferring data, 43-59
direct I/O to disk, 99
DMbuffer, 43-51
DMediaInfo, getting from buffer, 57
drain, 9
  blending and zooming, 29
  control for default, 171
  contrrol for default, 16
  node controls, setting, 22-37
  See also memory node, screen node, video node
dual-link
  texture node, 147, 148, 200
    packing, 200
  video drain node, 189
  video source node, 183

E

error codes, 163-167
event
  capture into TRAM, 154
  handling, 69-80
    routines, 75
  listed, 71-74
  masks, 20-21
  querying, 75-76
  specifying path-related, 20-22
  trigger, 49
  type, 76
eventex.c, 80
explicit routing, 20

F

fades, 119-121
field dominance
  memory drain node control, 193
  memory source node control, 186
  texture node, 157, 201
field mask, 33
filter, horizontal smoothing, 122, 179
flat background
  controls, 111
  generator, 105
  setting and turning off, 111
framebuffer node, 181-182
  and CC1 memory, 182

G

glBindTextureEXT, 159
glCopyTexSubImage2DEXT, 161
glEnable, 159
glGenTexturesEXT, 159
glGetError(), 161
glHint, 159
glitch, 89-90
glTexImage2D, 160
glTexParameteri, 160
glXMakeCurrentReadSGI, 160
GPI trigger, controls, 174
GST control access, 107, 170

H

header file
  OCTANE Digital Video, 7, 15
  VL, 7, 15
horizontal smoothing filter, 122, 179

I

image data, getting from buffer, 57
implicit and explicit routing, 20
  See also connection
I/O, direct to disk, 99

K

keyer, 122
  and alpha source A, 105
  controls, 176-177
  purpose, 105
keying, 103, 112-121
  See also chroma keying, luma keying, transitions

L

linking, 15
luma keying, 103, 114-115, 122
LUT, selecting, control for, 173
-lvl, 15

M

memory
- and data transfer, 22-32
- node, 183-194
  - drain, 189-194
    - controls, 190-194
    - rounding, 189-190
  - source, 183-188
    - CC1 and framebuffer, 182
    - controls, 184-188
- reading from buffer to, 56
- sending frames to video from, 57

memtovid, 7

mipmap, 147, 158, 201
- and scaling, 157

mtov.c (OpenGL), 60

multiple clients, 5

N

node, 8, 169-207
- adding, 18
- blender, 104-112, 175-179
  - controls, 176-179
  - setting up, 106-108
- color-space conversion, 179-181
- defined, 9-10
- device, 171-174
  - controls, 171-174
- framebuffer, 181-182
- memory, 183-194
  - drain, controls, 190-194
  - source, controls, 184-188
- screen, 194-199
  - drain, controls, 198-199
  - source, controls, 195
- setting controls, 22-37
- specifying, 15-16
- texture, 147-161, 199-202
  - controls, 153-159, 200-202
- video, 202-207
  - drain, controls, 205-207
  - source, controls, 203

normalization, 104, 108-110, 176

O

OpenGL
- functions for texture mapping, 159-161
- programs, 60

P

packings, 25-28
- color-space conversion, 129-130

path, 8
- blending, 10
- creating, 17
- creating and setting up, 17-22
- defined, 9-10
- setting up, 18-19
- specifying events, 20-22
- specifying nodes on, 15-16

picture glitch, 89-90

playback, 97-102

port
  defined, 10-11
  VL_BLENDER, 175
  VL_CSC, 133-134
  VL_DEVICE, 171
  VL_FB, 182
  VL_MEM
    drain, 189
    source, 183
  VL_SCREEN
    drain, 196
    source, 194
  VL_TEX, 200
  VL_VIDEO
    drain, 205
    source, 203
Porter-Duff model, 108-110, 176
  See also normalization
positioning windows, 195, 197

R

restart policy, 187, 193
return codes, 163-167
rounding for memory drain, 189-190

S

sample programs, 3, 7, 80, 140-145, 161
screen node, 194-199
  drain, controls, 198-199
  source, controls, 195
shadow, 111
  controls, 112, 179
simpleccapt.c, 60
simplegrab.c, 60
simplem2v.c, 60
simplev2s.c, 60
source, 9
  blending and zooming, 29
  control for default, 16, 171
  node controls, setting, 22-37
  See also memory node, screen node, video node
starvation policy, 188
status information, 6
streamusage, 19
syntax, 13
syssgi, 101

T

texture
  mapping, 147-161, 199-202
    block diagram, 148
    controls, 153-159
    OpenGL functions, 159-161
  node, 147-161, 199-202
    timing, 151-152
tiles, 119-121
timing
  glitch, 89-90
  texture node, 151-152
tools, VL, 6-7
transitions, 103, 119-121
trigger, 49

V

VBAR mux, 87, 90-91
vcp, 6
VGI1
  memory drain node, 189
  memory source
    node, 183
    timing restrictions, 91
video
  daemon, 5-6
    opening connection to, 15
  data transfer, 43-59
    ending, 58-59
    starting, 49-50, 54
    to and from memory, 22-32
  displaying data onscreen, 41-42
  drain, 9
  format, 12
    converting, 25-28
    for color-space conversion, 126
  node, 202-207
    drain, 204-207
        controls, 205-207
    source, 202-204
        controls, 203
  sending frames from memory to, 57
  source, 9
  unit, defined, 184
videod, 5-6
videoin, 6
Video Library. *See* VL
videoout, 6
videopanel, 6
vidtomem, 7
vidtomem.c (OpenGL), 60
vidtotex.c, 161
vintovout, 7
VL
  capabilities, 3-4
  control, 22-37, 61-67
    blending, 104, 106
    keying, 112-121
    See also control
  device management, 5-6
  header files, 7, 15
  programming model, 13-14
  requirements for running, 15
  status information, 6
  syntax, 13
  system software architecture, 4
  tools, 6-7
VL_BLENDER, 175-179
  See also blender
VL_CAP_TYPE, 33-37
  See also control *and* node
  texture node, 153
VL_CSC, 125-139, 179-181
  See also color space conversion
VL_DEVICE, 171-174
  See also device node
VL_FB, 181-182
  See also framebuffer node
VL_FORMAT, 25
  See also control *and* node
VL_MEM, 183-190
  See also memory node
vl_mgv.h, 7, 15, 61
VL_OFFSET, 32
  See also control *and* node
  texture node, 153, 156
VL_ORIGIN, 32
  See also control *and* node
VL_PACKING, 22, 25-28
  See also control *and* node
  texture node, 154
VL_RATE, 33-37

See also control *and* node
texture node, 154
VL_SCREEN, 194-198
See also screen node
VL_SIZE, 31, 32
See also control *and* node
texture node, 155, 156
VL_TEXTURE, 147-161, 199-202
See also texture node
VL_TIMING, 24
See also control *and* node
VL_VIDEO, 202-207
See also video node
VL_ZOOM, 28-30, 32
See also control *and* node
texture node, 155, 156
vlAddCallback(), 69
vlAddNode(), 18
vlBeginTransfer(), 49
VL buffer, 43-54
vlCheckEvent(), 76
vlCloseVideo(), 58-59
vlcmd, 6
vlCreateBuffer(), 53
vlCreatePath(), 17
vlDeregisterBuffer(), 58
vlDestroyBuffer(), 58-59
vlDestroyPath(), 58-59
vlEndTransfer(), 49, 58
vlEventToName(), 76
vlGetActiveRegion(), 57
vlGetControl(), 24
vlGetDevice(), 18
vlGetDMediaInfo(), 57
vlGetFD(), 76
vlGetImageInfo(), 57
vlGetLatestValid(), 55, 57
vlGetNextFree(), 57
vlGetNextValid(), 55, 56
vlGetNode(), 15, 106
vlGetTransferSize(), 52
vl.h, 7, 15
vlinfo, 7
vlMainLoop(), 69
vlNextEvent(), 76
vlOpenVideo(), 15
vlPeekEvent(), 76
vlPutFree(), 55, 57
vlPutValid(), 57
vlRegisterBuffer(), 54
vlSelectEvents(), 20, 69, 76, 77
vlSetConnection(), 20, 82-89
vlSetControl(), 24
vlSetupPaths(), 18

W

window positioning, 195, 197
wipes, 110, 119-121
controls, 177-178

Z

zoom, 28-30, 32
before or after blending, 29

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3513-001.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389