OCTANE™ Personal Video Programmer's Guide

Document Number 007-3595-001

CONTRIBUTORS

Written by Carolyn Curtis
Illustrated by Dany Galgani, Carolyn Curtis, Cheri Brown, and Scott Pritchett
Edited by Christina Cary
Document Production by Max Anderson
Engineering contributions by Michael Minakami, Bruno Wolf, Farrell Wymore, Grant Dorman, Chris Pirazzi, Scott Pritchett, and Ashok Yerneni
St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

OCTANE™ Personal Video Programmer's Guide
Document Number 007-3595-001

Contents

List of Figures

List of Tables

About This Guide

The OCTANE XIO Personal Video option board enables a Silicon Graphics OCTANE workstation to input and output graphic and video images and record them to disk or videotape.

The OCTANE Personal 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 Personal Video option and gives step-by-step instructions for creating VL programs that make use of OCTANE Personal 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 the OCTANE Personal Video option board.

Structure of This Document

This guide contains the following chapters and appendixes:

- Chapter 1, “Features of the OCTANE Personal Video Option,” introduces the features and capabilities of the OCTANE Personal Video board. It explains VL features and architecture, and presents the VL programming model.
- Chapter 2, “Setting Up Your VL Application,” explains how to open a connection to the video daemon, specify nodes, and set up a data path. It also summarizes the VL programming model.
- Chapter 3, “Setting Parameters for Data Transfer,” explains how to use controls for defining frame data size, video format and timing, color space, scaling, and other basic parameters.

- Chapter 4, “Synchronizing Data Streams and Signals,” gives instructions for using special signals—unadjusted system time (UST), media stream count (MSC), and Internal Video Sync signal—for refining synchronization in your application.
- Chapter 5, “Transferring Video Data and Ending Data Transfer,” explains using buffers for transferring video data, gives the steps for ending data transfer, and summarizes example programs that illustrate how to create simple video applications included in the software.
- Chapter 6, “Using VL Controls,” explains the VL control type and values, VL control fraction ranges, VL control classes, and VL control groupings.
- Chapter 7, “Event Handling,” presents the VL events for the OCTANE Personal Video option and details querying VL events, creating a VL event loop, and creating a main loop with callbacks.
- Chapter 8, “Video Real-Time Capture and Playback,” gives guidelines for optimizing capture or playback to system memory or disk.
- Appendix A, “Return Codes,” lists and explains VL return messages for the OCTANE Personal Video board.
- Appendix B, “OCTANE Personal Video Nodes and Their Controls,” summarizes the OCTANE Personal Video nodes and their controls.
- Appendix C, “Pixel Packings and Color Spaces,” sets forth all packing formats used by the OCTANE Personal Video option and includes information on sampling patterns.
- Appendix D, “OCTANE Personal Video Color-Space Conversions,” explains color spaces, mathematical operations performed during conversions, and implications of color-space conversions.

An index completes this guide.

Other Documents

The *Digital Media Programming Guide* (007-1799-060; online only) is available with the IRIX digital media development environment software (*dmedia_dev*). This guide is also online in the following locations:

- IRIS InSight Library: from the Toolchest, choose Help > Online Books > SGI EndUser or SGI Admin, and select the applicable owner’s or hardware guide.

Once you are in the library, choose Catalogs > Hardware Catalog > and look under the Owner's Guides for the applicable owner's guide.

- Technical Publications Library: if you have access to the Internet, enter the following URL in your Web browser location window: http://techpubs.sgi.com/library/

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 Personal Video Option

The OCTANE Personal Video board makes it possible to use your OCTANE workstation for video-based operations, such as

- capturing the screen display: recording directly to disk any portion of the screen, from the full 1280 x 1024 resolution down to 2 pixels by 2 lines, for editing with bundled media editing tools or converting to standard NTSC- or PAL-encoded video
- writing or using custom video applications
- using applications included with the OCTANE workstation:
  - videoconferencing, using the O2Cam digital camera and the InPerson videoconferencing software for real-time group collaboration
  - Web authoring, using video compression algorithms implemented through software and the Web-based authoring tool suite included with the OCTANE system for distributing video via the World Wide Web

This chapter introduces

- "OCTANE Personal Video Board Capabilities" on page 2
- "Video Library Capabilities" on page 10
- "VL System Software Architecture" on page 11
- "VL Architectural Model of Video Devices" on page 14
- "OCTANE Personal Video Formats" on page 18

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

OCTANE Personal Video Board Capabilities

This section explains

- “Board Features” on page 2
- “Video Capture” on page 6
- “Video Output” on page 7
- “Adjustments and Conversions” on page 9
- “Square-to-Nonsquare Pixel Filter” on page 9
- “Timing” on page 10

Board Features

The OCTANE Personal Video option supports 1280 x 1024 at 50 Hz, 60 Hz, and 72 Hz. Input speed is limited by the pixel clock and input to the video output controller (VOC). The VOC receives the complete graphics stream as input; it can select all or a portion of the screen to resize and fit into a standard video raster.

The section “OCTANE Personal Video Software Model” on page 17 explains how the board looks to the software.

Figure 1-1 diagrams the OCTANE Personal Video board.

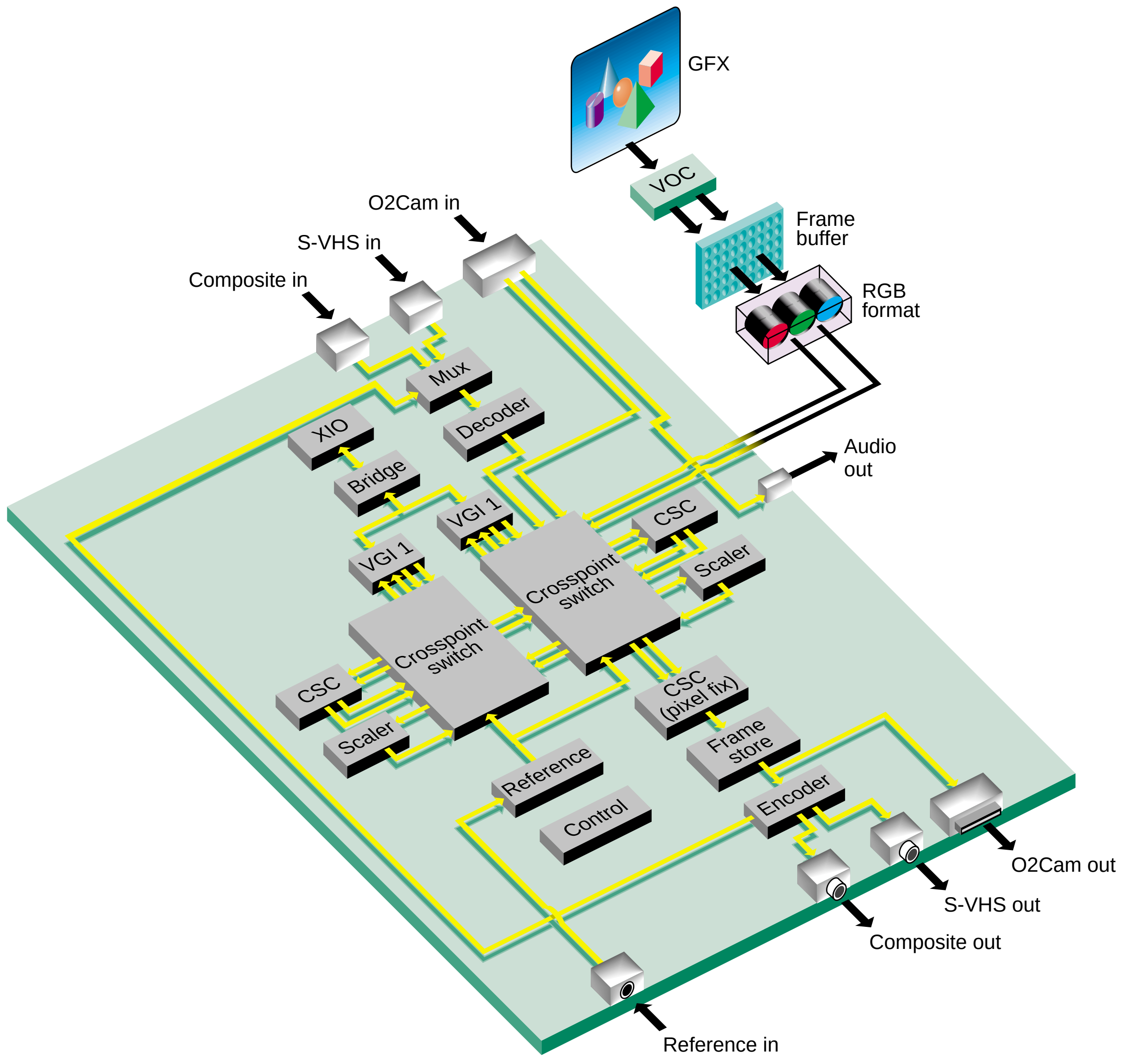


Figure 1-1 OCTANE Personal Video Board

Figure 1-2 shows OCTANE Personal Video option connections to OCTANE graphics and connectors.

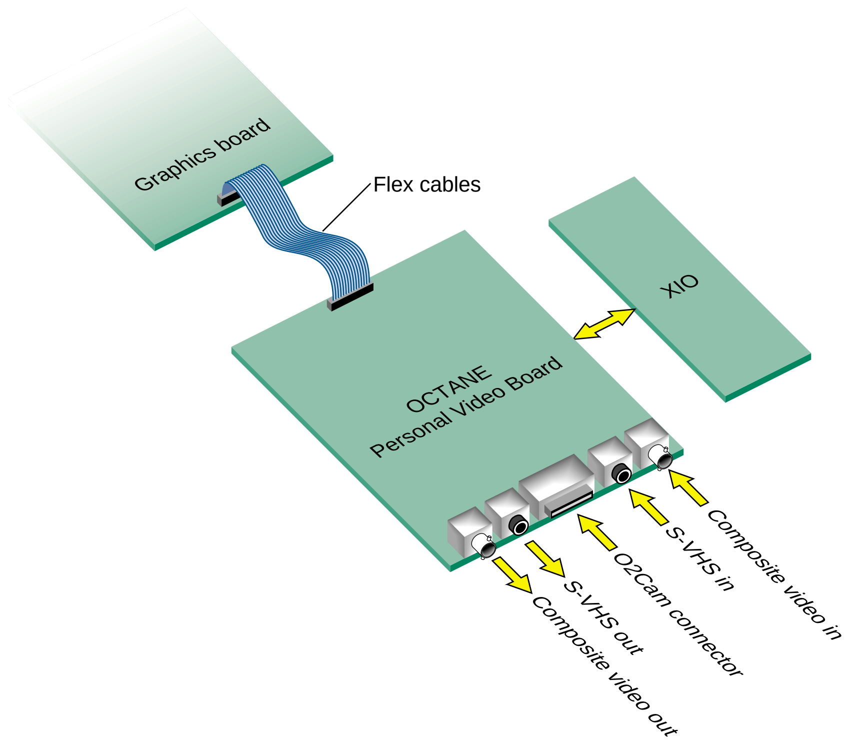


Figure 1-2 OCTANE Personal Video Connection to OCTANE Workstation

Note: In Figure 1-2, some I/O has been omitted for clarity.

All graphics data comes through three dedicated internal flex cables connected to OCTANE graphics. Through connectors on the OCTANE Personal Video board itself, you can

- send and receive live analog video, either S-Video or composite; you can software-select between the 525/60 NTSC and 625/50 PAL television standards
- receive component digital video from any serial CCIR-601/SMPTE-259M-compliant device
- send Silicon Graphics component digital video from any serial CCIR-601/SMPTE-259M-compliant device
- receive live video from the O2Cam digital camera, which you can convert to CCIR601 with a third-party adapter
- pass through audio from the O2Cam digital camera

Figure 1-3 shows the connectors.

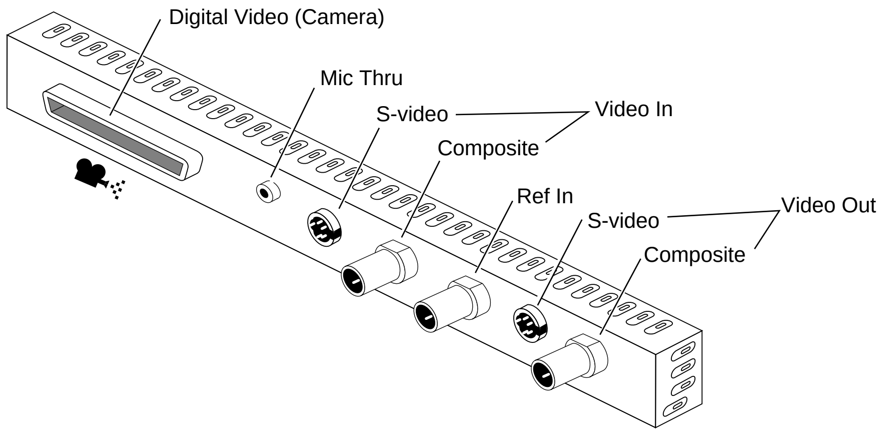


Figure 1-3 OCTANE Personal Video Option Board Connectors

Table 1-1 summarizes board connectors.

Table 1-1 OCTANE Personal Video Connectors

Input	Output	Format
Composite	Composite (single-channel video out)	BNC (75-ohm terminated)
S-Video (single-channel video in)	S-Video (single-channel video out)	4-pin mini-DIN
Composite locking reference	N/A	BNC (75-ohm terminated)
N/A	Audio: microphone through	3.5-mm jack
Silicon Graphics digital video (O2Cam digital camera)	Silicon Graphics digital video, with optional third-party devices connected	68-pin D connector

The board can genlock to external reference, O2 video, or Internal Video Sync. The sources for external reference are reference, composite video, and S-VHS video. The lock is color-frame accurate with respect to reference or composite video. Other important signals (reset, power rails, and so on) come from the board connection at the XIO slot.

For hardware details, including cabling information, see the *OCTANE Personal Video Installation Guide*.

Video Capture

The OCTANE Personal Video board can capture video from four sources:

- composite video, digitized and decoded into the nonsquare pixel YUV format
- S-VHS video, digitized into standard YUV format
- O2Cam connector (digital): parallel input as YUV (D1)

  The O2Cam connector can also be used for parallel D1 output.
- graphics signals encoded in 24-bit RGB

Video input processing captures input from one or more of these sources and transfers it to memory. During capture, you can scale (reduce, or minify) the image horizontally and vertically, such as for aspect ratio correction. The scaler is a YUV device only.

You can also color-space convert the image (CSC). The color-space converters allow transformation from YUV to RGB or from RGB to YUV.

Finally, you can DMA the image into memory; the board supports two DMA channels. Figure 1-4 diagrams the processing path.

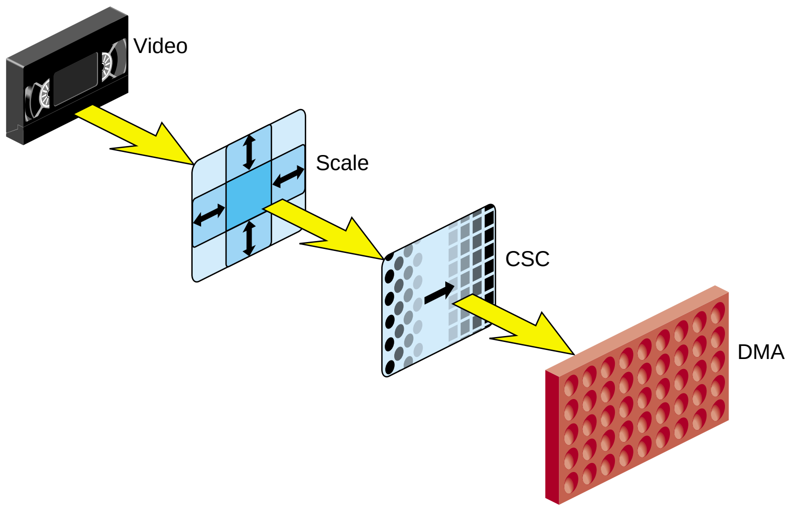


Figure 1-4 Video Input Processing

Video Output

Video output takes an image from memory and converts it into video, or performs the print-to-video process. Output video sources are memory, graphics (VOC), or video in (analog or digital). From memory, the board can process 8-bit YUV 422 or 24-bit RGB. (Although you can transfer 10- bit YUV, the processing path is only 8 bits wide.) If required, you can color-space convert the signal into the YUV 422 format.

After color-space conversion, the signal can go into a sampling rate converter, which scales a square-pixel image into a nonsquare-pixel image. The encoder is a nonsquare encoder; it expects 720 pixels per line. This conversion might be necessary because the video encoder operates on nonsquare pixels only; it does not directly accept square-pixel data. This filter is explained further in “Square-to-Nonsquare Pixel Filter,” later in this section.

Figure 1-5 diagrams the processing path of the output block.

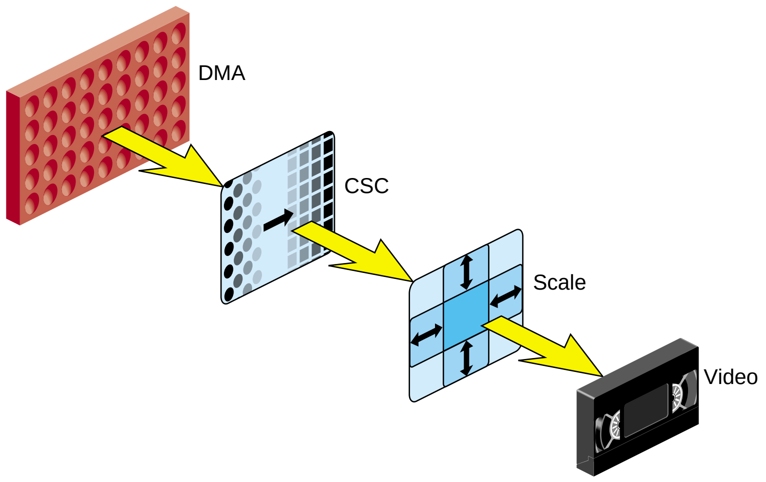


Figure 1-5 Video Output Processing

The scaler scales down horizontally and vertically only (no upscaling).

Figure 1-6 diagrams the processing path using graphics as the video source.

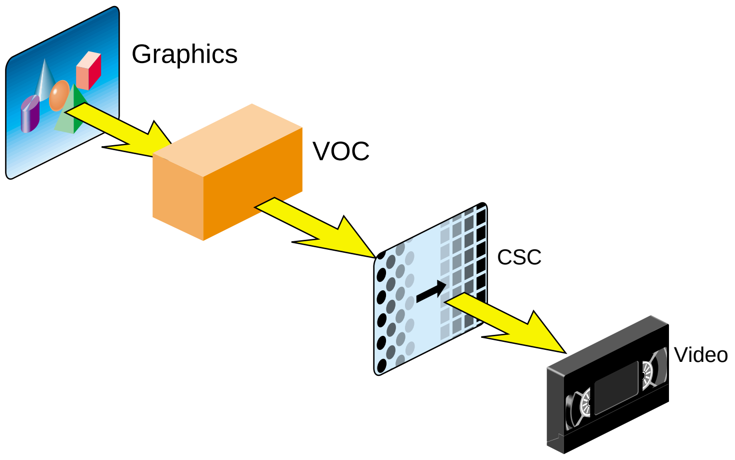


Figure 1-6 Video Output via VOC Chip

Adjustments and Conversions

The OCTANE Personal Video option supports

- Internal Video Sync: a synchronization signal produced or consumed by some audio and video devices

  This signal is explained in Chapter 2, “Setting Up Your VL Application.”
- output subcarrier horizontal phase (SCH): subcarrier phase can be adjusted through software
- coarse H: window-controlled coarse horizontal timing
- color-space conversion: RGB-to-YUV and YUV-to-RGB conversion

  OCTANE Personal Video color-space conversion is explained in “VL_COLORSPACE,” in Chapter 3, and Appendix D, “OCTANE Personal Video Color-Space Conversions.”.
- digital dither filter to reduce artifacts (a bypassable filter)
- square-to-nonsquare pixel conversion

Square-to-Nonsquare Pixel Filter

Most applications native to the computing environment work in the square-pixel aspect ratio. However, professional video applications require nonsquare pixels, which preserve the correct aspect ratio.

The OCTANE Personal Video board can process video in square-pixel format. For optimum video quality, however, it outputs only nonsquare pixels. To accommodate both environments, the board includes a bypassable square-to-nonsquare pixel filter that operates on video.

Square-pixel NTSC has 640 active pixels per line, based on a sample clock of 12.2727 MHz. Square pixel PAL has 768 pixels per line, based on a sample clock of 14.75 MHz. Nonsquare pixel NTSC and PAL both have 720 pixels per line, based on a sample clock of 13.5 MHz. Thus, in the horizontal direction, the filter must decimate in PAL and interpolate in NTSC. (For simplification, the vertical scale is not considered.)

Timing

The OCTANE Personal Video board utilizes timing signals from three kinds of sources:

- free run (default)
- Internal Video Sync, from another producer of this signal
- another external reference signal, such as from an analog reference, the O2Cam digital camera, or input video

The board can produce the Internal Video Sync signal.

Video Library Capabilities

The Video Library provides a software interface to the OCTANE Personal 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*; 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 OCTANE Personal Video board software includes a graphical user interface, */usr/sbin/vcp*. See the *OCTANE Personal Video Installation Guide* for how to use this panel with the OCTANE Personal Video option.

VL System Software Architecture

This section describes features of these VL system components and tools:

- "Video Daemon" on page 12
- "Generic Video Tools" on page 13
- "Library and Header Files" on page 14

Figure 1-7 diagrams the interaction between the VL, the video daemon, the kernel, the hardware, and the X Window System server.

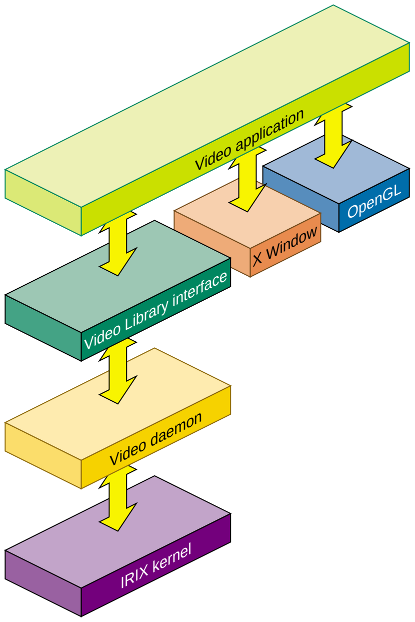


Figure 1-7 VL System Components

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 file for OCTANE Personal Video is */usr/include/dmedia/vl_evo.h* (linked to */usr/include/vl/dev_evo.h*).

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 Personal Video board, or external, such as a videotape recorder connected to the board.
- *nodes*: sources, drains, and internal nodes
- *paths*, connecting sources and drains
- *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
- *buffers*: 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* and *path*. This section explains

- "Node" on page 15
- "Path" on page 15
- "OCTANE Personal Video Software Model" on page 17

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 memory or a video output), or a *device* (video).

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 1-8 shows a simple path in which a frame from a videotape is captured to memory.

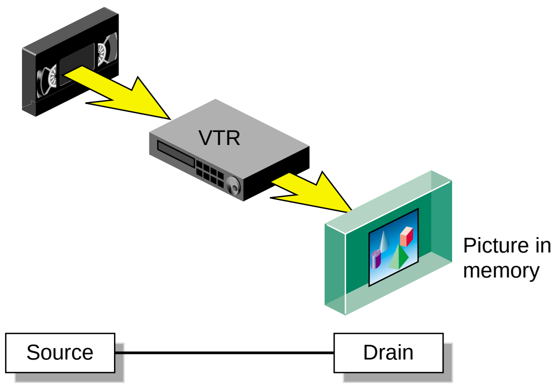
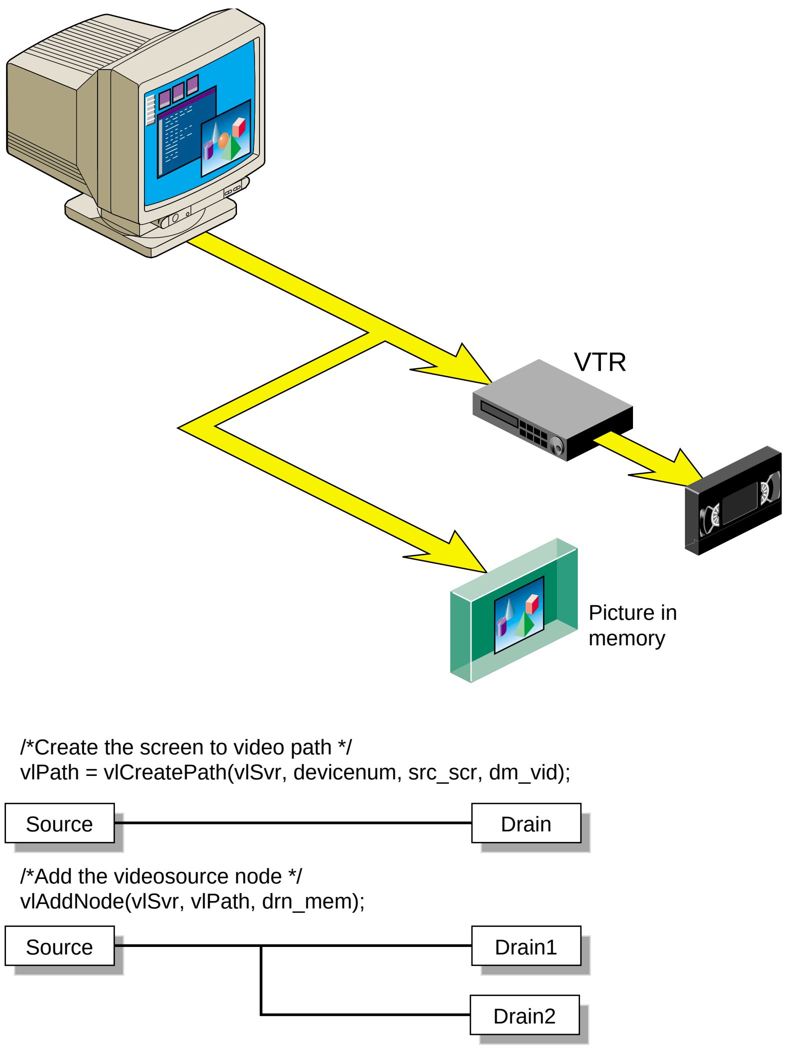


Figure 1-8 Simple VL Path

Figure 1-9 shows a more complex path with two video drains: a frame from a workstation window is captured to memory and sent to video output simultaneously. This path is set up in stages.



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

```
/*Add the videosource node */
vlAddNode(vlSvr, vlPath, drn_mem);
```

Figure 1-9 Complex VL Path

OCTANE Personal Video Software Model

Figure 1-10 diagrams how the OCTANE Personal Video board looks to software.

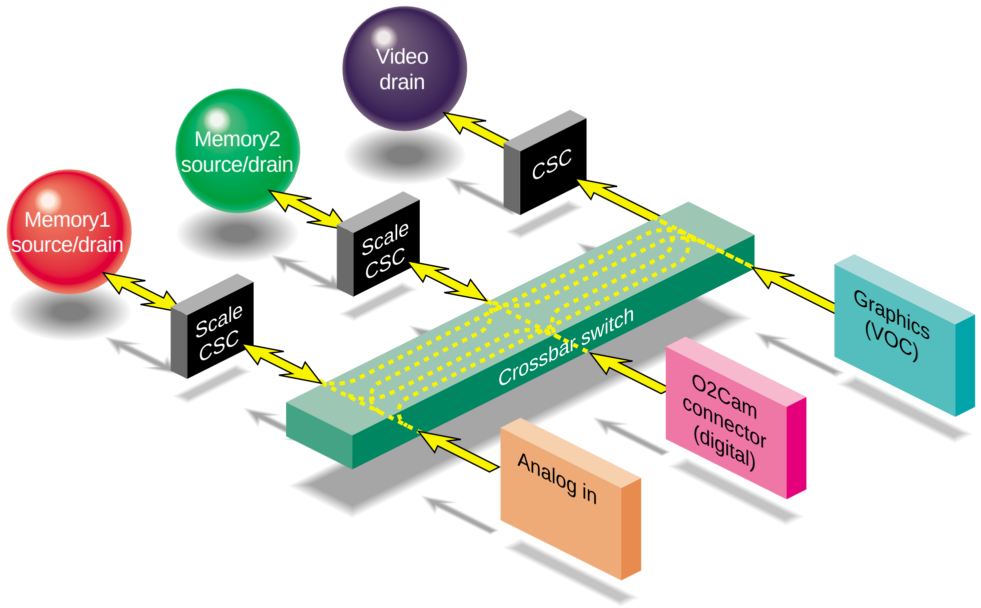


Figure 1-10 OCTANE Personal Video Software Model

Memory nodes appear as both source and drains. These nodes cannot be both at the same time; source and drain modes are mutually exclusive.

The board's crossbar switch allows any source to connect to any drain. Multiple connections can be made simultaneously. For example, the board can capture video and transfer it to memory simultaneously; it can capture video and transfer it to one memory drain in YUV format and the other in RGB format, all simultaneously.

OCTANE Personal Video Formats

The OCTANE Personal 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 1-2 summarizes the formats that the OCTANE Personal Video board supports.

Table 1-2 Video Formats for OCTANE Personal Video Memory Nodes

Format	Signal
Digital component YCrCb serial (VL_FORMAT_DIGITAL_COMPONENT_SERIAL)	YCrCb 4:2:2 serial digital signal with 8-bit words. Component ranges are 16 to 235. Conforms to the CCIR-601 specification.
SMPTE YUV (VL_FORMAT_SMPTE_YUV)	Contains YUV components in the range 1-254; superblack and superwhite values can be present.
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.
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).

These formats apply to memory source and drain nodes only.

Chapter 2

Setting Up Your VL Application

Used in conjunction with your OCTANE Personal Video option, Video Library (VL) calls let you capture any part of the screen display and scale it down, output computer-generated graphics to videotape or the O2Cam digital camera, and output the input video source to the graphics monitor, to a video device such as a VCR, or both.

This chapter explains the first steps for creating video programs for OCTANE Personal 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"

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. Start the data transfer (**vlBeginTransfer()**).
11. For transfers involving memory, get the data and manipulate it (DMbuffers: **vlDMBufferGetValid()**, **vlGetActiveRegion()**, **dmBufferFree()**; VL buffers: **vlGetNextValid()**, **vlGetLatestValid()**, **vlGetActiveRegion()**, **vlPutFree()**).
12. 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_evo.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 file for OCTANE Personal Video is */usr/include/dmedia/vl_evo.h* (linked to */usr/include/vl/dev_evo.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_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_MEM: region of workstation memory • VL_SCREEN: workstation screen (source only)

- VL_VIDEO: connection to a video device; for example, a video tape deck or O2Cam digital camera

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

number	is the number of the node in cases of two or more identical nodes, such as three 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 video source node and a device node:

```
vlAddNode(vlSvr, vlPath, src_vid);
vlAddNode(vlSvr, vlPath, dev_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);
}
```

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

Table 2-1 (continued) VL Event Masks

Symbol	Meaning
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 7, "Event Handling."

Chapter 3

Setting Parameters for Data Transfer

Transferring data to or from memory requires creating a DMbuffer or VL buffer, as explained in "Transferring Video Data to and From Devices" in Chapter 5. This chapter explains how to set node controls for data transfer and consists of these sections:

Device-Independent Controls for OCTANE Personal Video

To set frame data size and to convert from one video format to another, apply controls to the nodes.

Table 3-1 summarizes important data transfer controls for source and drain nodes. These controls are highly interdependent, so the order in which they are set is important; set them in the order in which they appear in the table. In most cases, the value being set takes precedence over other values that were previously set.

Note: Changes in one parameter may change the values of other parameters set earlier; for example, clipped size may change if VL_OFFSET is set after VL_SIZE.

Table 3-1 Nodes and Data Transfer Controls

Control	Sets ...	Memory Nodes	Screen Node	Video Nodes
VL_FORMAT	Format	Selects color space	Not applicable	Selects physical connector
VL_TIMING	Video timing	Yes	Yes	Yes
VL_CAP_TYPE	Type of field(s) or frame(s) to capture	Yes	Not applicable	Not applicable
VL _COLORSPACE	Color space of video data in memory	Yes	Not applicable	Not applicable
VL_PACKING	Pixel packing (conversion) format	Changes pixel format of captured data	Not applicable	Not applicable
VL_ZOOM	Scaling down to any size between the full field/frame size and a certain number of pixels	Yes	Scales a selected region of the graphics display from unity (1/1) to 2 pixels by 2 lines	Set only to 1/1
VL_ASPECT	Horizontal scale factor; used with VL_ZOOM to correct aspect ratio	Effective horizontal scale factor is VL_ZOOM * VL_ASPECT	Effective horizontal scale factor is VL_ZOOM * VL_ASPECT	Not applicable
VL_SIZE	Clipping size	Yes	Yes	Full size of video, read only

Table 3-1 (continued) Nodes and Data Transfer Controls

Control	Sets ...	Memory Nodes	Screen Node	Video Nodes
VL_OFFSET	Position within larger area relative to VL_ORIGIN	Offset relative to video offset	Sets where the scaled images produced by this node are inserted into a video frame, for centering or other placement	Set only to (0,0)
VL_ORIGIN	Position within video	Not applicable	Screen position of first pixel displayed; works with VL_SIZE	Not applicable
VL_RATE	Field or frame transfer speed	Yes	Not applicable	Not applicable
VL_FLICKER _FILTER	Enables or disables flicker reduction	Not applicable	Yes	Not applicable
VL_FREEZE	Freezes the image	Not applicable	Graphic updates are not reflected in the generated video signal	Set only to FALSE for source nodes because device does not support frozen inputs

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)
```

VL_TIMING

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

Table 3-2 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

VL_TIMING is applicable on all OCTANE Personal Video nodes. However, the timing standard for the O2Cam digital camera input can be only NTSC square-pixel or CCIR 525-line nonsquare-pixel timing (VL_TIMING_525_SQ_PIX or VL_TIMING_525_CCIR601).

The VL_TIMING control affects how the video signal is sampled. Internally, the OCTANE Personal Video board represents all video signals as nonsquare (CCIR601 525-line or 625-line).

Use parameters VL_TIMING_525_SQ_PIX and VL_TIMING_625_SQ_PIX, which convert nonsquare pixels to square pixels to avoid multiple filters being applied to video.

VL_EVO_FILTER_TYPE

Once the aspect ratio is accounted for, square-pixel and nonsquare-pixel images have differently sized active regions. In nonsquare modes, the active region is 720 pixels across. In square modes, the active region is 640 pixels for NTSC and 768 pixels for PAL. Correcting for the aspect ratio with a 11/10 filter, the result is $640 \times 11/10 = 704$, or 16 pixels short of the nonsquare sampling.

The OCTANE Personal Video option software includes a memory drain and video drain control, VL_EVO_FILTER_TYPE, that selects between two methods of converting square pixel images to nonsquare pixel images, or vice versa:

- VL_EVO_FILTER_TYPE_FREQ selects the frequency-preserving filter, which preserves the aspect ratio of the image exactly.

  For example, a circle displayed on a nonsquare video monitor looks the same as on a square graphics display, with no distortion. However, the width of the active region is not the same: a line of 720 nonsquare pixels does not map directly to a line of 640 or 768 square pixels.

- VL_EVO_FILTER_TYPE_SPAT selects the spatially preserving filter, which preserves the width of the active region.

  This filter takes in 640 horizontal pixels and produces 720 horizontal pixels. In the other direction, the 720-pixel line in nonsquare mode maps directly to the 640 or 768 pixels in square mode.

  Although the spatially preserving filter preserves the contents of a line, it does not preserve the frequency characteristics of the image. For example, a circle in square pixel mode is slightly distorted in nonsquare pixel mode when displayed on a video monitor; however, the entire active region in the nonsquare case is filled.

Note: When square-pixel data is sent to video out, the conversion can be either frequency-preserving or spatially preserving. When video is captured to memory, only frequency-preserving conversions are performed for NTSC square-to-nonsquare conversion and PAL nonsquare-to-square conversions (regardless of the VL_EVO_FILTER_TYPE setting).

VL_FORMAT

VL_FORMAT has two sets of parameters, depending on whether it is applied to a memory node or a video node. (It is not applicable to the screen node.) For memory nodes, VL_FORMAT specifies the color space; for video nodes, VL_FORMAT selects the connector on the board.

This section discusses

- "Using VL_FORMAT to Specify Color Space" on page 32
- "Selecting the Input Connector" on page 33
- "Specifying the Video Drain Node" on page 35

Using VL_FORMAT to Specify Color Space

Generally, you set color space with VL_COLORSPACE. VL_FORMAT values are provided for compatibility with existing applications. Table 3-3 shows the correspondence.

Table 3-3 VL_FORMAT and VL_COLORSPACE Correspondence

VL_FORMAT Value	VL_COLORSPACE Value
VL_FORMAT_DIGITAL_COMPONENT_SERIAL	VL_COLORSPACE_CCIR601
VL_FORMAT_DIGITAL_COMPONENT_RGB_SERIAL	VL_COLORSPACE_RP175
VL_FORMAT_SMPTE_YUV	VL_COLORSPACE_YUV
VL_FORMAT_RGB	VL_COLORSPACE_RGB

Selecting the Input Connector

VL_FORMAT works in conjunction with VL_VIDEO node parameters to select the input connector on the OCTANE Personal Video option board. Applied to video source nodes, VL_FORMAT has five parameters, as summarized in Table 3-4.

Table 3-4 Using VL_FORMAT to Select Input Connector (Video Source Nodes)

Control Parameter	Connector
VL_FORMAT_SVIDEO	S-Video (Y/C)
VL_FORMAT_COMPOSITE	Composite (BNC)
VL_FORMAT_CAMERA	O2Cam connector (68-pin D)
VL_EVO_FORMAT_LOOPBACK	None; used for loopback from analog video out to analog video in with no cable necessary
VL_FORMAT_DIGITAL_COMPONENT_SERIAL	O2Cam connector used as digital input, with appropriate third-party converter attached

Note: The O2Cam connector can be a digital input when an appropriate third-party serial digital converter is connected to it.

These control parameters must be used in conjunction with VL_VIDEO source node parameters to select the connector:

- VL_EVO_NODE_NUMBER_VIDEO_1: digital video node, that is, the O2Cam input used with a third-party serial converter
- VL_EVO_NODE_NUMBER_VIDEO_2: O2Cam digital camera connector
- VL_EVO_NODE_NUMBER_VIDEO_3: analog video node; the VL_FORMAT control selects between composite, Y/C, or loopback

Table 3-5 summarizes how the VL_VIDEO source node parameters and the VL_FORMAT parameters work together.

Table 3-5 VL_VIDEO Source Nodes

To select ...	For Connector ...	Set VL_VIDEO Parameter ...	And VL_FORMAT Parameter ...
4:2:2 YCrCb	O2Cam connector with third-party serial digital converter	VL_EVO_NODE_NUMBER _VIDEO_1	VL_FORMAT_DIGITAL _COMPONENT_SERIAL
O2Cam digital camera	O2Cam connector with O2Cam digital camera attached	VL_EVO_NODE_NUMBER _VIDEO_2	VL_FORMAT_CAMERA
Composite	Analog video BNC	VL_EVO_NODE_NUMBER _VIDEO_3	VL_FORMAT_COMPOSITE
Y/C (S-Video)	Analog video 8-pin mini-DIN	VL_EVO_NODE_NUMBER _VIDEO_3	VL_FORMAT_SVIDEO
Loopback from analog video out to analog video in	None	VL_EVO_NODE_NUMBER _VIDEO_3	VL_EVO_FORMAT_LOOPBACK

To select a video input format, call **vlGetNode()** with the video source node number of interest. After setting up a path, set the VL_FORMAT control on that node to the appropriate value. For example, to select the composite video BNC, get a node by passing node number VL_EVO_NODE_NUMBER_VIDEO_3 to **vlGetNode()**, set up a path, and then set VL_FORMAT to VL_FORMAT_COMPOSITE, as illustrated in this example:

```
src_node = vlGetNode(svr, VL_SRC, VL_VIDEO,VL_EVO_NODE_NUMBER_VIDEO_3);
path = vlCreatePath(svr, dev, src_node, drain_node);
vlSetupPaths(svr, (VLPathList)&path, 1, VL_SHARE, VL_SHARE);
ctlVal.intVal = VL_FORMAT_COMPOSITE;
vlSetControl(svr, path, src_node, VL_FORMAT, &ctlVal);
```

Specifying the Video Drain Node

Because there is only one VL_VIDEO drain node, it has no number parameters and drives all output connectors at once. Thus, it is not strictly necessary to set a VL_FORMAT value to select the output connector. However, Table 3-6 shows VL_FORMAT values for various connectors.

Table 3-6 Using VL_FORMAT to Select Output Connector (Optional)

To select ...	For Output Connector ...	VL_FORMAT Value
Y/C (S-Video)	Analog video 4-pin mini-DIN	VL_FORMAT_SVIDEO
Composite	BNC	VL_FORMAT_COMPOSITE
4:2:2 YCrCb	O2Cam connector with third-party serial digital converter	VL_FORMAT_DIGITAL_COMPONENT _SERIAL

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 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.

Table 3-7 summarizes packing types for eight bits per component.

Table 3-7 Packing Types for Eight Bits per Component

Type	Use	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	U0	Y0	V0	Y1	U2	Y2	V2	Y3
VL_PACKING_YUVA_4444_8	YUVA 4:4:4:4	A0	U0	Y0	V0	A1	U1	Y1	V1
VL_PACKING_AUYV_4444_8	AUYV 4:4:4:4	V0	Y0	U0	A0	V1	Y1	U1	A1

Table 3-7 (continued) Packing Types for Eight Bits per Component

Type	Use	63-56	55-48	47-40	39-32	31-24	23-16	15-8	7-0
VL_PACKING_UYV_8_P	YCrCb, 8 bits per component packed into 24 bits (3 bytes) per pixel	V0	Y0	U0	V1	Y1	U1	V2	Y2
VL_PACKING_RGBA_8	RGBA	A0	B0	G0	R0	A1	B1	G1	R1
VL_PACKING_ABGR_8	ABGR	R0	G0	B0	A0	R1	G1	B1	A1
VL_PACKING_RGB_332_P	RGB; each 8-bit pixel, P*n*, is shown as BBGGGRRR	P7	P6	P5	P4	P3	P2	P1	P0
VL_PACKING_Y_8_P	Grayscale (luminance only)	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
VL_PACKING_RGB_8	RGB, 24-bit word, X*n* are ignored	X0	B0	G0	R0	X1	B1	G1	R1
VL_PACKING_BGR_8_P	RGB	R0	G0	B0	R1	G1	B1	R2	G2

Note: All these packings are native to the OCTANE Personal Video option, except VL_PACKING_RGB_332_P and VL_PACKING_Y_8_P, which are implemented automatically in software.

The OCTANE Personal Video option also supports VL_PACKING_YVYU_422_10, a YUV 4:2:2 packing type for ten bits per component. The ten data bits are left-aligned within a 16-bit word; Table 3-7 summarizes this packing. The hardware sets the lower six bits to zero before it writes them to memory. In reading from memory, it ignores the lower six bits.

Table 3-8 VL_PACKING_YVYU_422_10 Bits

Bits	Component
63-48	[U0]000000
47-32	[Y0]000000
31-16	[V0]000000
15-0	[Y1]000000

Appendix C, “Pixel Packings and Color Spaces,” shows the layout of each packing for the OCTANE Personal Video option. It also gives the corresponding names for these packings that are used by other libraries.

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. For example:

```
VLControlValue val;

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

VL_ZOOM

VL_ZOOM controls the scaling (decimation) of the video image, with values less than one performing the scaling. Figure 3-1 illustrates scaling.

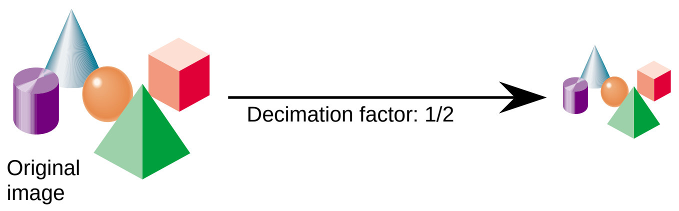


Figure 3-1 Zoom (Scaling)

This section explains

- “VL_ZOOM on Memory Nodes” on page 37
- “VL_ZOOM on Video Nodes” on page 38
- “VL_ZOOM on Screen Nodes” on page 39

VL_ZOOM on Memory Nodes

Applied to OCTANE Personal Video memory nodes, VL_ZOOM does not increase the size of an image, but decimates only. VL_ZOOM changes the x and y scale factors by the same amount. The effective scale size is determined by both VL_ZOOM and

VL_ASPECT: the horizontal scale factor is VL_ZOOM * VL_ASPECT, and the vertical scale factor is VL_ZOOM. An example using VL_ASPECT is given later in this section.

Besides the setting of VL_ASPECT, scaling with VL_ZOOM depends on the capture type and the packing. Table 3-9 summarizes the minimum values achievable for VL_ZOOM applied to memory source and drain nodes; for these minimum values, you must also use VL_ASPECT.

Table 3-9 VL_ZOOM on Memory Nodes: Minimum Values With VL_ASPECT

Minimum Pixel Number	VL_CAP_TYPE Value	VL_PACKING Value
4 x 1	VL_CAPTURE_NONINTERLEAVED	All but VL_PACKING_BGR_8_P or VL_PACKING_UYV_8_P
4 x 2	VL_CAPTURE_INTERLEAVED	All butVL_PACKING_BGR_8_P or VL_PACKING_UYV_8_P
24 x 1	VL_CAPTURE_NONINTERLEAVED	VL_PACKING_BGR_8_P or VL_PACKING_UYV_8_P only
24 x 2	VL_CAPTURE_INTERLEAVED	VL_PACKING_BGR_8_P or VL_PACKING_UYV_8_P only

VL_ZOOM takes a nonzero fraction as its argument; do not use negative values.

To correct the x scale, use VL_ASPECT. The following example sets the horizontal scale to 1/2 and the vertical scale to 1/1:

```
val.fractVal.numerator=1;
val.fractVal.denominator=1;
vlSetControl(svr, path, memnode, VL_ZOOM, &val);
     val.fractVal.numeriator=1;
     val.fractVal.denominator=2;
vlSetControl(svr, path, memnode, VL_ASPECT, &val);
```

VL_ZOOM on Video Nodes

On video source and drain nodes, you can set VL_ZOOM to unity (1/1) only, because OCTANE Personal Video device does not support scaling on a video node.

VL_ZOOM on Screen Nodes

On screen source nodes, if the VL_ZOOM value makes the resulting size invalid (that is, larger than a frame size), the size is constrained and a VLControlChanged event is generated. If the scaled size of the selected graphics region is smaller than the video frame size, use VL_OFFSET on the drain node to position the generated video.

Because the OCTANE Personal Video option can scale data coming from the screen source node, the minimum size (2 x 2) is the size after zooming. However, user-specified VL_SIZE numbers refer to the size before zooming. Thus, the minimum size the user can specify is the size that, when zoomed, yields 2 x 2. For example:

```
/*
 Set the (unzoomed) size to 20 pixels by 20 lines
*/
val.xyVal.x = 20;
val.xyVal.y = 20;
if (vlSetControl(server, screeen_path, screen_source_node, VL_SIZE,
&val)) {
    vlPerror(“Unable to set size”);
    exit(1);
}
/*
  Set the scale factor to 1/10. This results in a scaled size of
  20*1/10 by 20*1/10, or 2 pixels by 2 lines
*/
val.fractVal.numerator = 1;
val.fractVal.denominator = 10;
if (vlSetControl(server, screen_path, screen_source_node, VL_ZOOM,
&val)) {
    vlPerror(“Unable to set zoom”);
    exit(1);
}
```

This fragment causes the screen source node to send *xsize* × *ysize* video with as much scaling 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_source_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_source_node,
    VL_SIZE, &val))
{
   vlPerror("Unable to set size");
   exit(1);

}.
```

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 scaled to half size, the limits on the size control change by a factor of 2. Figure 3-2 illustrates clipping.

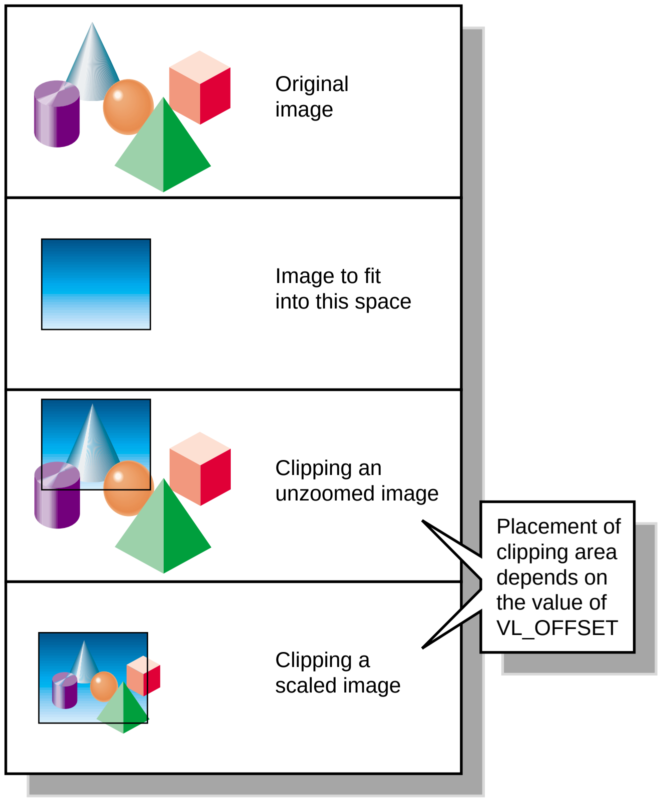

Original image
Image to fit into this space
Clipping an unzoomed image
Clipping a scaled image
Placement of clipping area depends on the value of VL_OFFSET

Figure 3-2 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);
```

On video source and drain nodes, VL_SIZE is 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

Note: Because this control 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.

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);
```

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

Figure 3-3 shows the controls that you can apply before and after you set offset, clipping, and scaling. Once the image is clipped, you can apply VL_ZOOM to scale it further.

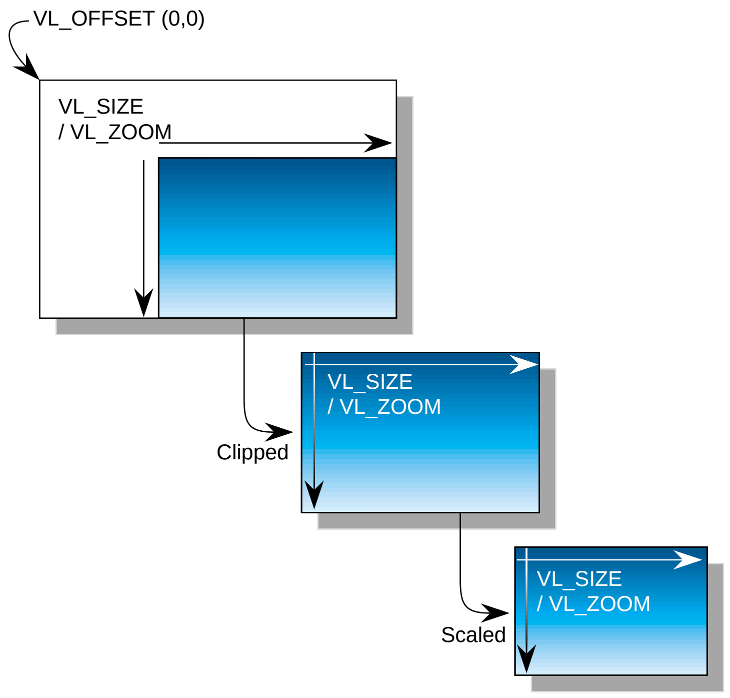


Figure 3-3 Zoom, Size, and Offset, and Origin

For memory nodes, VL_OFFSET and VL_SIZE in combination define the active region of video that is transferred to or from memory. On video source and drain nodes, VL_OFFSET can be set only to (0,0).

VL_CAP_TYPE and VL_RATE

An application can request that the OCTANE Personal 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 Personal 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; 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
- 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 3-10.

Table 3-10 VL_RATE Values (Items per Second)

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

Note: All rates are supported on all memory nodes. The buffer size must be set in accordance with the capture type.

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 to 60 fields per second, and for PAL 2 to 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 Personal 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 Personal 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_EVO_DOMINANCE_FIELD (see Table B-2 or Table B-3 in Appendix B for details). The OCTANE Personal 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 to 30 frames per second for NTSC and 1 to 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 Personal 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 Personal 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.

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 Personal 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 to 30 fields per second, and for PAL 1 to 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. These synchronization features are explained in Chapter 4, “Synchronizing Data Streams and Signals.”

The rate for this capture type is expressed in fields. For NTSC, the range is 1 to 60 fields per second, and for PAL 1 to 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, or oddness play no role in this capture type.

VL_COLORSPACE

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 on-board OCTANE Personal Video color-space conversion capability can perform many types of image-processing operations on a video path.

This section explains

- “Color Spaces” on page 48
- “Determining the Color Space” on page 49
- “Constant Hue” on page 50
- “Color-Space Converter for Image Processing” on page 52
- “Coefficients” on page 53

Note: For background information on color-space conversion, see Appendix D, “OCTANE Personal Video Color-Space Conversions,” later in this guide.

Color Spaces

Each component of an image has

- a color that it represents
- a canonical minimum value
- a canonical maximum value

Normally, a component stays within the minimum and maximum values. For example, for a luma signal such as Y, you can think of these limits as the black level and the peak white level, respectively. For a component with *n* bits, there are two possibilities for [minimum value, maximum value]:

- full range: [0, (2^{nbits})-1], which provides the maximum resolution for each component
- headroom range (compressed range):
  - Cr and Cb: [$(2^n) \div 16$, $15 \times (2^n) \div 16$]
  - Y, A, R, G, B: [$(2^n) \div 16$, $235 \times (2^n) \div 256$]

    This range is defined for 8 and 10 bits in ITU-R BT.601-4 (Rec. 601). For example, for 8-bit components: Cr and Cb: [16, 240]. Y, A, R, G, B: [16, 235]; for 10-bit components: Cr and Cb: [64, 960]. Y, A, R, G, and B: [64, 940].

  Headroom range provides numerical headroom, which is often useful when processing video images.

Two sets of colors are commonly used together, RGB (RGBA) and YCrCb/YUV (VYUA). YCrCb (YUV), the most common representation of color from the video world, represents each color by a luma component called Y and two components of chroma, called Cr (or V), and Cb (or U). The luma component is loosely related to brightness or luminance, and the chroma components make up a quantity loosely related to hue. These components are defined rigorously in ITU-R BT.601-4 (also known as Rec. 601 and CCIR 601).

The alpha channel is not a color. For that channel, the canonical minimum value means completely transparent, and the canonical maximum value means completely opaque.

For more information about color spaces, see *A Technical Introduction to Digital Video*, by Charles A. Poynton (New York: Wiley, 1996).

Determining the Color Space

For OpenGL, IRIS GL, and DM:

- the library constant indicates whether the data is RGBA or VYUA
- RGBA data is full-range by default

- VYUA data in DM can be full-range or headroom-range; you must determine this from context

In the VL_PACKING tokens from IRIX 6.2, the VL_PACKING constant indicates whether the data is RGBA or VYUA (as in VL_PACKING_UYV_8_P). The VL for the OCTANE Personal Video option (for IRIX 6.4) makes all of the parameters (packing, set of colors, range of components) explicit:

- Use VL_PACKING to specify only the memory layout. The new memory-only VL_PACKING tokens are disjoint from the old, and the old tokens are still honored, so this change is backward-compatible.
- Use VL_COLORSPACE to specify the color space parameters, as shown in Table C-11.

Table C-11 VL_COLORSPACE Options

Color Set	Full-Range Components	Headroom-Range (Compressed-Range) Components
RGBA	VL_COLORSPACE_RGB	VL_COLORSPACE_RP175
VYUA	VL_COLORSPACE_YUV	VL_COLORSPACE_CCIR601

The OCTANE Personal Video option performs color-space conversion if the color space implied by VL_FORMAT on the video node disagrees with that implied by VL_COLORSPACE. VL_COLORSPACE applies to memory source and drain nodes only.

Constant Hue

In addition to the standard color-space conversion model, the OCTANE Personal 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 lookup table (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. A particular control (VL_EVO_CSC_ALPHA_CORRECTION) 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_EVO_CSC_CONST_HUE: boolean control to enable (TRUE) or disable (FALSE) the constant-hue algorithm (memory source node only)
- VL_EVO_CSC_ALPHA_CORRECTION: boolean control to select the contents of the alpha channel (memory source and drain nodes only)

  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_EVO_CSC_ALPHA_CORRECTION has no effect if VL_EVO_CSC_CONST_HUE is disabled. When both VL_EVO_CSC_CONST_HUE and VL_EVO_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.

Color-Space Converter for Image Processing

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 D for a description of the color-space converter model and the relationships between the various internal processing blocks.

Table 3-12 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 3-12 Image-Processing Controls

Control	Default	Type	Use
VL_EVO_CSC_COEF	Multiplier operates in pass-through mode	extendedVal; data type EVO_CSC_COEF	Specifies the matrix multiplier coefficients
VL_EVO_CSC_LUT_IN_PAGE	0	intVal	Selects the active LUT
VL_EVO_CSC_LUT_ALPHA_PAGE	0	intVal	Selects the active LUT for the alpha channel
VL_EVO_CSC_LUT_IN_YG VL_EVO_CSC_LUT_IN_UB VL_EVO_CSC_LUT_IN_VR VL_EVO_CSC_LUT_ALPHA	Pass-through (1:1 mapping)	extendedVal; data type EVO_CSC_LUT_INPUT _AND_ALPHA	Specifies the contents of the input or alpha lookup tables
VL_EVO_CSC_LUT_OUT_YG VL_EVO_CSC_LUT_OUT_UB VL_EVO_CSC_LUT_OUT_VR	Pass-through (1:1 mapping)	extendedVal data type EVO_CSC_LUT_OUTPUT	Specifies the contents of the output lookup tables

Coefficients

The control VL_EVO_CSC_COEF 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 3-13 shows values.

Table 3-13 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 Personal 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 31st 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 Personal Video color-space feature provides for up to four input LUTs (each with YG, UB, and VR), and four alpha LUTs. Use the control VL_EVO_CSC_LUT_IN_PAGE or VL_EVO_CSC_LUT_ALPHA_PAGE in the application to select one of the four LUTs as active.

Using Input and Alpha LUTs

The controls for specifying the contents of the input or alpha lookup tables are VL_EVO_CSC_LUT_IN_YG, VL_EVO_CSC_LUT_IN_UB, VL_EVO_CSC_LUT_IN_VR, and VL_EVO_CSC_LUT_ALPHA.

The data pointer of the extended value points to a VL_EVOInAlphaLutValue structure, as defined in *dev_evo.h*. This structure contains the page number for the LUT being specified and a lookup table of 1024 integer entries (see VL_EVO_CSC_LUT_IN_PAGE and VL_EVO_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).

Camera Controls

When the O2Cam digital camera is selected—VL_VIDEO is set to VL_EVO_NODE_NUMBER_VIDEO_2—you can apply the O2Cam digital camera controls to set O2Cam features.

Most of these controls are accessible in the Pro menu of *vcp*: select Pro > Camera In > Coding Controls. For those controls accessible in *vcp*, the default value is Persistent, that is, the value set in *vcp*.

Table 3-14 summarizes O2Cam digital camera controls.

Table 3-14 O2Cam Digital Camera Controls

Control	Type	vcp Pro menu	Range	Use
VL_CAMERA_AUTO_GAIN _CONTROL	boolVal	Yes	0-1	Toggles automatic gain control on or off. When automatic gain control is on, the camera continually adjusts to changing lighting conditions to produce an even level of brightness. When automatic gain control is off, the user or program sets the proper gain level (VL_CAMERA_GAIN). If gain is too low, the picture appears dark; when gain is too high, it appears white or washed out. Factory default is 128.
VL_CAMERA_BLUE _BALANCE	intVal	Yes	0-255	Controls the proportion of blue in the image.
VL_CAMERA_BLUE _SATURATION	intVal	Yes	0-255	Sets overall blue color intensity in the image. A setting of 0 removes all blue from the image.
VL_CAMERA_BRIGHTNESS	intVal	No	0-255	Specifies the brightness level when VL_CAMERA_AUTO_GAIN_CONTROL is FALSE; read-only.
VL_CAMERA_BUTTON	boolVal	No	0-1	Indicates whether the button on the top of the camera is pressed or not; read-only.
VL_CAMERA_GAIN	intVal	Yes	0-255	Allows manual adjustment of camera exposure when automatic gain is off (VL_CAMERA_AUTO_GAIN_CONTROL is set to FALSE).
VL_CAMERA_GAMMA	intVal	Yes	0-255	Controls brightness of the dark areas in the image. Increasing the gamma value increases the brightness of dark areas.
VL_CAMERA_RATE	intVal	No	0-255	Specifies the frame rate of the camera; read-only.
VL_CAMERA_RED_BALANCE	intVal	Yes	0-255	Controls the proportion of red in the image.
VL_CAMERA_RED _SATURATION	intVal	Yes	0-255	Sets overall red color intensity in the image. A setting of 0 removes all red from the image.

Table 3-14 (continued) O2Cam Digital Camera Controls

Control	Type	vcp Pro menu	Range	Use
VL_CAMERA_SHUTTER	intVal	Yes	0-8	Controls shutter speed: faster shutter speed lets in less light and can be used when the amount of light is too high to be compensated for by the gain control (VL_CAMERA_GAIN). Values: VL_CAMERA_SHUTTER_60 VL_CAMERA_SHUTTER_100 VL_CAMERA_SHUTTER_125 VL_CAMERA_SHUTTER_250 VL_CAMERA_SHUTTER_500 VL_CAMERA_SHUTTER_1000 VL_CAMERA_SHUTTER_2000 VL_CAMERA_SHUTTER_4000 VL_CAMERA_SHUTTER_10000
VL_CAMERA_VENDOR_ID	stringVal	No	N/A	Returns the value VL_CAMERA_VENDOR_ID_SGI to identify the camera as a Silicon Graphics digital camera; read-only.
VL_CAMERA_VERSION	intVal	No	N/A	Returns the value VL_CAMERA_VERSION_SGI_DVC1 to identify the camera as the O2Cam digital camera; read-only.

Chapter 4

Synchronizing Data Streams and Signals

You can use special signals recognized or generated by the OCTANE Personal Video board—UST (unadjusted system time), MSC (media stream count)—to synchronize data streams, and use the board-generated Internal Video Sync signal to synchronize video and audio signals. This chapter explains

- "Using UST, MSC, and Buffered Media Streams for Synchronization" on page 57
- "Media Library Interfaces for UST and MSC" on page 59
- "Using the Internal Video Sync Signal" on page 61

Using UST, MSC, and Buffered Media Streams for Synchronization

Whenever a VL path is open in continuous mode, the OCTANE Personal 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 to 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, 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

The code below 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 or 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) 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 or 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.

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" on page 61
- "Setting the Internal Video Sync Signal Producer" on page 62

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 4-1 lists Silicon Graphics options that produce or consume the Internal Video Sync signal.

Table 4-1 Internal Video Sync Signal Producers and Consumers

Producer	**Consumer**
OCTANE Personal Video board	OCTANE Personal Video board
OCTANE Digital Video board	OCTANE Digital Video board
Digital Audio Option board	Digital Audio Option board
DIVO digital video option board for Origin2000™/Onyx2™	DIVO option board
OCTANE Compression board	
	InfiniteReality™ graphics
	OCTANE and Onyx2 built-in audio

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("Personal 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 5

Transferring Video Data and Ending Data Transfer

This chapter explains how to use buffers for data transfer, set execution times, and end data transfer, in these sections:

- "Transferring Video Data to and From Devices" on page 65
- "Ending Data Transfer" on page 79
- "Example Programs" on page 81

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 5-1 shows correspondences between VL buffer and DMbuffer API functions.

Table 5-1 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 Personal 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 8.
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()**
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_INTERLEAVED
- 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 5-2 shows the relationship between capture type and minimum VL buffer size.

Table 5-2 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 memory nodes, real-time memory or video transfer can be performed only as long as buffer sectors are available to the OCTANE Personal 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 5-3 lists the buffer-related calls.

Table 5-3 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 5-1 illustrates the difference between **vlGetNextValid()** and **vlGetLatestValid()**.

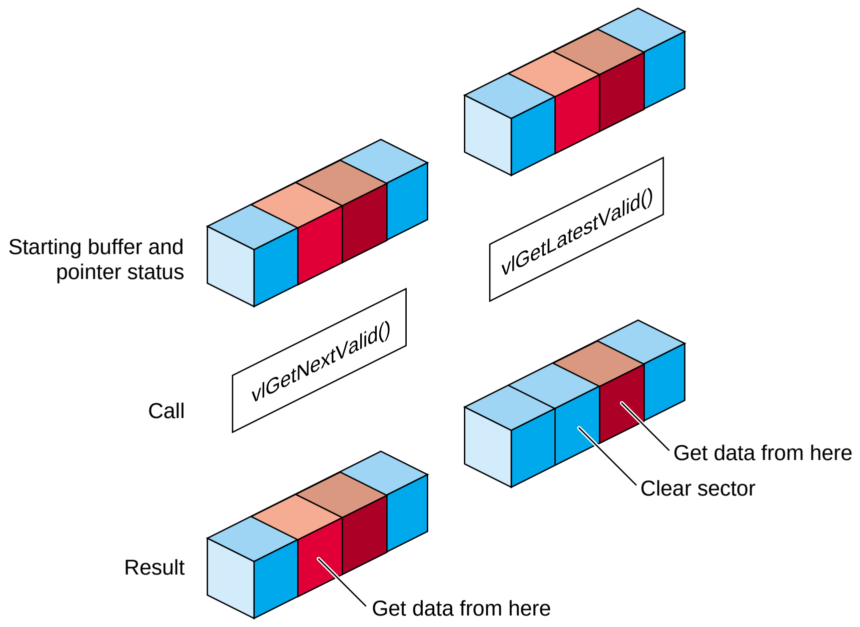


Figure 5-1 vlGetNextValid() and vlGetLatestValid()

Table 5-4 lists the calls that extract information from a buffer.

Table 5-4 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. Memory nodes 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 7 for a description of *eventex.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 output
- *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 6

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 Personal Video option are documented in the header file */usr/include/dmedia/vl_evo.h* (linked to */usr/include/vl/dev_evo.h*).

Note: For information on the controls used for specific nodes, see Appendix B, "OCTANE Personal Video Nodes and Their Controls." For detailed information on using controls such as VL_CAP_TYPE, VL_FORMAT, VL_TIMING, and so on, see Chapter 3, "Setting Parameters for Data Transfer.".

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. HUE, BRIGHTNESS */
    VLControlClass ctlClass; /* SLIDER, DETENT, KNOB, BUTTON */
    VLControlGroup group;    /* 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 unsigned int VLControlGroup;
```

The maximum length of a control or range name is VL_NAME_SIZE.

Table 6-1 summarizes the VL control groupings.

Table 6-1 VL Control Groupings

Grouping	Includes controls for...
VL_CTL_GROUP_VISUALQUALITY	Visual quality of sources or drains; for example, VL_H_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

Table 6-1 (continued) VL Control Groupings

Grouping	Includes controls for...
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_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_PRO	Specifying values not commonly found on the front panel of a real-world video device; for example, filter quality selections
VL_CTL_GROUP_MASK	Masking optional bits to extract only the control group

Chapter 7

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

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

OCTANE Personal Video VL Events

This section describes the events that the OCTANE Personal 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/dmedia/vl.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 screen or memory nodes or the framebuffer. A path receives a VLBadPath error from **vlSelectEvents()** if it attempts to register for events it cannot receive.

Table 7-1 summarizes the VL events for the OCTANE Personal Video device.

Table 7-1 VL Events for the OCTANE Personal Video Device

Event	Structure	Description
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."

Table 7-1 (continued) VL Events for the OCTANE Personal 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) 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.
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 have controls to abort a transfer when they detect invalid timing. In that case, a VLTransferFailed event is generated in its place.
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 memory nodes; this event may not be generated until a valid unit is transferred.

Table 7-1 (continued) VL Events for the OCTANE Personal Video Device

Event	Structure	Description
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.
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 or screen 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 or screen node to receive VLOddVerticalRetrace events.

Table 7-1 (continued) VL Events for the OCTANE Personal Video Device

Event	Structure	Description
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 or screen 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.
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.

Table 7-1 (continued) VL Events for the OCTANE Personal Video Device

Event	Structure	Description
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.

Querying VL Events

General VL event handling routines are summarized in Table 7-2.

Table 7-2 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

Table 7-2 (continued) VL Event Handling Routines

Routine	Use
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 7-1 illustrates the use of **vlAddCallback()**.

Example 7-1 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 8

Video Real-Time Capture and Playback

The OCTANE Personal Video 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"
- "Direct I/O to Disk"
- "syssgi"
- "Asynchronous I/O"

Video Library Buffers

Data transfer between the VL and an application takes place through a DMbuffer or VL buffer. When the OCTANE Personal 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 5.

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*`)`

`int vlBufferSetPoolDefaults(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.

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 Personal 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 5.

The VL_EVO_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_EVO_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_EVO_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.

Return Codes

This appendix explains the return codes that are used with the Video Library for the OCTANE Personal 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 to which the request was made.

`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 **vlSetControl()**, 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.

`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).

`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. Note that the OCTANE Personal Video device does not support these calls.

`VLPathInUse`

This error is generated by a call to (**vlSetupPaths()**) when a 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 with a control usage of 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.

OCTANE Personal Video Nodes and Their Controls

This appendix describes the nodes available to the OCTANE Personal Video option. It lists the 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. The *vcp* is the only supervisory application. Some controls 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 is indicated before the control table for the node; it 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 described in the following sections:

- "VL_DEVICE" on page 110
- "VL_MEM" on page 112
- "VL_SCREEN" on page 123
- "VL_VIDEO" on page 125

Note: In the tables that summarize controls for a node, VL device-independent controls (VL_) are listed first, in alphabetical order, followed by VL device-dependent controls (VL_EVO_). For information on camera controls, see "Camera Controls" on page 54 in Chapter 3.

VL_DEVICE

The device node (video source node) provides controls that affect the operation of the OCTANE Personal 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

Table B-1 lists device node controls. For VL_DEFAULT_DRAIN and VL_DEFAULT_SOURCE, access is GST; for all other controls, access is GS.

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.

Table B-1 (continued) Device Node Controls

Control	Default	Type	Use
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()**.
VL_SYNC	Persistent	intVal	The OCTANE Personal 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 Personal Video device. Parameters are GEN_PORT: genlock input on the board GEN_DVIN: serial digital input GEN_AVIN: composite input GEN_YCIN: Y/C input GEN_KSYNC: Internal Video Sync signal
VL_TIMING	Persistent	intVal	Selects the device timing for the OCTANE Personal Video device and affects the timing for the screen source node, the memory source nodes, and the video drain node. 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 nonsquare pixel timing VL_TIMING_625_SQ_PIX: PAL, 625-line square pixel timing VL_TIMING_625_CCIR601: CCIR 601, 625-line nonsquare pixel timing

VL_MEM

This discussion divides the VL_MEM nodes into their manifestations as source and drain.

VL_MEM Source

The OCTANE Personal Video option supports two memory source nodes, which provide real-time paths from main memory to the OCTANE Personal Video option. For the memory source node:

- type for both memory source nodes is VL_SRC
- kind for both memory source nodes is VL_MEM
- number is VL_EVO_NODE_NUMBER_MEM_1 and VL_EVO_NODE_NUMBER_MEM_2

Table B-2 lists memory source node controls. For all these controls, access is GS.

Table B-2 Memory Source Node Controls

Control	Default	Type	Use
VL_ASPECT	1/1	fractVal	Affects the horizontal scale factor. The effective scale factor is VL_ZOOM * VL_ASPECT.
VL_CAP_TYPE	VL_CAPTURE _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 VL_CAPTURE_FIELDS See “VL_CAP_TYPE and VL_RATE” in Chapter 3 for information on capture types.
VL_COLORSPACE	VL_COLORSPACE _CCIR601	intVal	Specifies color space of video data in memory: VL_COLORSPACE_RGB VL_COLORSPACE_CCIR601 VL_COLORSPACE_RP175 VL_COLORSPACE_YUV

Table B-2 (continued) Memory Source Node Controls

Control	Default	Type	Use
VL_FORMAT	VL_FORMAT_DIGITAL _COMPONENT_SERIAL	intVal	Specifies the type of video format to be produced: VL_FORMAT_DIGITAL_COMPONENT_SERIAL VL_FORMAT_DIGITAL_COMPONENT_RGB_SERIAL VL_FORMAT_SMPTE_YUV VL_FORMAT_RGB See "VL_FORMAT" in Chapter 3.
VL_PACKING	VL_PACKING_YVYU _422_8	intVal	Specifies the bit order in which the video components are stored in memory. See "VL_PACKING" in Chapter 3 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.
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 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.

Table B-2 (continued) Memory Source Node Controls

Control	Default	Type	Use
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. This control is applied before VL_ZOOM. It is recommended that VL_OFFSET be set before VL_SIZE.
VL_TIMING	Dynamic; from device node	intVal	Retrieves the current video timing value for the path. See “VL_SCREEN” and “VL_VIDEO” in this appendix for more details. This control is read-only.
VL_ZOOM	1.0	fractVal	Specifies the amount of scaling to be applied to the video before it is transferred to memory. See “VL_ZOOM” in Chapter 3.

Table B-2 (continued) Memory Source Node Controls

Control	Default	Type	Use
VL_FIELD _DOMINANCE	VL_F1_IS_DOMINANT	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_F1_IS_DOMINANT: 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_F2_IS_DOMINANT: 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_F1_IS_DOMINANT: The F1 field is in the first buffer of the pair, and the F2 field in the second. VL_F2_IS_DOMINANT: The F2 field is in the first buffer of the pair, the F1 field in the second.

Table B-2 (continued) Memory Source Node Controls

Control	Default	Type	Use
VL_EVO_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_EVO_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_EVO_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). See "Constant Hue" on page 50 in Chapter 3.
VL_EVO_CSC _COEF	Multiplier operates in pass-through mode	extendedVal	Specifies the matrix multiplier coefficients. See "Color-Space Converter for Image Processing" on page 52 in Chapter 3.
VL_EVO_CSC _CONST_HUE	TRUE	boolVal	Enables or disables constant-hue algorithm. See "Constant Hue" on page 50 in Chapter 3.
VL_EVO_CSC_LUT _IN_PAGE	0	intVal	Selects the active page for the input LUT. Valid page numbers are 0 through 3.
VL_EVO_CSC_LUT _ALPHA_PAGE	0		Selects the active page for the alpha LUT; valid page numbers are 0 through 3. See "Color-Space Converter for Image Processing" on page 52 in Chapter 3.
VL_EVO_CSC_LUT _IN_YG VL_EVO_CSC_LUT _IN_UB VL_EVO_CSC_LUT _IN_VR VL_EVO_CSC_LUT _ALPHA	Pass-through (1:1 mapping)	extendedVal	Specifies the contents of the input or alpha lookup tables. See "Color-Space Converter for Image Processing" on page 52 in Chapter 3.

Table B-2 (continued) Memory Source Node Controls

Control	Default	Type	Use
VL_EVO_CSC_LUT_OUT_YG VL_EVO_CSC_LUT_OUT_UB VL_EVO_CSC_LUT_OUT_VR	Pass-through (1:1 mapping)	extendedVal	Specifies the contents of the output lookup tables. See “Color-Space Converter for Image Processing” on page 52 in Chapter 3.
VL_EVO_DMA_ERROR_RESTART	VL_EVO_DMA_RESTART_ON	intVal	If enabled (VL_EVO_DMA_RESTART_ON), a video transfer continues when an error is encountered (the error is reported). Otherwise (VL_EVO_DMA_RESTART_OFF), the video transfer fails. 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_EVO_DMA_VOUT_BLANK	VL_EVO_DMA_VO_BLK_YUVA for YUV transfers; VL_EVO_DMA_VO_BLK_RGBA for RGB transfers	intVal	Sets output blanking, overriding the default output blanking set by VL_PACKING: VL_EVO_DMA_VO_BLK_YUVA: Y = A = 16, U = V = 128 VL_EVO_DMA_VO_BLK_RGBA: R = G = B = A = 16 Set this control after VL_PACKING.

Table B-2 (continued) Memory Source Node Controls

Control	Default	Type	Use
VL_EVO_DMA _VOUT _STARVATION	VL_EVO_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 three 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_EVO_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_EVO_DMA_VO_STARV_FLD: Causes the last field output to be repeated. VL_EVO_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 Galileo Video™ products, where a framebuffer is incorporated, the default value for this control is VL_EVO_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 Personal Video option supports two memory drain nodes, which provide real-time paths from the OCTANE Personal Video device to ring buffers. For the memory drain:

- type for both memory drain nodes is VL_DRN
- kind for both memory drain nodes is VL_MEM
- number is VL_EVO_NODE_NUMBER_MEM_1 and VL_EVO_NODE_NUMBER_MEM_2

Table B-3 lists memory drain node controls. For all these controls, access is GS.

Table B-3 Memory Drain Node Controls

Control	Default	Type	Use
VL_ASPECT	1/1	fractVal	Affects the horizontal scale factor. The effective scale factor is VL_ZOOM * VL_ASPECT.
VL_CAP_TYPE	VL_CAPTURE _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 “VL_CAP_TYPE and VL_RATE” in Chapter 3 for information on capture types.)
VL_COLORSPACE	VL_COLORSPACE _CCIR601	intVal	Specifies color space of video data in memory: VL_COLORSPACE_RGB VL_COLORSPACE_CCIR601 VL_COLORSPACE_RP175 VL_COLORSPACE_YUV
VL_FORMAT	Dynamic	intVal	Specifies the type of video format to be produced: VL_FORMAT_DIGITAL_COMPONENT_SERIAL VL_FORMAT_DIGITAL_COMPONENT_RGB_SERIAL VL_FORMAT_SMPTE_YUV VL_FORMAT_RGB See “VL_FORMAT” in Chapter 3.

Table B-3 (continued) Memory Drain Node Controls

Control	Default	Type	Use
VL_PACKING	VL_PACKING_YVYU _422_8	intVal	Specifies the bit order in which the video components are stored in memory. See “VL_PACKING” in Chapter 3 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.
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 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.

Table B-3 (continued) Memory Drain Node Controls

Control	Default	Type	Use
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; read only	intVal	Retrieves the current video timing value. See "VL_SCREEN" and "VL_VIDEO" in this appendix for more details. This control is read-only.
VL_ZOOM	1.0	fractVal	Specifies the amount of scaling to be applied to the video before it is transferred to memory. See "VL_ZOOM" in Chapter 3.
VL_EVO_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_EVO_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_EVO_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). See "Constant Hue" on page 50 in Chapter 3.
VL_EVO_CSC _COEF	Multiplier operates in pass-through mode	extended Val	Specifies the matrix multiplier coefficients. See "Color-Space Converter for Image Processing" on page 52 in Chapter 3.
VL_EVO_CSC _CONST_HUE	TRUE	boolVal	Enables or disables constant-hue algorithm. See "Constant Hue" on page 50 in Chapter 3.

Table B-3 (continued) Memory Drain Node Controls

Control	Default	Type	Use
VL_EVO_CSC_LUT_IN_PAGE	0	intVal	Selects the active page for the input LUT. Valid page numbers are 0 through 3.
VL_EVO_CSC_LUT_ALPHA_PAGE	0		Selects the active page for the alpha LUT; valid page numbers are 0 through 3. See "Color-Space Converter for Image Processing" on page 52 in Chapter 3.
VL_EVO_CSC_LUT_IN_YG VL_EVO_CSC_LUT_IN_UB VL_EVO_CSC_LUT_IN_VR VL_EVO_CSC_LUT_ALPHA	Pass-through (1:1 mapping)	extendedVal	Specifies the contents of the input or alpha lookup tables. See "Color-Space Converter for Image Processing" on page 52 in Chapter 3.
VL_EVO_CSC_LUT_OUT_YG VL_EVO_CSC_LUT_OUT_UB VL_EVO_CSC_LUT_OUT_VR	Pass-through (1:1 mapping)	extendedVal	Specifies the contents of the output lookup tables. See "Color-Space Converter for Image Processing" on page 52 in Chapter 3.
VL_EVO_DMA_ERROR_RESTART	VL_EVO_DMA_ERROR_RESTART_OFF	intVal	If enabled (VL_EVO_DMA_RESTART_ON), a video transfer continues when an error is encountered (the error is reported). Otherwise (VL_EVO_DMA_RESTART_OFF), the video transfer fails. 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_FIELD_DOMINANCE	VL_F1_IS_DOMINANT	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_FIELD_DOMINANCE in Table B-2 earlier in this appendix for more details.

Table B-3 (continued) Memory Drain Node Controls

Control	Default	Type	Use
VL_EVO_FILTER _QUALITY	Persistent	intVal	Determines the quality of the filter employed, ranked by filter quality: VL_EVO_FILTER_QUALITY_LO: pixel replication VL_EVO_FILTER_QUALITY_MED: linear interpolation VL_EVO_FILTER_QUALITY_HI: four-tap filter The CPU cost of the filters directly relates to their quality: the low-quality filter consumes about 25% of an R10000™ 195 MHz CPU; the high-quality filter consumes almost the entire CPU.
VL_EVO_FILTER _TYPE	Persistent	intVal	OCTANE Personal Video supports square and nonsquare video conversion to memory. This control selects how the conversion between the two formats is performed. The values are VL_EVO_FILTER_TYPE_FREQ (selects the frequency-preserving filter) VL_EVO_FILTER_TYPE_SPAT (selects the spatially preserving filter) See “VL_EVO_FILTER_TYPE” in Chapter 3.

VL_SCREEN

The OCTANE Personal Video option screen source node provides a means of using the graphics screen as a source of video data. This node allows for extracting an area larger than video size, which can be scaled to video size. The screen source node extracts pregamma 8-bit RGB pixel values and inserts the data into a video signal. The RGB pixel values can be

- captured to memory directly as RGB
- color-space converted and captured to memory
- sent to video out as YCbCr

For the screen node:

- type is VL_SRC
- kind is VL_SCREEN

Table B-4 lists screen source node controls. For all these controls, access is GST.

Table B-4 Screen Source Node Controls

Control	Default	Type	Use
VL_ASPECT	1/1	fractVal	Affects the horizontal scale factor. The effective scale factor is VL_ZOOM * VL_ASPECT.
VL_FLICKER _FILTER	TRUE	boolVal	Enables (TRUE) or disables flicker reduction. As the graphics display is progressive scanned, each field of the generated signal corresponds to one frame of the graphics display. Even and odd fields select different scan lines from the graphics frame, which can result in an undesirable flicker in the generated video if proper filtering is not applied.
VL_FREEZE	FALSE	boolVal	Freezes (TRUE) the image sent by the screen source node. If this control is set, graphics updates are not reflected in the generated video signal.
VL_OFFSET	(0, 0)	xyVal	Sets where the scaled images produced by the screen source node are inserted into a video frame, for centering or other placement of the images. For this node, only a value of (0,0) is valid.
VL_ORIGIN	(0, 0)	xyVal	Specifies the upper left corner (0,0) of a graphics display in unscaled screen coordinates. With VL_SIZE, this control selects a region of the graphics display to convert to video. If moving the origin causes the region to be clipped, VL_SIZE is updated and a VLValueChanged event is generated.
VL_SIZE	Dynamic	xyVal	Selects, with VL_ORIGIN, the height and width of the region of the graphics display that is converted to video. The value of VL_SIZE is specified in unscaled coordinates. Specify timing with VL_TIMING on the screen node. This control is applied before VL_ZOOM.

Table B-4 (continued) Screen Source Node Controls

Control	Default	Type	Use
VL_TIMING	Persistent	intVal	This control selects the device timing for the OCTANE Personal Video device and affects the timing for the screen source node, the memory source nodes, and the video drain node. 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 nonsquare pixel timing VL_TIMING_625_SQ_PIX: PAL, 625-line square pixel timing VL_TIMING_625_CCIR601: CCIR 601, 625-line nonsquare pixel timing
VL_ZOOM	1.0	fractVal	Sets the amount of scaling applied to the graphics area before it is converted to video. VL_ZOOM sets both the horizontal and vertical scale factors; use VL_ASPECT to modify the horizontal scale factor. See “VL_ZOOM” in Chapter 3. If the VL_ZOOM value makes the resulting size invalid (that is, larger than a frame size), the size is constrained and a VLControlChanged event is generated. If the scaled size of the selected graphics region is smaller than the video frame size, use VL_OFFSET on the drain node to position the generated video.

VL_VIDEO

This section discusses the VL_VIDEO source and drain nodes separately.

VL_VIDEO Source

The video source nodes correspond to the O2Cam connector converted to a digital input with appropriate third-party hardware, the O2Cam connector with the digital camera attached, and the analog connector.

For the video source:

- type for all three video source nodes is VL_SRC
- kind for all three video source nodes is VL_VIDEO

- number is
  - VL_EVO_NODE_NUMBER_VIDEO_1: digital video node, that is, the O2Cam connector used with a third-party serial converter
  - VL_EVO_NODE_NUMBER_VIDEO_2: O2Cam connector with O2Cam digital camera attached

    Note: Controls for the O2Cam digital camera are summarized in "Camera Controls" on page 54 in Chapter 3.
  - VL_EVO_NODE_NUMBER_VIDEO_3: analog video node; the VL_FORMAT control selects between composite, Y/C, or loopback

Table B-5 lists video source node controls. For all these controls, access is GST, except VL_FORMAT and VL_TIMING, which are GS.

Table B-5 Video Source Node Controls

Control	Default	Type	Use
VL_FORMAT	Persistent	intVal	Specifies the type of video format to be produced: VL_FORMAT_SVIDEO VL_FORMAT_COMPOSITE VL_FORMAT_CAMERA VL_EVO_FORMAT_LOOPBACK VL_FORMAT_DIGITAL_COMPONENT_SERIAL See "VL_FORMAT" in Chapter 3.
VL_FREEZE	FALSE	boolVal	Since the OCTANE Personal Video device does not support frozen inputs, this control can be set only to FALSE.
VL_OFFSET	(0, 0)	xyVal	Pans within the video. The OCTANE Personal Video source 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: 720 x 576 NTSC square pixel: 640 x 486 PAL square pixel: 768 x 576 Specify timing with VL_TIMING on the video source node.

Table B-5 (continued) Video Source Node Controls

Control	Default	Type	Use
VL_TIMING	Persistent	intVal	Specifies timing standard of incoming video signal: VL_TIMING_525_SQ_PIX: NTSC square pixel timing VL_TIMING_625_SQ_PIX: PAL square pixel timing; not applicable if video source is O2Cam digital camera (VL_EVO_NODE_NUMBER_VIDEO_2) VL_TIMING_525_CCIR601: CCIR 525-line nonsquare pixel timing VL_TIMING_625_CCIR601: CCIR 625-line nonsquare pixel timing; not applicable if video source is O2Cam digital camera (VL_EVO_NODE_NUMBER_VIDEO_2)
VL_ZOOM	1/1	fractVal	Sets scaling performed at video source. Since the OCTANE Personal Video device does not support scaling on a video node, this value can be set only to unity (1/1). Use memory nodes for scaling.
VL_EVO_VIN_AGC_UPDATE _INTERVAL	Persistent	intVal	Determines how often the automatic gain control circuit can change the gain. The update rate can be every line (VL_EVO_UPDT_INTVL_LINE) or every field (VL_EVO_UPDT_INTVL_FIELD).
VL_EVO_VIN_ANALOG _PROCESS	Persistent	intVal	Before analog-to-digital conversion, the analog signal can be amplified and anti-alias filtered. This control selects one of these actions: VL_EVO_INPUT_PROCESS_BYPASS: no action VL_EVO_INPUT_PROCESS_AMPLIFY: amplify only VL_EVO_INPUT_PROCESS_AMPLIFY_ALIAS _FILTER: amplify and anti-alias filter
VL_EVO_VIN_APERTURE _BAND_PASS	Persistent	intVal	You can peak the higher frequencies of the luminance signal by running the signal through a bandpass filter. The filtered signal is weighted by an aperture factor (see VL_EVO_VIN_APTERTURE_FACTOR) and added to the original unfiltered signal, boosting the frequencies passed by the filter. This control selects the center frequency of the bandpass filter. A value of 0 selects 4.1 MHz, 1 selects 3.8 MHz, 2 selects 2.6 MHz, and 3 selects 2.9 MHz.

Table B-5 (continued) Video Source Node Controls

Control	Default	Type	Use
VL_EVO_VIN_APERTURE _FACTOR	Persistent	intVal	Sets the aperture factor, which determines the contribution of a bandpass-filtered signal to the output luminance signal (see VL_EVO_VIN_APTERTURE_BANDPASS). A value of 0 selects 0.0 (no contribution), 1 selects 0.25, 2 selects 0.5, and 3 selects 1.0 (full contribution).
VL_EVO_VIN_CHROMA _BANDWIDTH	Persistent	intVal	Selects the bandwidth of the chrominance difference signals. Legal values are 0 (620 KHz), 1 (800 KHz), 2 (920 KHz) and 3 (1000 KHz). The color difference signals are low-pass filtered to achieve the selected bandwidth.
VL_EVO_VIN_CHROMA _HUE	Persistent	intVal	Affects the hue of the chrominance signal within a range of -180 degrees to 178.6 degrees.
VL_EVO_VIN_CHROMA _SATURATION	Persistent	fractVal	Determines the saturation of the chrominance signal with a gain of -2 (inverse chrominance) to 1.999. A value of 1.0 selects the CCIR level.
VL_EVO_VIN_COLOR _STANDARD	Persistent	intVal	Selects the color standard of the video input. The PAL and NTSC color standards vary by region: 0 selects PAL BGHI or NTSC M 1 selects NTSC 4.43 or PAL 4.43 2 selects PAL N or NTSC 4.43 3 selects NTSC N or PAL M
VL_EVO_VIN_FAST_COLOR _TIME_CNSTNT	Persistent	boolVal	Sets time constant: 0 selects nominal time constant, 1 selects fast time constant.
VL_EVO_VIN_GAIN_CH1 VL_EVO_VIN_GAIN_CH2	Persistent	fractVal	When AGC is not in use, these controls affect the gain of a composite signal (CH1), or the luminance (CH1) and chrominance (CH2) components of a Y/C signal. The gain can be adjusted between -5.98 dB to 5.98 dB.
VL_EVO_VIN_GAIN _CONTROL_FIX	Persistent	boolVal	Enables (TRUE) or disables the gain control circuit, which limits the gain at signal overshoots. TRUE enables the gain control circuit; FALSE disables it.
VL_EVO_VIN_GAIN_HOLD	Persistent	boolVal	Determines whether the automatic gain control (AGC) circuit is active or frozen (gain is held at a fixed value). TRUE freezes the gain; FALSE indicates active AGC.

Table B-5 (continued) Video Source Node Controls

Control	Default	Type	Use
VL_EVO_VIN_GAIN _HYSTERESIS	Persistent	intVal	When automatic gain control (AGC) is active, this control determines hysteresis of the gain circuit, or the amount that the computed gain must change before is actually used. Thus the AGC circuit need not constantly adjust to small variations in brightness. Set this control from 0 to 7, corresponding to 0 to 7 LSBs of a 9-bit gain value.
VL_EVO_VIN_HSYNC _STOP VL_EVO_VIN_HSYNC _START	Persistent	intVal	Determine the range in which a horizontal sync pulse is expected to be detected. Each control ranges from -107 to +108.
VL_EVO_VIN_LUMA _BRIGHTNESS	Persistent	intVal	Affects the brightness of the luminance signal. A value of 128 selects the CCIR luminance level, with 255 the brightest and 0 the darkest.
VL_EVO_VIN_LUMA _CONTRAST	Persistent	fractVal	Determines the contrast of the luminance signal. Contrast gain can be adjusted up to 1.999 and down to -2 (inverse luminance); 1.109 selects the CCIR level.
VL_EVO_VIN_LUMA _DELAY_COMPENSATE	Persistent	intVal	Delays the luminance components with respect to chrominance. The legal range is from -4 to 3 pixels.
VL_EVO_VIN_PREFILTER _ACTIVE	Persistent	boolVal	Enables (TRUE) or bypasses the prefilter, which emphasizes the high-frequency components of the luminance signal, compensating for loss.
VL_EVO_VIN_VERT _BLANK_SEL	Persistent	boolVal	Determines whether vertical blanking is long (0) or short (1).
VL_EVO_VIN_VERT_NOISE _REDUCT	Persistent	intVal	Determines how the video input responds to vertical noise in the signal: 0 selects normal mode, 1 selects searching mode, 2 selects free-running mode. A value of 4 bypasses the noise reduction circuit.
VL_EVO_VIN_TV_VTR_SEL	Persistent	boolVal	Affects how the video input locks (synchronizes) to a video signal: 0 selects TV mode (recommended only for poor-quality signals), 1 selects VTR mode.

Table B-5 (continued) Video Source Node Controls

Control	Default	Type	Use
VL_EVO_VIN_WHITE_PEAK	Persistent	boolVal	The white peak, or signal peak, control is part of the automatic gain control function; it limits the gain at signal overshoots. To limit the gain, set this control to TRUE; otherwise, set it to FALSE.

VL_VIDEO Drain

The OCTANE Personal Video option supports one video drain node that corresponds to the two analog video output connectors on the OCTANE Personal Video device, composite and S-Video (Y/C), and to the O2Cam connector, when an optional third-party converter provides SDI I/O at that port. This drain node drives all video outputs simultaneously.

For the video drain:

- type is VL_DRN
- kind is VL_VIDEO

Table B-6 lists video drain node controls. For all these controls, access is GST, except VL_FORMAT and VL_TIMING, which are GS.

Table B-6 Video Drain Node Controls

Control	Default	Type	Use
VL_DITHER_FILTER	Persistent	boolVal	Determines whether the square-to-nonsquare converter performs dithering (TRUE) at the video output.
VL_FORMAT	Persistent	intVal	Specifies the type of video format to be produced: VL_FORMAT_SVIDEO VL_FORMAT_COMPOSITE VL_FORMAT_DIGITAL_COMPONENT_SERIAL See “VL_FORMAT” in Chapter 3.
VL_FREEZE	TRUE	boolVal	Freezes (TRUE) the video sent to the video output connector.

Table B-6 (continued) Video Drain Node Controls

Control	Default	Type	Use
VL_H_PHASE	Persistent	fractVal	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 -3071 to 1023, which can advance the output by slightly more than three lines or delay the output by slightly more than one line. The default value 0 makes the output match the timing of the video input.
VL_OFFSET	(0, 0)	xyVal	Pans within the video. The OCTANE Personal Video drain node supports 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: 720 x 576 NTSC square pixel: 640 x 486 PAL square pixel: 768 x 576 Specify timing with VL_TIMING on the video drain node.
VL_TIMING	Persistent	intVal	Specifies timing standard of incoming video signal and affects the timing for the screen source node, the memory source nodes, and the video drain node. Because the analog video encoder operates in nonsquare mode only, square-to-nonsquare conversion is performed when VL_TIMING is set to one of the square pixel timings (that is, square pixel video is sent to the video out node). Values are as follows: VL_TIMING_525_SQ_PIX: NTSC square pixel timing VL_TIMING_625_SQ_PIX: PAL square pixel timing VL_TIMING_525_CCIR601: CCIR 525-line nonsquare pixel timing VL_TIMING_625_CCIR601: CCIR 625-line nonsquare pixel timing
VL_ZOOM	1/1	fractVal	Sets scaling performed at video drain. Since the OCTANE Personal Video device does not support scaling on a video node, this value can be set only to unity (1/1). Use memory nodes for scaling.

Table B-6 (continued) Video Drain Node Controls

Control	Default	Type	Use
VL_EVO_ALPHA_NOT _PIXEL	FALSE	boolVal	If the node supplying the video drain node has both pixel and alpha outputs (for example, RGBA or YUVA 4:2:2:4/4:4:4:4 sent from memory), this control selects whether alpha (1: A) or pixel (0: YUV/RGB) is sent to the video output.
VL_EVO_COLOR_BAR _ENABLE	FALSE	boolVal	Enables (1) or disables (0) automatic color bar generation on the video output, such as for test purposes. When enabled, color bars supersede any other video sent to the video output.
VL_EVO_COLOR_FRAME _LOCK	Persistent	boolVal	Enables (TRUE) or disables (FALSE) color-frame locking. The video output of Personal Video can be color-frame locked to either the reference input or to composite analog input. For frame locking to be enabled, genlock (VL_SYNC) must be enabled and either the reference input or the composite analog input must be selected as the genlock source (VL_SYNC_SOURCE).
VL_EVO_COLOR_HPHASE	Persistent	intVal	Adjusts the color subcarrier phase with respect to horizontal sync from 0 to 360 degrees in 360/256 degree steps. Normally, the color subcarrier is in phase with the horizontal sync (0 degree separation).
VL_EVO_CSC_COEF	Multiplier operates in pass-through mode	extended Val	Specifies the matrix multiplier coefficients. See "Color-Space Converter for Image Processing" on page 52 in Chapter 3.
VL_EVO_CSC_LUT_IN _PAGE	0	intVal	Selects the active page for the input LUT. Valid page numbers are 0 through 3.
VL_EVO_CSC_LUT_ALPHA _PAGE	0		Selects the active page for the alpha LUT; valid page numbers are 0 through 3. See "Color-Space Converter for Image Processing" on page 52 in Chapter 3.
VL_EVO_CSC_LUT_IN_YG VL_EVO_CSC_LUT_IN_UB VL_EVO_CSC_LUT_IN_VR VL_EVO_CSC_LUT_ALPHA	Pass-through (1:1 mapping)	extended Val	Specifies the contents of the input or alpha lookup tables. See "Color-Space Converter for Image Processing" on page 52 in Chapter 3.

Table B-6 (continued) Video Drain Node Controls

Control	Default	Type	Use
VL_EVO_CSC_LUT_OUT_YG VL_EVO_CSC_LUT_OUT_UB VL_EVO_CSC_LUT_OUT_VR	Pass-through (1:1 mapping)	extendedVal	Specifies the contents of the output lookup tables. See "Color-Space Converter for Image Processing" on page 52 in Chapter 3.
VL_EVO_FILTER_TYPE	Persistent	intVal	OCTANE Personal Video supports square and nonsquare conversions at the video drain. This control selects how the conversion between the two formats is performed. The values are VL_EVO_FILTER_TYPE_FREQ (selects the frequency-preserving filter) VL_EVO_FILTER_TYPE_SPAT (selects the spatially preserving filter) See "VL_EVO_FILTER_TYPE" in Chapter 3.
VL_EVO_VOUT_BLACK_LEVEL	Persistent	fractVal	Sets black level of video signal with respect to the sync level. For a white-to-sync range of 140 IRE (NTSC), the recommended black level is 47.5 IRE with respect to the sync level. The black level can be set between 24 IRE and 49 IRE in steps of 25/63 IRE. For a white-to-sync range of 143 IRE (PAL), the recommended black level is 43 IRE with respect to sync. The actual black level can range between 24 IRE and 50 IRE in steps of 26/63 IRE. Some video standards define a pedestal, or offset, of the black level to separate active video from the blanking level. NTSC does so; PAL and SECAM do not.
VL_EVO_VOUT_CAP_1ST_BYTE_ODD VL_EVO_VOUT_CAP_2ND_BYTE_ODD	None	intVal	If closed caption encoding is enabled, these controls determine the text of the closed captioning for the odd and even fields. Each field can encode two bytes of closed caption data.
VL_EVO_VOUT_COLOR_BURST_AMP	Dynamic	fractVal	Sets colorburst amplitude with respect to its nominal value: for NTSC, the nominal burst amplitude is 40 IRE (adjustable from 0 to 1.25x nominal); for PAL, 43 IRE (adjustable from 0 to 1.67x nominal).

Table B-6 (continued) Video Drain Node Controls

Control	Default	Type	Use
VL_EVO_VOUT_CC _ENCODING	VL_EVO _ENCODING_OFF	intVal	Enables or disables (0) closed caption encoding. A value of 1 enables encoding in field 1 (odd fields); 2 enables encoding of field 2 (even fields); 3 enables encoding in both fields. Use VL_EVO_VOUT_CAP* and VL_EVO_VOUT_XTN* to specify the content of the captioning in each field.
VL_EVO_VOUT_ FIRST _ACTIVE_LN	0	fractVal	Specifies the first line of active video in each field. Lines outside of the active video are blanked. A value of 0 represents line 17 in NTSC and line 22 in PAL.
VL_EVO_VOUT_LUMA_WT _GN_92_5	Persistent	boolVal	Determines whether the white-to-black range for luminance is 92.5 IRE (TRUE) or 100 IRE (FALSE). The 92.5 IRE setting typically includes a 7.5 IRE setup (pedestal) of black. See also VL_EVO_VOUT_BLACK_LEVEL.
VL_EVO_VOUT_U_GAIN VL_EVO_VOUT_V_GAIN	Persistent	fractVal	Set the amount of gain applied to the U and V color difference signals. The U gain ranges from -2.17x to 2.16x for NTSC and -2.05x to 2.04x for PAL. The V gain ranges from -1.55x to 1.55x for NTSC and -1.46 to 1.46x for PAL.
VL_EVO_VOUT_XTN_1ST _BYTE_EVN VL_EVO_VOUT_XTN_2ND _BYTE_EVN	None	intVal	If closed caption encoding is enabled, these controls determine the text of the closed captioning for the odd and even fields. Each field can encode two bytes of closed-caption data.

Appendix C

Pixel Packings and Color Spaces

This appendix explains

Packings

This section presents each packing used by the OCTANE Personal Video option, giving a diagram and its tokens in the pertinent libraries. It explains

Packings and Color Spaces

A packing

- determines which of the four components are sampled, either RGBA or VYUA (more correctly, CrYCbA)
- determines the sampling pattern (for example, 4:4:4 or 4:2:2), which specifies where and how often each component of the image is sampled
- allocates a certain number of bits to represent the component samples, and positions those samples along with possible padding in memory

  Each sample is an unsigned number.

A color space

- determines the color in each component by specifying the color set
- specifies a canonical minimum and maximum value for each component, either full-range or headroom-range

  See "Color Spaces" on page 48 in Chapter 3 for an explanation.

In most Silicon Graphics libraries, a single token encodes both color space and packing. For example, VL_PACKING_RGBA_8 is a 32-bit packing in the RGBA color space. In the VL of the OCTANE Personal Video option and other advanced products, the two parameters are specified separately with different controls: VL_PACKING and VL_COLORSPACE. The color space must be defined with the VL_COLORSPACE control.

Packing Diagram Conventions

In all illustrations, as you move from left to right:

- Each byte goes from the most significant bit to the least significant bit.
- The bytes increase in memory address by 1.
- Component samples go from most significant bit to least significant bit.

Each illustration shows the smallest repeating spatial pattern of component samples that is a multiple of 8 bits wide. No additional padding or alignment is to be inferred. For example, a 24-bit-per-pixel diagram, such as that for VL_PACKING_BGR_8_P and VL_PACKING_UYV_8_P, indicates 3-byte quantities packed together in memory. The values are not padded out to 32-bit boundaries; see Figure C-1.

Pixel 1		
Byte 1	Byte 2	Byte 3
r r r r r r r r	g g g g g g g g	b b b b b b b b
v v v v v v v v	y y y y y y y y	u u u u u u u u

Figure C-1 VL_PACKING_BGR_8_P and VL_PACKING_UYV_8_P

An x ("don't care") in a bit means

- Readers may get any garbage in the bit.
- Writers may leave the bit as garbage.

A 0 means

- Readers may assume the bit is zero.
- Writers must zero out the bit.

  Note: Writers in a memory-to-video VL path may leave the bit as garbage.

The packing defines a bit layout, but for convenience, as shown in Figure C-1, the component slots are filled with the RGBA or VYUA color set where appropriate. See "Color Spaces" on page 48 in Chapter 3 for an explanation.

Note: For chroma components, Cr and Cb are more accurate terms than V and U, because the analog NTSC video specification ANSI/SMPTE 170M uses V and U with a slightly different meaning. However, this chapter uses the letters V and U in the illustrations of packings for typographical convenience.

Packings that use 4:2:2 sampling also show the location of each component sample: left and right for 4:2:2. The diagrams assume row-major, left-to-right ordering of pixels in memory.

The OCTANE Personal Video device can natively transfer data of all the packings shown in this appendix in real time except VL_PACKING_Y_8_P and VL_PACKING_RGB_332_P.

Packings and Library Tokens

Accompanying each packing diagram are comments and library tokens for that packing. For most packings, two indications are given for VL:

- The main reference uses the old-style token, for example, VL_PACKING_Y_8_P. These packings encode both the bit layout (packing) and color space.
- New-style VL tokens are included for reference. The indication includes the new-style packing control value and a color-space control value; for example, VL_PACKING_4_8 + VL_COLORSPACE_{CCIR,YUV}.

For the OCTANE Personal Video option, you set packing and color space separately for memory nodes. The new definitions provide a more flexible way to specify memory layout of pixels and their color spaces.

DM refers to the tokens in */usr/lib/dmedia/dm_image.h*, which are used by several libraries (*libdmedia* (dmParams, dmIC, dmColor), *libmoviefile*, *libmovieplay*, and others). See “Color Spaces” on page 48 in Chapter 3 for an explanation.

Packing Naming Conventions (New-Style Tokens)

In packing tokens, the following applies:

- _L and _R appended to the end of tokens with padding (0 bits) indicate that the 0 bits are at the left end or the right end of the pattern, respectively; for example, VL_PACKING_4444_10_in_16_L and VL_PACKING_4444_10_in_16_R.
- R before the numerical part of the token indicates reverse order of the components; for example, VL_PACKING_242_8 and VL_PACKING_R242_8 have the same pattern of component bits, but the order is reversed in VL_PACKING_R242_8.
- Z at the end of the token name means that the packing is padded to the word boundary; for example, the packing in VL_PACKING_2424_10_10_10_2Z is 30 bits per pixel, but it is padded to 32 bits per pixel.

Table C-1 lists the OCTANE Personal Video packings in the order of the number of bits in the pattern of component samples—the order in which they are described in the rest of this section.

Table C-1 OCTANE Personal Video Packings

Old-Style Packing and Color-Space Token	Bits	Color Space	New-Style Packing Name
VL_PACKING_Y_8_P	8	VYUA monochrome/luma only	VL_PACKING_4_8
VL_PACKING_RGB_332_P	8	RGBA	VL_PACKING_R444_332
VL_PACKING_YVYU_422_8	16	VYUA	VL_PACKING_R242_8
VL_PACKING_BGR_8_P	24	RGBA	VL_PACKING_444_8
VL_PACKING_UYV_8_P	24	VYUA	VL_PACKING_444_8
VL_PACKING_ABGR_8	32	RGBA	VL_PACKING_4444_8
VL_PACKING_AUYV_8 or VL_PACKING_AUYV_4444_8	32	VYUA	VL_PACKING_4444_8
VL_PACKING_RGBA_8	32	RGBA	VL_PACKING_R4444_8
VL_PACKING_YUVA_4444_8	32	VYUA	VL_PACKING_R4444_8
VL_PACKING_RGB_8	32	RGBA	VL_PACKING_R0444_8
VL_PACKING_YVYU_422_10	32	VYUA	VL_PACKING_R242_10_in_16_L

8-Bit Pixel Packings

Figure C-2 shows the VL_PACKING_Y_8_P, an 8-bit packing useful for VYUA monochrome (luma) only.

Pixel 1
Byte 1
y y y y y y y y

Figure C-2 VL_PACKING_Y_8_P

This packing is

- VL_PACKING_4_8 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style
- GL_LUMINANCE GL_UNSIGNED_BYTE in OpenGL
- DM_IMAGE_PACKING_LUMINANCE in DM

This packing is not native to the OCTANE Personal Video option, but is implemented in software.

Figure C-3 shows VL_PACKING_RGB_332_P, an 8-bit packing in the RGBA color space.

Pixel 1
Byte 1
b b g g g r r r

Figure C-3 VL_PACKING_RGB_332_P

This packing is

- VL_PACKING_R444_332 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
- VL_PACKING_RGB_332_P in the VL, old style
- DM_IMAGE_PACKING_BGR233 in DM

This packing is not native to the OCTANE Personal Video option, but is implemented in software.

16-Bit Pixel Packing

Figure C-4 shows VL_PACKING_YVYU_422_8, a 16-bit 4:2:2 VYUA packing. The most commonly used 4:2:2 packing, it is used by other Silicon Graphics video hardware as well as the OCTANE Personal Video option.

Pixels 1-2			
Byte 1	Byte 2	Byte 3	Byte 4
u u u u u u u u	y y y y y y y y	v v v v v v v v	y y y y y y y y
left			right

Figure C-4 VL_PACKING_YVYU_422_8

This packing is

- VL_PACKING_R242_8 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style
- GL_YCRCB_422_SGIX GL_UNSIGNED_BYTE in OpenGL
- DM_IMAGE_PACKING_CbYCrY in DM

24-Bit Pixel Packings

Figure C-5 shows VL_PACKING_BGR_8_P and VL_PACKING_UYV_8_P, which are 24-bit RGBA/VYUA packings.

Pixel 1		
Byte 1	Byte 2	Byte 3
r r r r r r r r	g g g g g g g g	b b b b b b b b
v v v v v v v v	y y y y y y y y	u u u u u u u u

Figure C-5 VL_PACKING_BGR_8_P and VL_PACKING_UYV_8_P

VL_PACKING_BGR_8_P is

- GL_RGB GL_UNSIGNED_BYTE in OpenGL
- VL_PACKING_444_8 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
- DM_IMAGE_PACKING_RGB in DM

VL_PACKING_UYV_8_P is VL_PACKING_444_8 + VL_COLORSPACE_{RGB,RP175} in the VL, new style.

32-Bit Pixel Packings

Figure C-6 shows VL_PACKING_ABGR_8 and VL_PACKING_AUYV_8. These packings are supported by many Silicon Graphics video products.

Pixel 1			
Byte 1	Byte 2	Byte 3	Byte 4
r r r r r r r r	g g g g g g g g	b b b b b b b b	a a a a a a a a
v v v v v v v v	y y y y y y y y	u u u u u u u u	a a a a a a a a

Figure C-6 VL_PACKING_ABGR_8 and VL_PACKING_AUYV_8

VL_PACKING_ABGR_8 is

- VL_PACKING_4444_8 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
- GL_RGBA GL_UNSIGNED_BYTE in OpenGL (most commonly used OpenGL packing)
- DM_IMAGE_PACKING_RGBA in DM

VL_PACKING_AUYV_8 is

- VL_PACKING_4444_8 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style
- also VL_PACKING_AUYV_4444_8 in the VL, old style

Figure C-7 shows VL_PACKING_RGBA_8 and VL_PACKING_YUVA_4444_8, which are supported by many Silicon Graphics video products. VL_PACKING_RGBA_8 is the default IRIS GL packing.

Pixel 1			
Byte 1	Byte 2	Byte 3	Byte 4
a a a a a a a a	b b b b b b b b	g g g g g g g g	r r r r r r r r
a a a a a a a a	u u u u u u u u	y y y y y y y y	v v v v v v v v

Figure C-7 VL_PACKING_RGBA_8 and VL_PACKING_YUVA_4444_8

VL_PACKING_RGBA_8 is

- VL_PACKING_R4444_8 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
- PM_ABGR PM_UNSIGNED_BYTE in IRIS GL (the default)
- GL_ABGR_EXT GL_UNSIGNED_BYTE in OpenGL
- DM_IMAGE_PACKING_ABGR in DM

VL_PACKING_YUVA_4444_8 is VL_PACKING_R4444_8 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style.

Figure C-8 shows VL_PACKING_RGB_8, an IRIS GL-like 32-bit packing. This packing is supported by many Silicon Graphics video products.

Pixel 1			
Byte 1	Byte 2	Byte 3	Byte 4
x x x x x x x x	b b b b b b b b	g g g g g g g g	r r r r r r r r
x x x x x x x x	u u u u u u u u	y y y y y y y y	v v v v v v v v

Figure C-8 VL_PACKING_RGB_8

VL_PACKING_RGB_8 is

- VL_PACKING_R0444_8 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
- DM_IMAGE_PACKING_XBGR

  Use DM_IMAGE_PACKING_ABGR instead of this packing unless you specifically want to inform a piece of software (such as dmColor) not to spend processing time on the alpha channel.

Figure C-9 shows VL_PACKING_YVYU_422_10, a 4:2:2 10_in_16 32-bit VYUA packing. This packing is supported by several recent Silicon Graphics video products.

Pixels 1-2							
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
u u u u u u u u	u u 0 0 0 0 0 0	y y y y y y y y	y y 0 0 0 0 0 0	v v v v v v v v	v v 0 0 0 0 0 0	y y y y y y y y	y y 0 0 0 0 0
left	0 0 0 0 0 0	left	0 0 0 0 0 0	left	0 0 0 0 0 0	right	0 0 0 0 0

Figure C-9 VL_PACKING_YVYU_422_10

This packing is

- 4:2:2 sampling (2 bits of A); see “Sampling Patterns,” later in this appendix
- VL_PACKING_R242_10_in_16_L + VL_COLORSPACE_{CCIR,YUV} in the VL, new style

Sampling Patterns

Sampling patterns are

- "4:4:4 and 4:4:4:4 Sampling" on page 146
- "4:2:2 and 4:2:2:4 Sampling" on page 147

4:4:4 and 4:4:4:4 Sampling

Some of the packing diagrams earlier in this appendix indicate 4:4:4 or 4:4:4:4 sampling. This video industry terminology means that each of the three or four components is sampled at every pixel. Figure C-11 diagrams this sampling pattern.

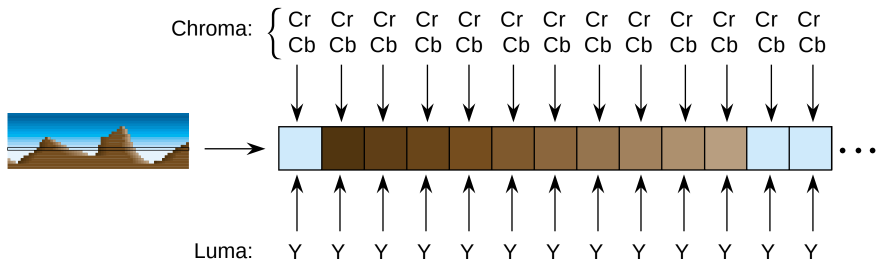

Chroma:
Cr Cb Cr Cb Cr Cb Cr Cb Cr Cb Cr Cb Cr Cb Cr Cb Cr Cb Cr Cb Cr Cb Cr Cb Cr Cb
Luma:
Y Y Y Y Y Y Y Y Y Y Y Y Y

Figure C-10 4:4:4 Sampling

4:2:2 and 4:2:2:4 Sampling

The packings shown in diagrams that indicate 4:2:2 sampling make sense only in the VYUA color spaces. For every two pixels, there are two luma samples (two Y's) but only one chroma sample (one sample of Cr and Cb, which together determine the chroma), as shown in Figure C-11.

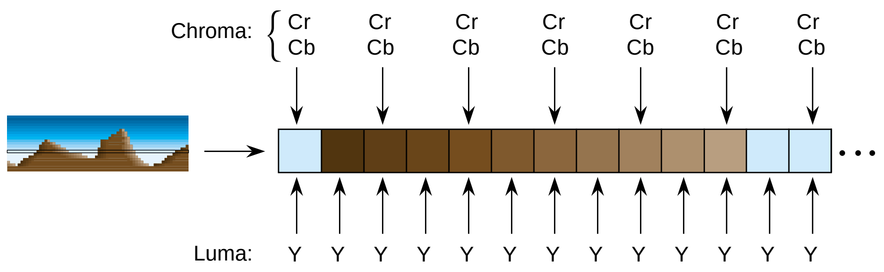


Figure C-11 4:2:2 Sampling

The chroma samples belong at the same instant in space as the left Y sample (the chrominance samples and the left Y are co-sited). The diagrams for 4:2:2 packings in the "Packings" section of this appendix *s*how the spatial location of each Y, Cr, or Cb component as left or right. The first pixel of each line is a left pixel.

Converting 4:4:4 video to 4:2:2 video is like converting 44.1 kHz audio into 22.05 kHz audio: just dropping every other Cr,Cb sample yields extremely poor results. Video devices that need to convert between 4:4:4 and 4:2:2 use carefully designed filters. The characteristics of the required filter are specified in ITU-R BT.601-4 (Rec. 601).

4:2:2 sampled packings that also include alpha are called 4:2:2:4. This method has one alpha value per pixel, like the Y value.

Appendix D

OCTANE Personal Video Color-Space Conversions

The OCTANE Personal 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 Personal Video option can translate between YUV and RGB with high accuracy in real time.

Understanding the capabilities of the OCTANE Personal 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 Personal Video Color Spaces

The OCTANE Personal 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 10-bit values are clamped to the range [4..1019]. Black has the CrYCb value [512,64,512]. White has the value [512,940,512]. For 8 bits, the values are clamped to the range [1..254]; black has the CrYCb value [128,16,128], and white has the value [128,235,128].

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 Personal 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 Personal 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 Personal Video Option

The OCTANE Personal 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 D-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.

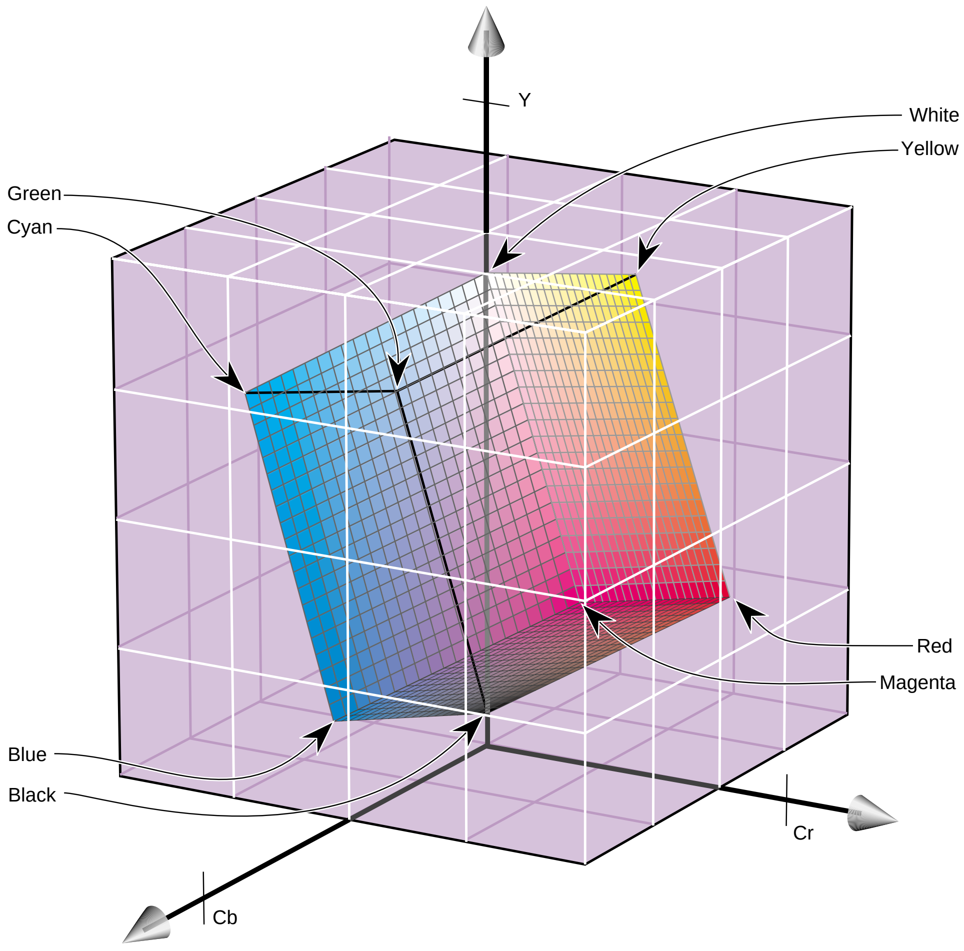


Figure D-1 RGB Cube in CCIR Space

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.

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 D-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.

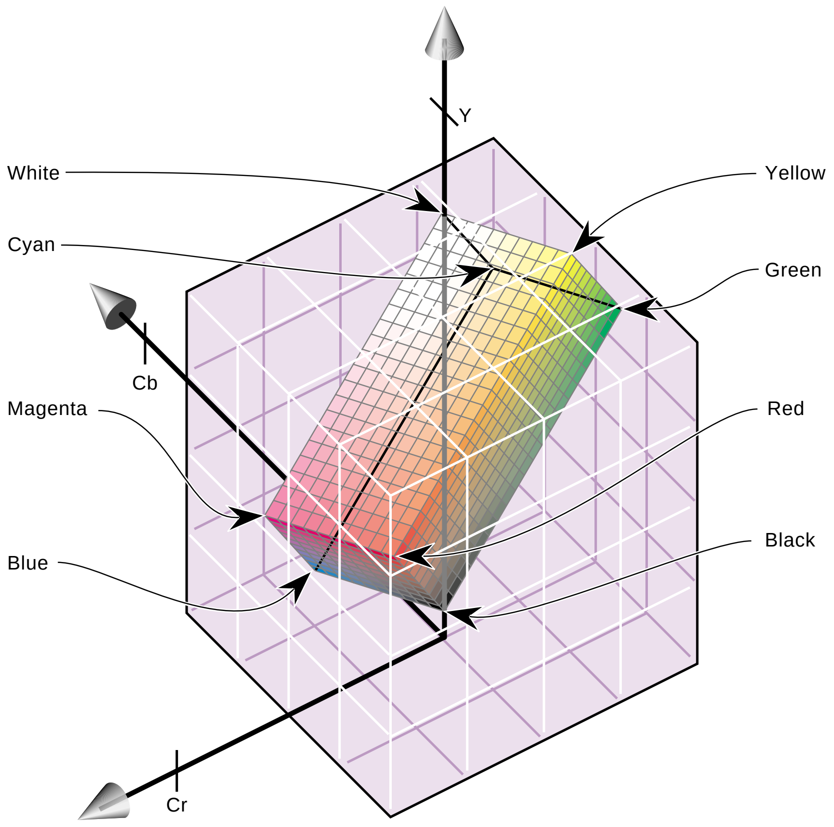


Figure D-2 Color Cube With Luminance/Chrominance Ramp Vector

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 Personal Video option, passed it through a 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 Personal 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 D-3 shows the red and Cr components, Figure D-4 the green and Y components, and Figure D-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.

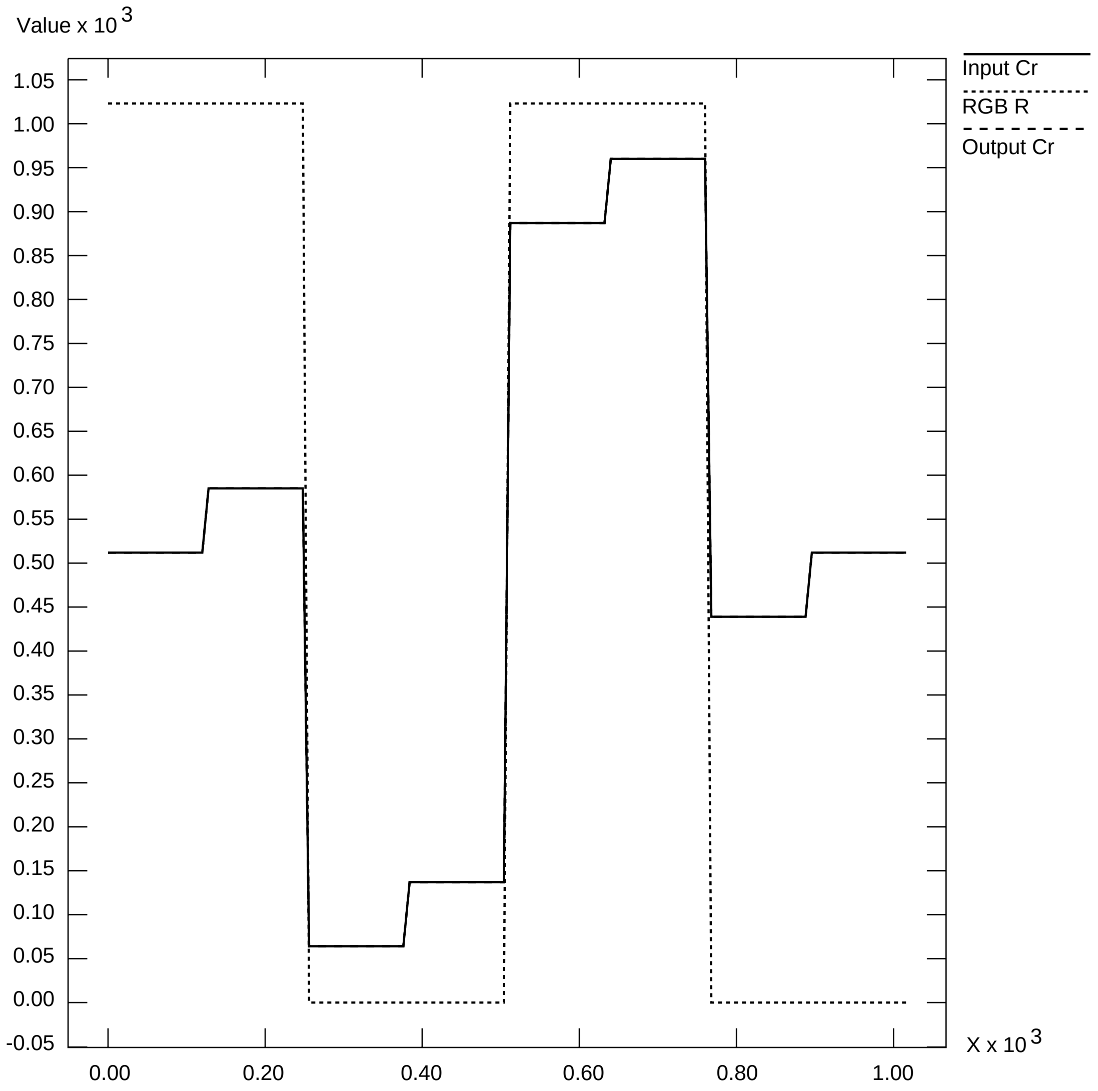


Figure D-3 100% Color Bars: Cr/R

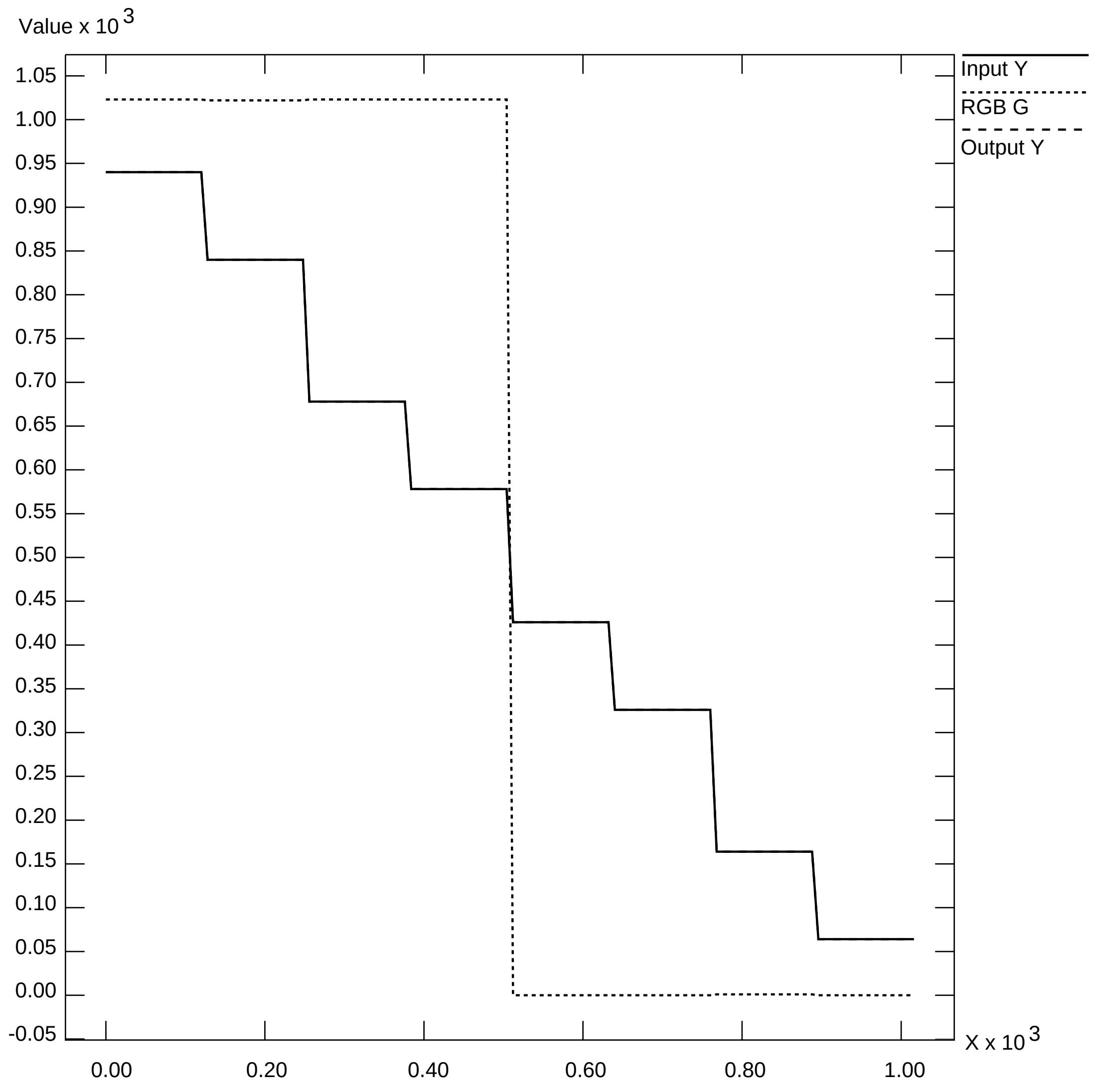


Figure D-4 100% Color Bars: Y/G

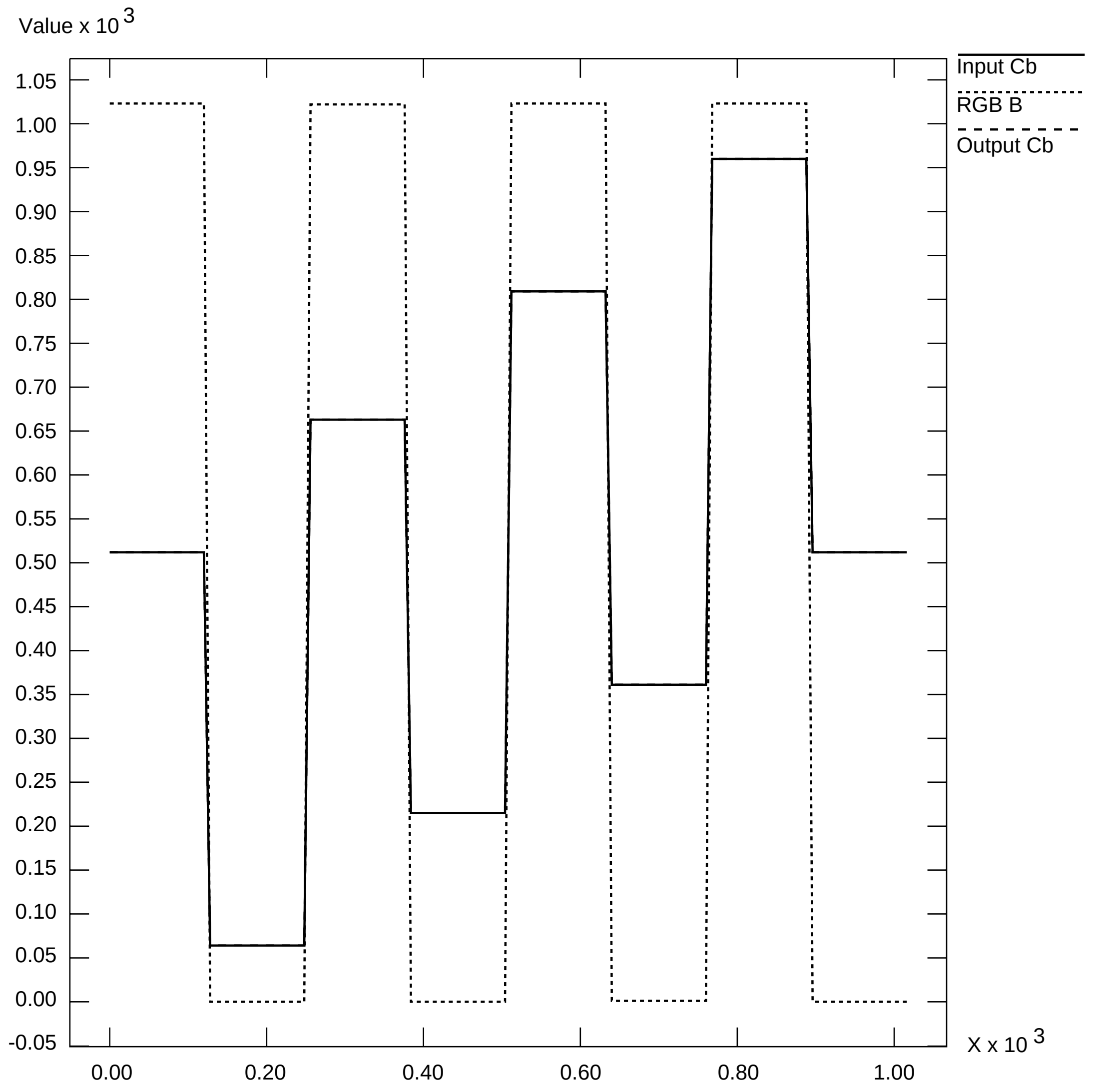


Figure D-5 100% Color Bars: Cb/B

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.

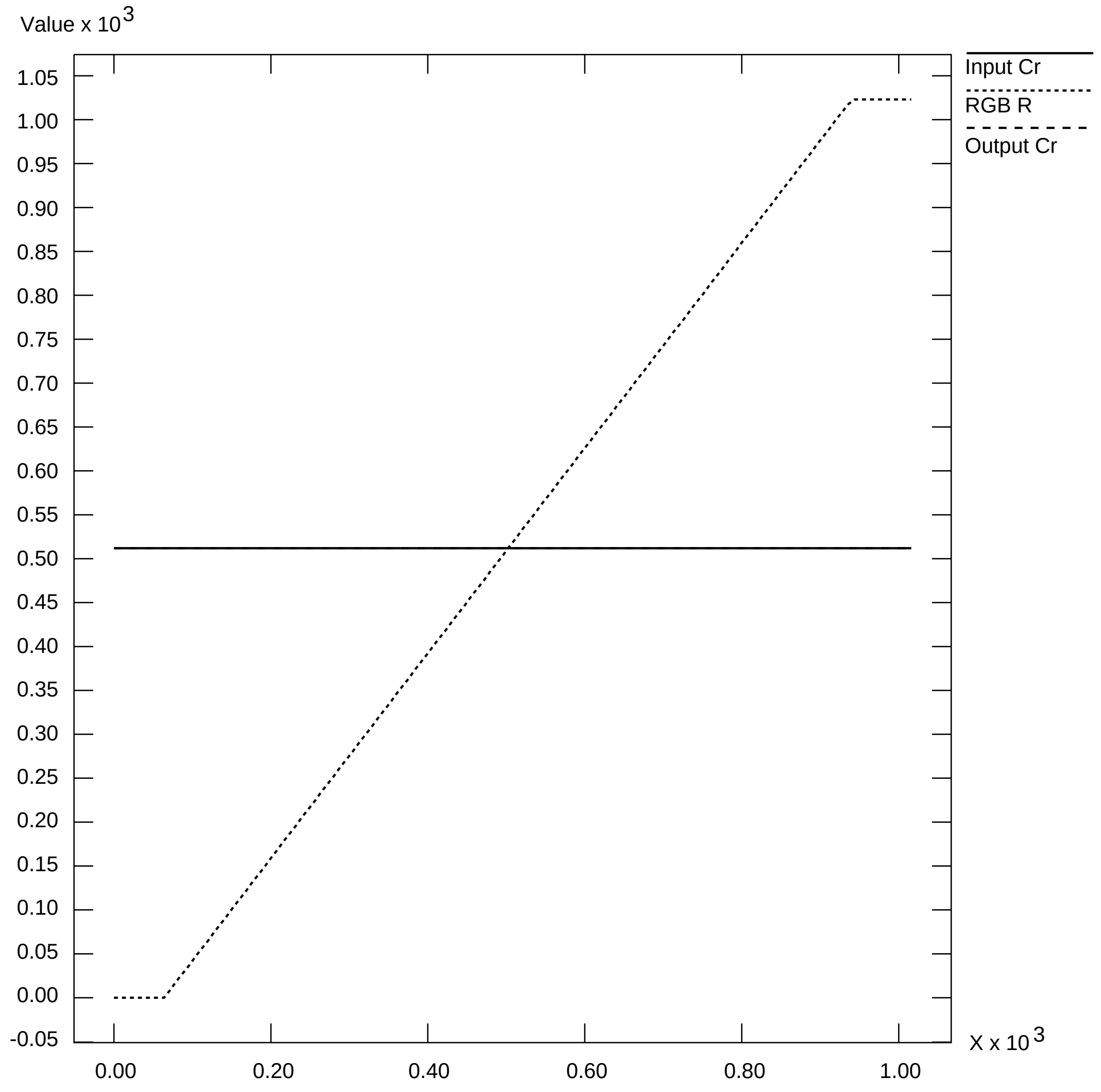


Figure D-6 Luminance Ramp: Cr/R

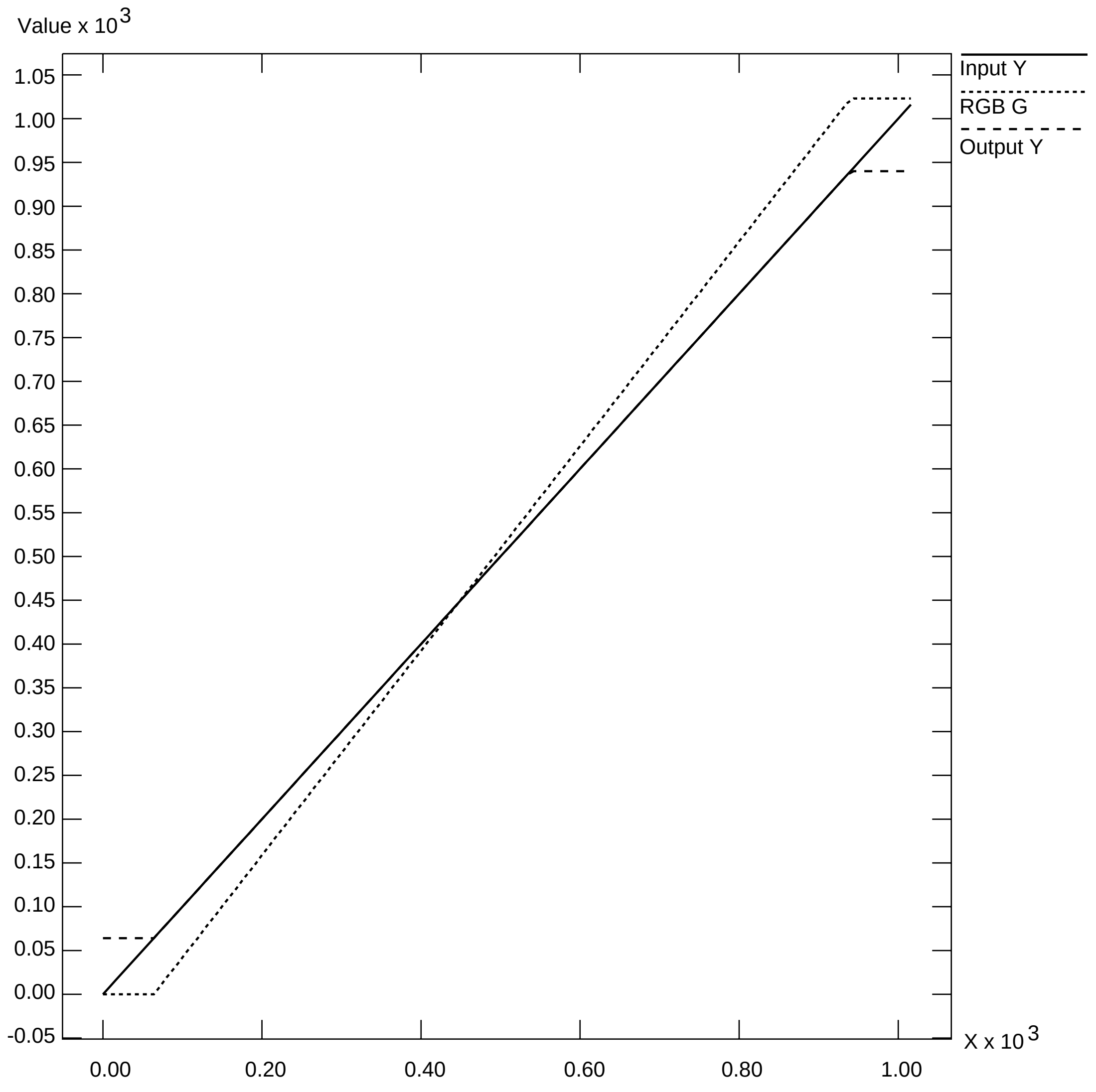


Figure D-7 Luminance Ramp: Y/G

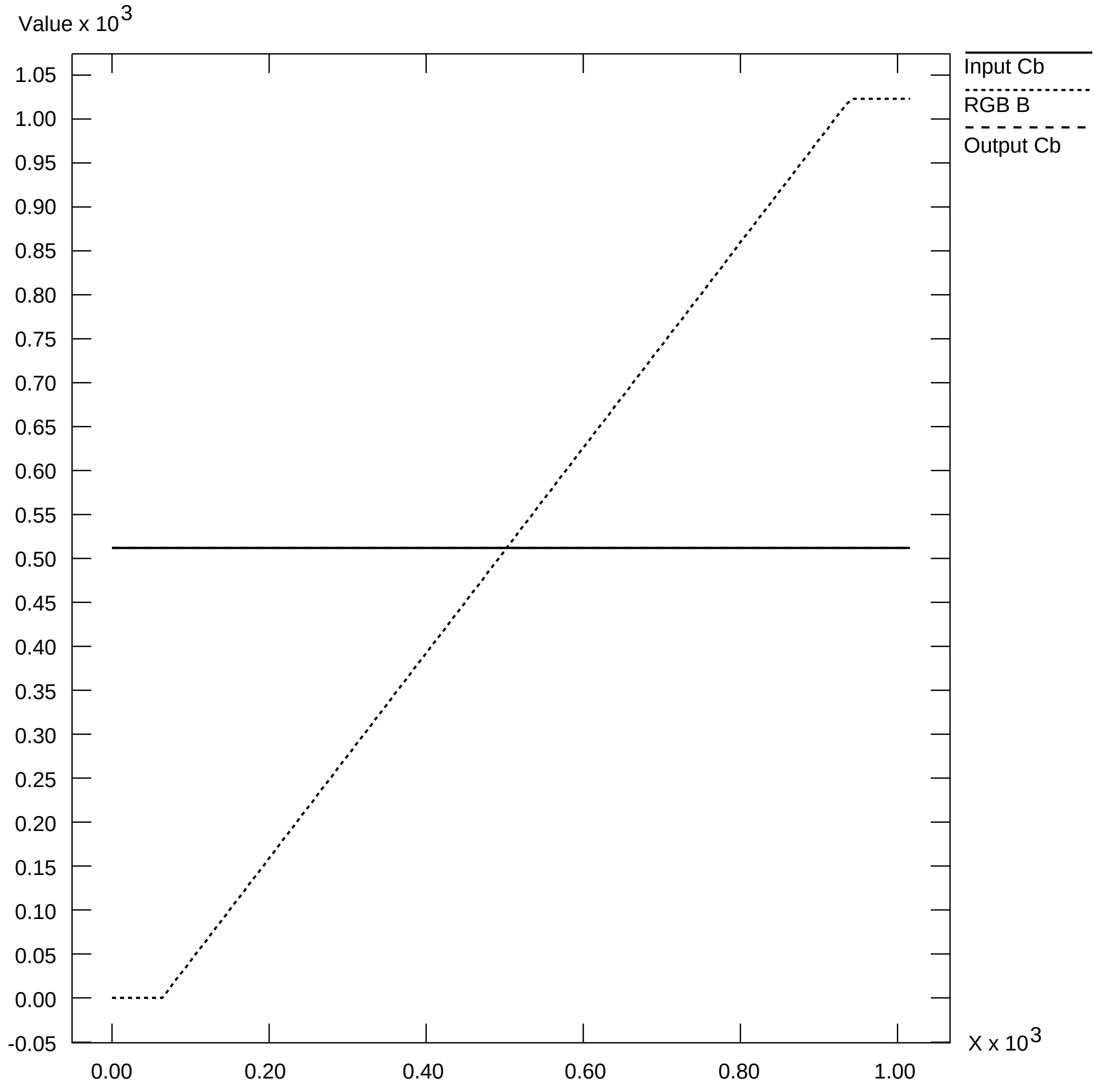


Figure D-8 Luminance Ramp: Cb/B

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.

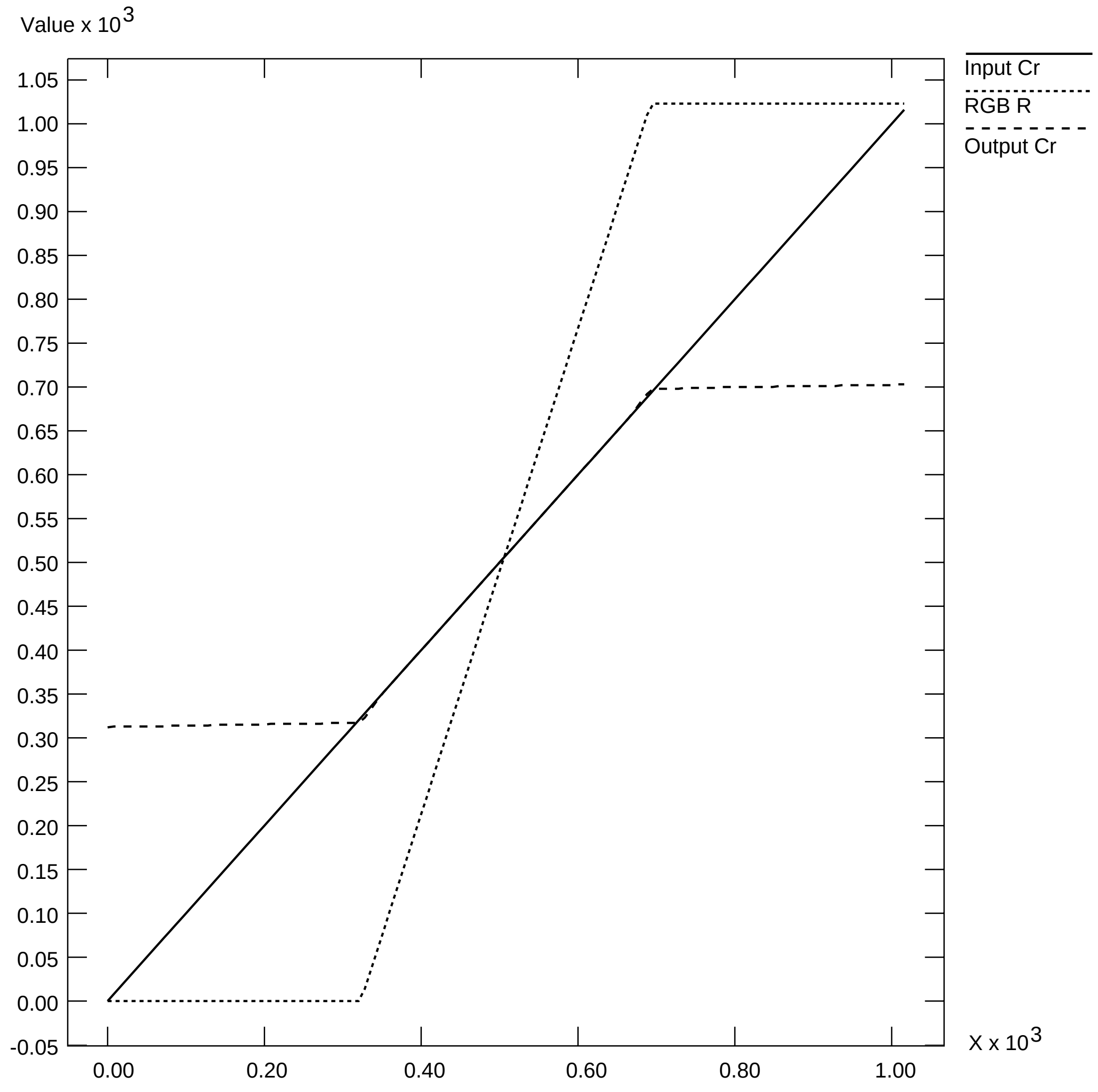


Figure D-9 Chroma/Luma Ramp: Cr/R

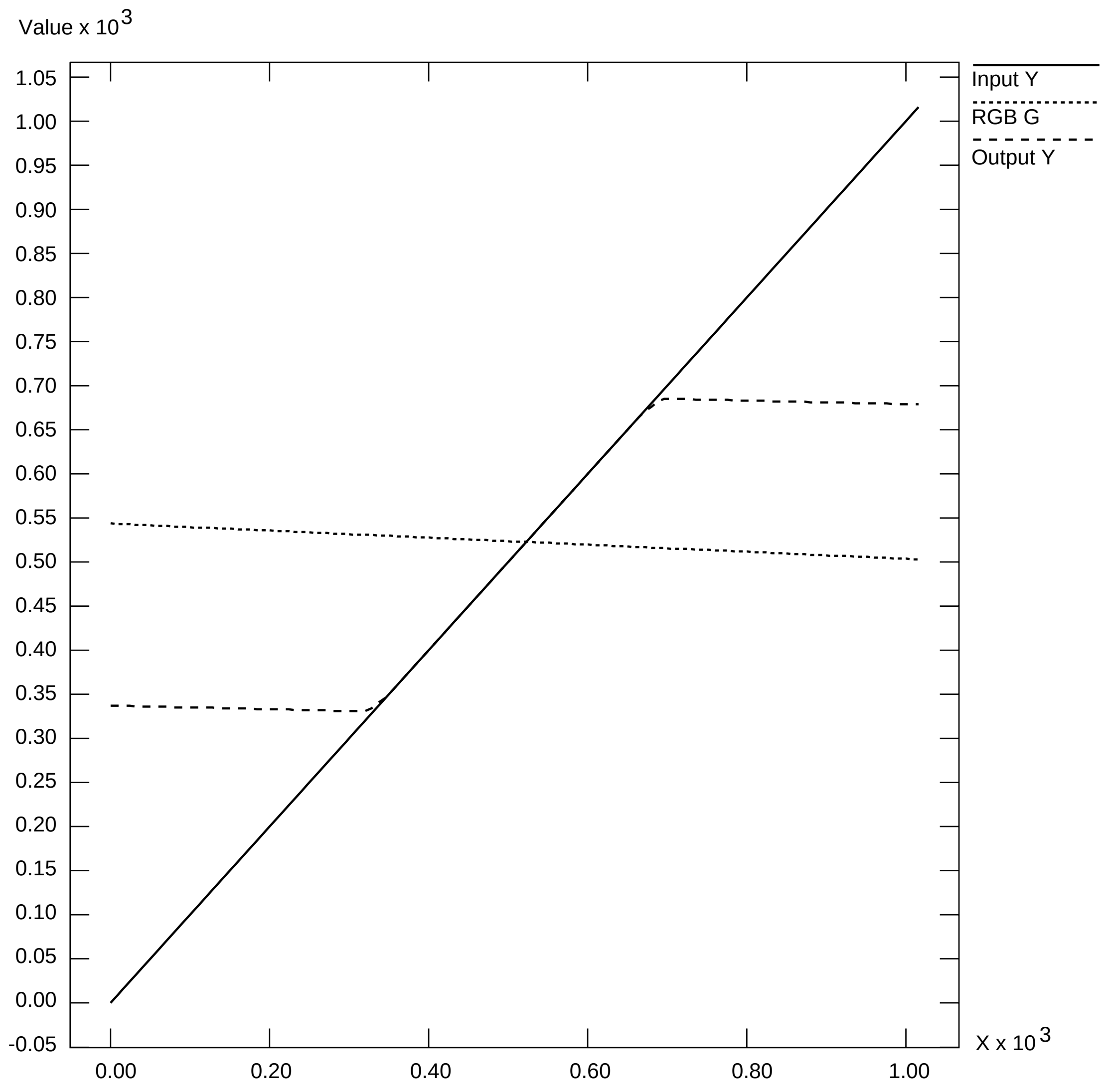


Figure D-10 Chroma/Luma Ramp: Y/G

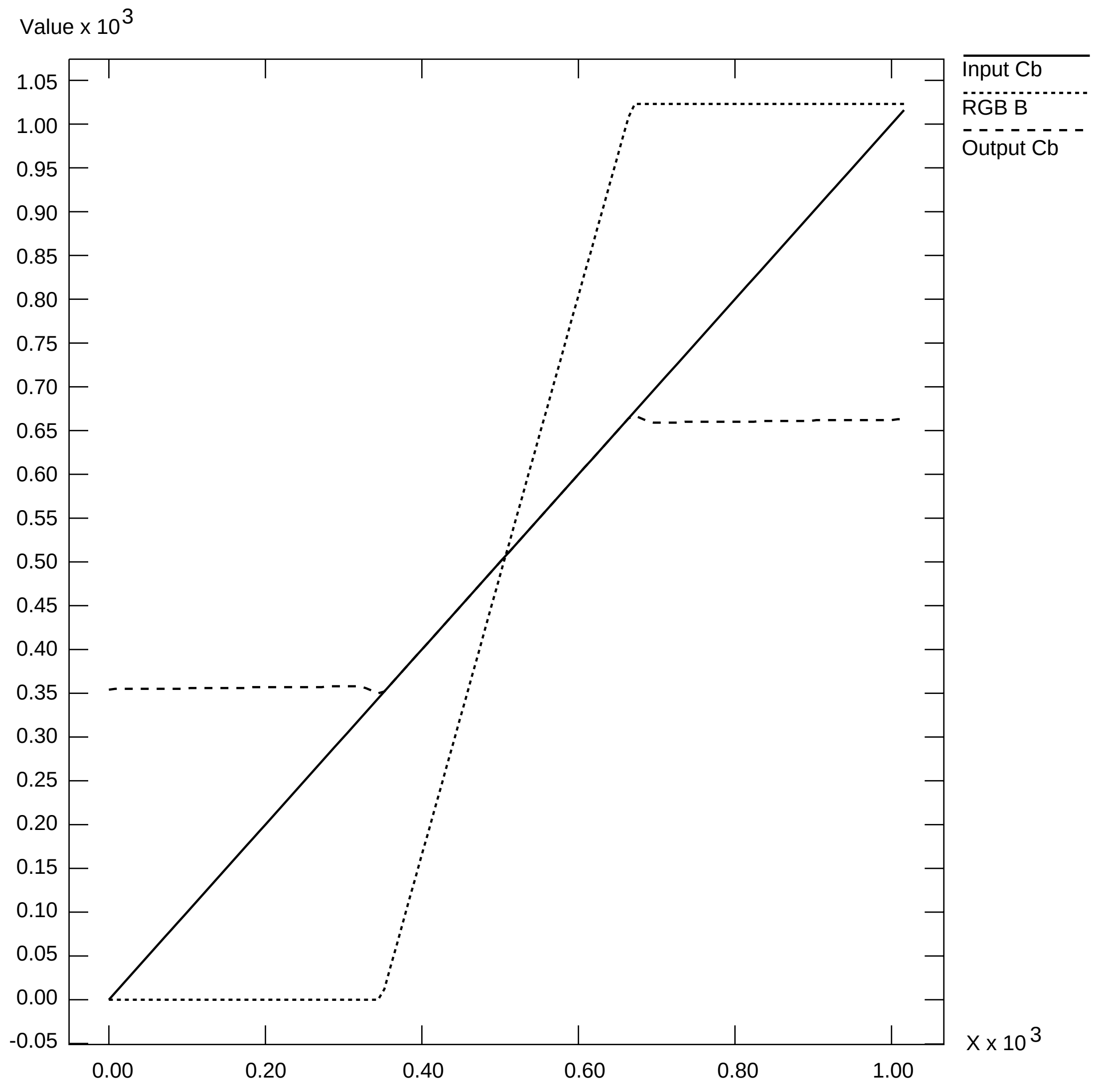


Figure D-11 Chroma/Luma Ramp: Cb/B

Index

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 007-3595-001.

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: 650-965-0964
- 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