DIVO Option and DIVO-DVC Option Owner's Guide

Document Number 007-3524-004

CONTRIBUTORS

Written by Carolyn Curtis and Alan Stein, with material by Sandra Motroni

Illustrated by Dany Galgani, Dan Young, Cheri Brown, and Carolyn Curtis

Production by Kirsten Pekarek and Karen Jacobson

Engineering contributions by Frank Bernard, Dheeren Bebortha, Tri Tran, Ashok Yerneni, Douglas Scott, Tony Chatzigianis, Scott Pritchett, Ed Miszkiewicz, Will McGovern, and Paul Spencer

Cover design by Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications

For regulatory and compliance information, see your system owner's guide.

Contents

List of Figures

List of Tables

About This Guide

The Digital Video Option (DIVO) board (marketing code XT-DIVO) is an XIO option board that provides broadcast-quality video for SGI workstations and servers that accept XIO boards. The option also provides 4 channels of audio.

The DIVO-DVC XIO option board (marketing code XT-DIVO-DVC) has all the functionality of the DIVO board and also supports DVCPRO, DV video coding and decoding, and DVCam.

Note: These option boards require IRIX 6.4 or later. Installations on SGI 3000 series systems require IRIX 6.5.10 or later (with patches, as required).

Features of both options are controlled with the Video Library (VL) and the Audio Library (AL). VL device-independent calls and controls are explained in the *Digital Media Programming Guide* (007-1799-060 or later; online only). That manual also gives information on using the AL.

Audience

This guide was written for the sophisticated video user in a professional or research environment. You should be familiar with video standards, the operation of the SGI workstation or server, and the VL information in the *Digital Media Programming Guide*.

Many current SGI owner's guides, programming guides, and user's guides are available through the World Wide Web: http://techpubs.sgi.com/library.

Structure of This Guide

This guide includes the following chapters and appendices:

- Chapter 1, “DIVO and DIVO-DVC Features and Capabilities,” outlines the main components of the DIVO option.
- Chapter 2, “Programming DIVO and DIVO-DVC,” describes using the VL to accomplish common specific tasks.
- Chapter 3, “Audio Data Conversion,” describes the Digital Media Audio Conversion Library, its converters, and its use.
- Appendix A, “,” summarizes technical specifications for the option boards.
- Appendix B, “,” describes connecting video equipment to DIVO or DIVO-DVC board connectors and using the control panel *vcp* to configure the DIVO board for the equipment.
- Appendix C, “,” sets forth all packing formats used by the DIVO or DIVO-DVC hardware.
- Appendix D, “,” explains DIVO/DIVO-DVC color spaces, the mathematical operations performed during conversions, and the implications of color-space conversions.
- Appendix E, “,” explains programming concepts, such as real-time disk I/O, and gives examples.
- Appendix F, “,” explains the DV and DVCPRO standards.
- Appendix G, “,” contains information on the DIVO board’s GPI receiver settings.

An index completes this guide.

Other Documents

Besides this guide, *Digital Media Connections* (007-3525-002 or later) is shipped with the DIVO and DIVO-DVC option boards.

The *Digital Media Programming Guide* is available with the IRIX digital media development environment software (`dmedia_dev`).

It is also a good idea to have your system owner’s guide available.

If you do not have these guides handy, the information 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 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/

  Once you are in the library, choose Catalogs > Hardware Catalog > and look under the Owner's Guides for the applicable owner's guide. For software guides, look on the bookshelf for the applicable IRIX version.

Conventions Used in This Guide

In command syntax descriptions and examples, square brackets ([ ]) surrounding an argument indicate an optional argument. Variable parameters are in *italics*. Replace these variables with the appropriate string or value.

In text descriptions, IRIX filenames are in *italics*.

Helvetica Bold font is used for labels on hardware, such as the names of ports on the I/O panel.

Messages and prompts that appear on-screen are shown in `typewriter font`. Entries that are to be typed exactly as shown are in **`boldface typewriter font`**.

Because the DIVO and DIVO-DVC options share most features, most of the information in this guide covers both options at once, such as the VL controls explained in Chapter 2. Specifically:

- The options are referred to as DIVO/DIVO-DVC in sections that describe features common to both (identical functionality); for example, "DIVO/DIVO-DVC Panel" in Chapter 1.
- The options are referred to as DIVO and DIVO-DVC in sections that describe both options; for example, "DIVO and DIVO-DVC Board Architecture" in Chapter 1.
- Separate portions of the guide describe features that occur in one option only; the heading or title clearly distinguishes these sections; for example, "DIVO-DVC Features" in Chapter 1, or Appendix G, "."

In each chapter or appendix in which the *Digital Media Programming Guide* is referenced, it is referred to by its full title at the first occurrence and thereafter as the DMPG.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please send them to SGI. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, you can find the document number on the back cover.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  `techpubs@sgi.com`

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

  `http://techpubs.sgi.com`

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

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Pkwy., M/S 535
  Mountain View, California 94043-1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801.

SGI values your comments and will respond to them promptly.

DIVO and DIVO-DVC Features and Capabilities

The DIVO XIO option board provides broadcast-quality video input and output for deskside and server workstations that accept XIO boards. Supporting the SMPTE 259 10-bit digital video standard, it fully integrates video into SGI workstation and server environments.

The DIVO option also provides transparent color space conversion between the YCbCr and RGB color spaces. The DIVO option board can be genlocked to an external digital or analog sync signal, and provides loopthrough to allow genlocking of other video equipment. The DIVO option provides built-in lossless compression (Rice compression), which results in an average 2:1 compression ratio, while maintaining bit-accurate data.

The DIVO-DVC option board has all the functionality of the DIVO option, but also supports DVCPRO, and DV video coding and decoding. The DIVO-DVC option consists of the DIVO base board and two factory-installed daughterboards.

Both options utilize one XIO slot and provide independent dual-link 10-bit serial digital component input and output ports. Depending on the system type, multiple DIVO or DIVO-DVC option boards can be installed on a system for video server applications.

The DIVO and DIVO-DVC options have no direct physical connection with the graphics board set, unlike some previous SGI video boards, such as Sirius Video. Any transfers to or from graphics must go through memory: DIVO has a CCIR 601 dual-link input to memory, and a CCIR 601 dual-link output from memory. Low-latency CCIR 601 output from graphics requires the GVO option board in addition; GVO is a daughterboard for the DG5 board, which is part of the Onyx2 InfiniteReality or Reality graphics board set.

This chapter consists of these sections:

Note: Installing an XIO board, such as the DIVO or DIVO-DVC board, in an Origin200 GIGAchannel chassis is explained in Chapter 2 of the *Origin200 and Origin200 GIGAchannel Maintenance Guide* (007-3709-001). Installing the DIVO or DIVO-DVC board in an Onyx2, Onyx 3000, Origin2000, or Origin 3000 chassis is performed only by a qualified SGI System Support Engineer (SSE).

DIVO and DIVO-DVC Features

The DIVO and DIVO-DVC options include these features:

- dual-link 10-bit serial digital video (SMPTE 259) streaming to and from system memory
- serial digital transport interface (SDTI; SMPTE 305) streaming to and from system memory
- transparent color-space conversion between YUV and RGB
- lossless built-in (Rice) compression and decompression using adaptive entropy coding that retains bit-accurate data with approximately 2:1 compression
- flexible data-packing capability to facilitate easy integration to OpenGL component packing methods

- unadjusted system time/media stream count (UST/MSC) hardware-supported audio/video synchronization mechanisms
- low latency in video transfers to or from system memory (typically less than one frame)
- support for fields and frames
- encoding through the in pipe and decoding through the out pipe
- AES3-1992 (AES/EBU) embedded audio and ancillary data (SMPTE 272M) with up to 24-bit precision and a sample rate of 48 KHz; up to 4 channels output and 16 channels input
- vertical interval time code (VITC) extraction and insertion recorded in the subcode and video auxiliary regions
- SDTI multiplexing and demultiplexing via a dmIC API
- hardware error detection and handling (EDH) and link autophasing
- for each video pipe, two channels of input and output trigger signal pairs based on the general-purpose interface (GPI)
- support for the Orion graphical V-LAN control utility
- output genlocking

The DIVO and DIVO-DVC option boards support video and audio data transfers to and from system memory only. You can view the video in real time on the Onyx2 and Onyx 3000 workstations using the OpenGL interface to copy video images to graphics. The DIVO and DIVO-DVC options support a wide variety of packing formats to facilitate easy integration of video in graphics; Appendix C, “,” explains these packings.

For controlling videotape recorders, you can use a direct RS-422 connection to the deck with third-party software, or an RS-422 V-LAN controller option and V-LAN software from SGI with the on-board GPI triggering mechanism.

The DIVO and DIVO-DVC options are fully integrated into the SGI Digital Media Library interfaces. The Video Library (VL) API has been enhanced to support some of its advanced features.

DIVO-DVC Features

In addition to the features summarized in "DIVO and DIVO-DVC Features," the DIVO-DVC option provides DV video coding and decoding. For explanations of the DVCPRO and DV standards, see Appendix F, "."

The option consists of the DIVO option board, which can manipulate digital video, including Rice lossless compression and decompression, and two daughterboards. With its speed of 25 Mbps, the DIVO-DVC option provides a flexible and cost-effective way to accomplish a wide range of video projects.

The DIVO-DVC option supports 4:1:1 sampling, which yields a 5:1 compression ratio, for both NTSC and PAL. The DIVO-DVC option also supports 4:2:0 PAL for the DV video formats. Audio processing for DV and DVCPRO is done via a dmAC; Chapter 3, "Audio Data Conversion," explains the procedure.

Despite its tape-oriented technology, DVCPRO, like DV, is an open format that is also designed for use with nonlinear editors and server-based systems. A 4X transfer speed record and play capability is possible for DVCPRO and DV, using SDTI; the mechanism is explained in "Programming the DIVO/DIVO-DVC Board for SDTI" on page 29 in Chapter 2.

VL-based sample applications, such as *divo_vitc*, *divo_ee*, *divo_memtovid*, *divo_vidtomem*, *divo_videowarp*, *divo_vidtotex*, *inline.c*, and *vflip*, are included with the DIVO and DIVO-DVC options in the directory */usr/share/src/dmedia/video/DIVO/*.

DIVO and DIVO-DVC Board Architecture

Figure 1-1 is a top-level diagram of the DIVO board. Each DIVO board has two independent pipes, each with its own dedicated R4650 processor and SDRAM.

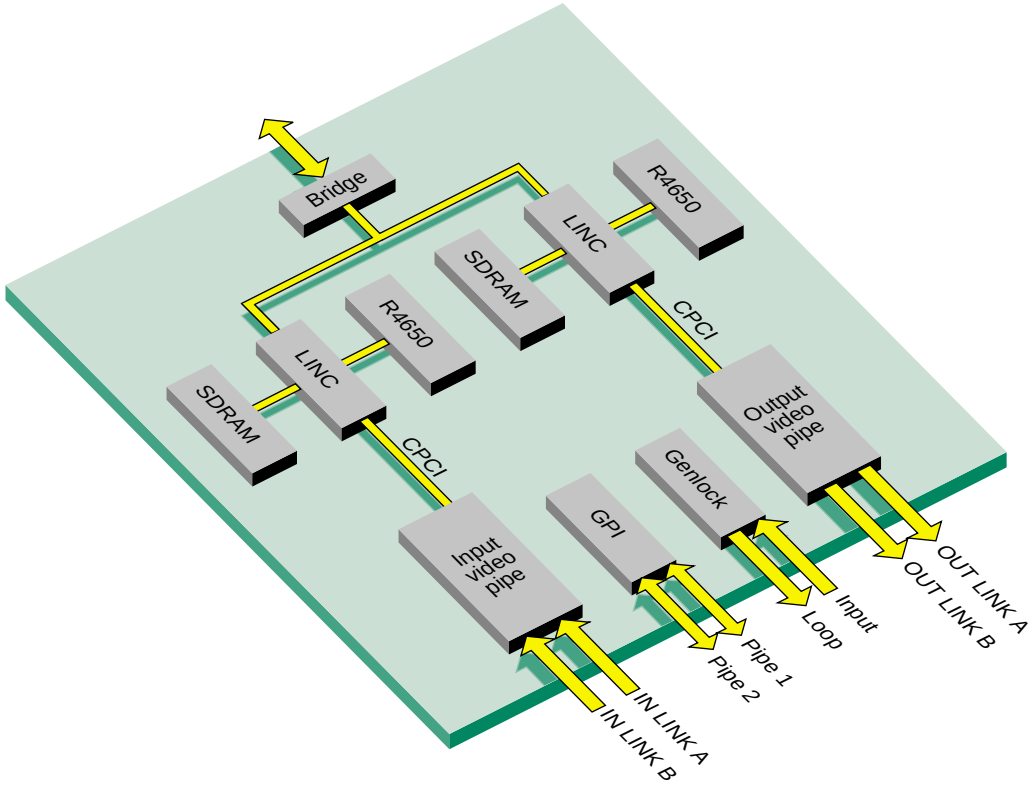

Figure 1-1 DIVO Board Architecture

Figure 1-2 is a top-level diagram of the DIVO-DVC board. Each DIVO-DVC board has two independent pipes, each with its own dedicated R4650 processor and SDRAM, and two identical daughterboards that provide DVC encoding and decoding.

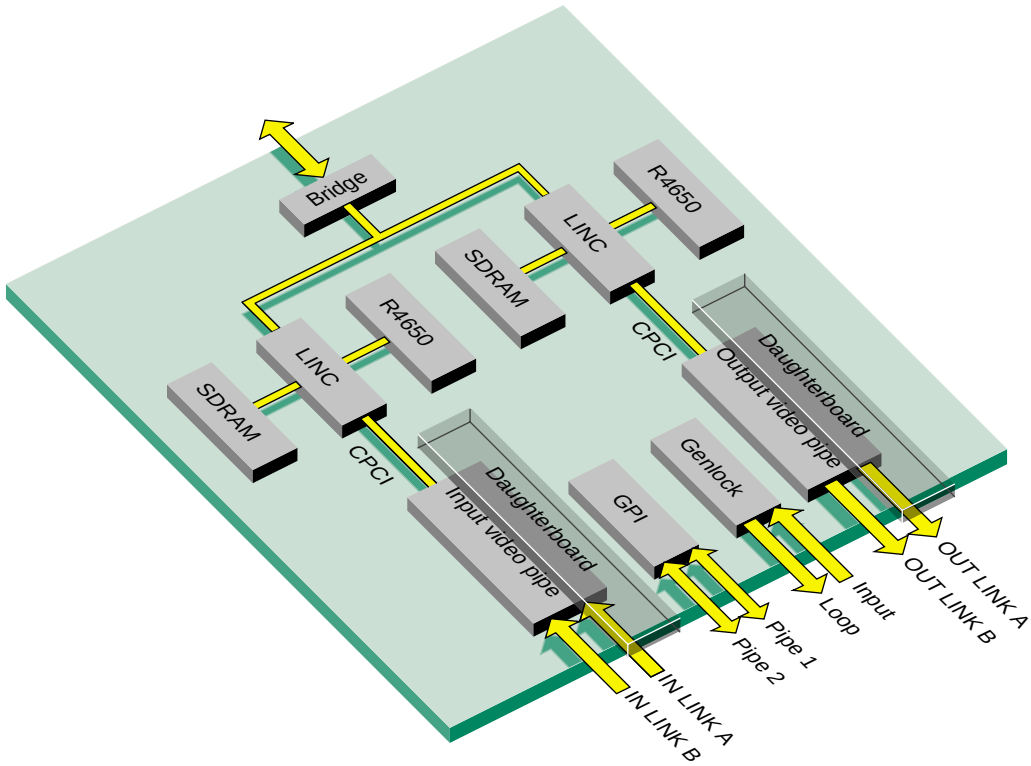

Figure 1-2 DIVO-DVC Board Architecture

The DVC decode card can decode DVCPRO- or DVC-formatted data stored on disk as a digital interface format (DIF) file. After decoding, the audio and video are encoded into a SMPTE 272-compliant audio/video serial stream.

DVCPRO 625-line rate data is processed as 4:1:1 data; in DVC, it is processed as 4:2:0 data. Thus, the DVCPRO superblock structure is similar to that for 525-line rate data, but the two formats are not compatible. Also, for DVCPRO, only 48 KHz audio is encoded.

Besides the sampling patterns 4:4:4 and 4:2:2, the DIVO-DVC board is also capable of 4:1:1 and 4:2:0 sampling. Details are in “4:1:1 Sampling (DIVO-DVC Only)” on page 107 and “4:2:0 Sampling (DIVO-DVC Only)” on page 108, respectively, in Appendix C.

DIVO/DIVO-DVC Panel

Figure 1-3 shows features of the DIVO/DIVO-DVC I/O panel. Although the board is installed vertically in the chassis, Figure 1-3 shows the panel sideways to facilitate reading of the connector and LED labels.

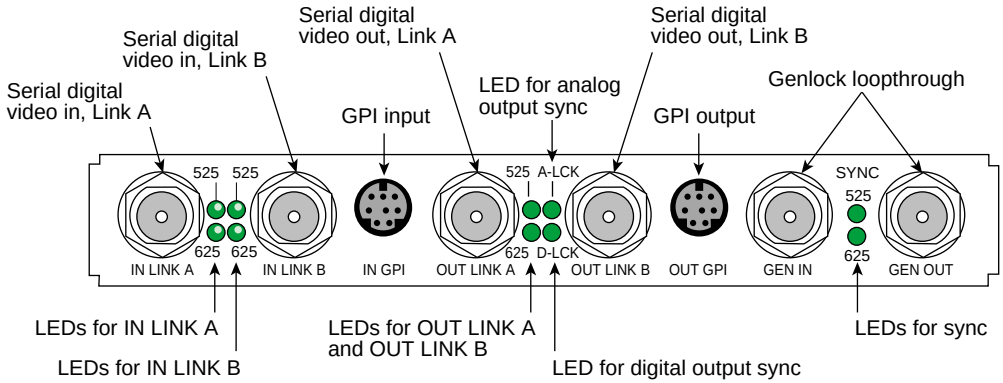

Figure 1-3 DIVO/DIVO-DVC I/O Panel

Table 1-1 summarizes DIVO/DIVO-DVC board external connectors that interface with video equipment.

Table 1-1 Interface for Video Equipment

Connector	Format	Use
IN LINK A, **IN LINK B**	10-bit CCIR 601-2 75-ohm BNCs Terminated, unbalanced	Serial digital video input from digital tape deck or other recording device. Conforms to SMPTE 259M for component video, SMPTE 272M for embedded audio, and SMPTE 266M for DVITC. Both inputs autophased.
OUT LINK A, **OUT LINK B**	10-bit CCIR 601-2 75-ohm BNCs	Serial digital video output to digital tape deck or other recording device. Conforms to SMPTE 259M for component video, SMPTE 272M for embedded audio, and SMPTE 266M for DVITC. **Note:** The transfer mode (packing format) selected determines LINK A and LINK B usage, as explained in Table A-3 and Table A-4 in Appendix A, “DIVO/DIVO-DVC I/O Panel Connectors”.

Table 1-1 (continued) Interface for Video Equipment

Connector	Format	Use
GEN IN	75-ohm BNC Loopthrough, unbalanced, unterminated	External analog sync source (precision time base or other source of house sync) or analog loopthrough.
GEN OUT	75-ohm BNC Loopthrough, unbalanced, unterminated	External reference loop out; passive loopthrough for genlock input with buffered signal to workstation. **Note:** If you attach a cable to one **GEN** connector, you must attach either another cable to other equipment accepting analog sync or a 75-ohm BNC terminator to the other **GEN** connector.
GPI IN, **GPI OUT**	8-pin mini-DIN	General Purpose Interface for each video port; frame-accurate event triggering to or from source or destination (tape deck or digital recorder). Configurable for switch closure (factory setting) or current sense operation.

See Appendix A, "," for technical details of the connectors, including GPI pinouts.

Table 1-2 summarizes the function of the LEDs on the panel.

Table 1-2 DIVO Panel LEDs

LED	Purpose
Leftmost LEDs between **IN LINK A** and **IN LINK B** (**525** and **625**)	Top LED lights when valid 525-line serial digital signal detected on **IN LINK A**. Bottom LED lights when valid 625-line serial digital signal detected on **IN LINK A.**
Rightmost LEDs between **IN LINK A** and **IN LINK B** (**525** and **625**)	Valid 525-line (top LED) or 625-line (bottom LED) serial digital signal detected on **IN LINK B.**
Leftmost LEDs between **OUT LINK A** and **OUT LINK B** (**525** and **625**)	Valid 525-line (top LED) or 625-line (bottom LED) serial digital signal detected on **OUTLINK A** and **OUT LINK B**; these outputs are locked together, regardless of whether **OUT LINK B** is used.
A-LCK	Output is locked to an analog source: standalone, genlock (to another sync source), or free run.
D-LCK	Output is locked to a digital source (**IN LINK A** or **IN LINK B**).
SYNC 525 and **625**	Valid 525-line sync source (top LED) or 625-line sync source (bottom LED) detected.

Figure 1-4 diagrams how the DIVO and DIVO-DVC option boards interact with other workstation components.

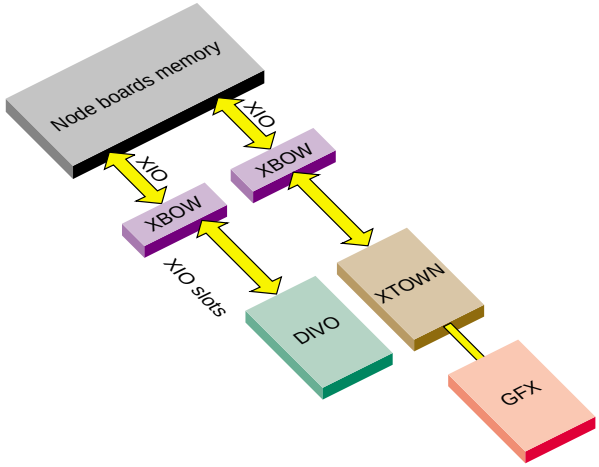

Figure 1-4 DIVO/DIVO-DVC Video Top-Level System Diagram

Digital Video Ports

The DIVO and DIVO-DVC options each have two independent 10-bit serial digital video ports for equipment that complies with the CCIR 601-2 standard. The ports can be configured for 4:4:4:4 or 4:2:2:4 in dual-link mode, or 4:2:2 in single-link mode where alpha is ignored.

Each port consists of two unidirectional interconnections, Link A and Link B:

- In 4:4:4:4 mode, Link A carries Y plus Cr and Cb from even-numbered sample points; Link B carries alpha plus Cr and Cb from odd-numbered sample points.
- In 4:2:2:4 mode, Link A carries Y plus Cr and Cb; Link B carries alpha only.

The video format selected determines Link A and Link B usage. For more information, see the following standards, which contain provisions for video signals:

- CCIR 601-2: Encoding Parameters of Digital Television for Studios (4:2:2 component video signals, single link)
- ANSI/SMPTE 125M-1992: Television—Component Video Signal 4:2:2—Bit-Parallel Digital Interface
- SMPTE Recommended Practice (RP) 175-1993: Digital Interface for 4:4:4:4 Component Video Signals (Dual Link)
- SMPTE 259M-1993: Television—10-Bit 4:2:2 Component and $4f_{sc}$ NTSC composite Digital Signals—Serial Digital Interface
- SMPTE RP 157-1990: Key Signals
- SMPTE 272M: Television - Formatting AES/EBU Audio and Auxiliary Data into Digital Video Ancillary Data Space

Color-Space Converters

Four color spaces are native to the DIVO/DIVO-DVC option: full-range RGBA, compressed range RGB (RP-175), CCIR601, and full-range YUV. The video interface supports only RP-175 and CCIR 601-2. The memory interface supports all four color spaces. See “Color Spaces” in Appendix C for an explanation of full- and compressed-range color spaces.

The DIVO/DIVO-DVC option uses the Gennum GF9105 component digital transcoder. The GF9105 uses 13-bit multiplier coefficients and provides up to 13-bit output resolution, allowing for transparent color-space conversion between YUV and RGB.

Serial Data Transport Interface (SDTI)

The DIVO/DIVO-DVC option supports the transmission of audio, subcode data, and compressed video packets associated with DV-based 25-Mbps data structures for 525/60 and 625/50 systems, including faster than real-time transfers and multichannel transmission.

The section "Programming the DIVO/DIVO-DVC Board for SDTI" in Chapter 2 explains how to program the DIVO/DIVO-DVC option board for SDTI. For more information on SDTI and on SDTI with embedded DV, refer to the most recent editions of the following:

- Proposed SMPTE standard 305M for Television—Serial Data Transport Interface
- Proposed SMPTE Standard 306M for Television Digital Recording—6.35-mm Type D-7 Component Format, Video Compression at 25 Mb/s, 525/60 and 625/50
- Proposed SMPTE Standard 314M for Television—Data Structure for DV-Based Audio, Data and Compressed Video - 25 and 50 Mb/s

SMPTE's web site is http://www.smpte.org/.

Interpolation and Decimation Filters

The GF9105 transcoder provides interpolation and decimation filtering between the 4:2:2:4 and 4:4:4:4 sampling rates. Both interpolation and decimation operations are fully compliant with the CCIR 601-2 standard.

Orion Graphical V-LAN Control Utility

The DIVO/DIVO-DVC option supports the SGI Orion graphical utility for tapedeck control and for capture and playback. Figure 1-5 shows the main window.

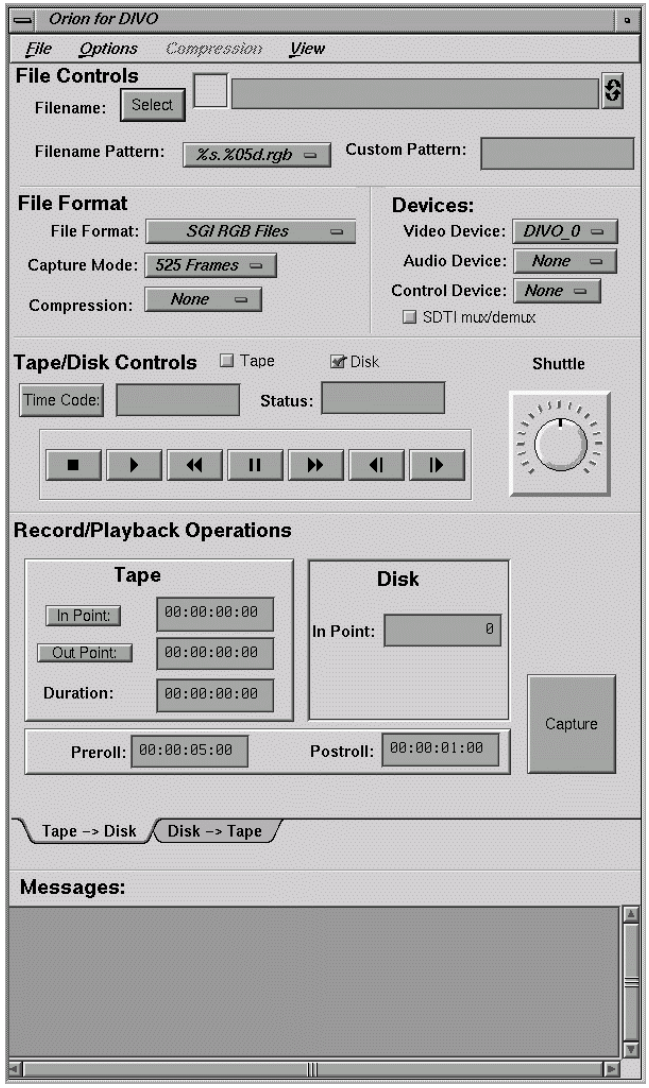

Figure 1-5 Orion Main Window

The Orion utility offers

- standard deck controls: rewind, fast forward, stop, play, and so on
- editing controls, such as in point, out point, preroll, postroll
- deck shuttle control
- a variety of video file formats including QuickTime
- capture and playback in both field and frame mode
- support for synchronized audio and video capture and playback
- V-LAN triggering for capture and playback
- support for multiple DIVO/DIVO-DVC boards and multiple V-LAN devices

---

Note: If "Audio Device" is set to Default in the video control panel (`vcp`) and the DIF stream contains audio data, audio is played out on the default audio devices while Video Decode/Playout is in progress.

The Orion utility is at `/usr/dmedia/bin/DIVO/orion`. The utility is fully described in its man page, orion(1M).

DIVO/DIVO-DVC Audio

The DIVO/DIVO-DVC option provides one 4-channel wide input device and one four-channel wide output device, which are compliant with the SMPTE 272M standard. Applications can open either device as two or four channels. Unused output channels are set to zero; unused input channels are discarded. DIVO/DIVO-DVC audio supports a sample rate of 48 KHz. It supports 20- and 24-bit word sizes. Access to the audio is through the Audio Library (AL) interfaces specified in the *Digital Media Programming Guide*.

Programming DIVO and DIVO-DVC

The DIVO and DIVO-DVC boards support the Video Library (VL) and the Audio Library (AL) programming APIs. The APIs are described in the *Digital Media Programming Guide* (007-1799-060 or later; hereafter referred to as the DMPG).

This chapter explains

VL Basics for DIVO and DIVO-DVC

To build programs that run under VL, you must

- install the *dmedia_dev* and *dmedia_eoe* options
- link with *libvl*
- include *dmedia/vl.h* and *dmedia/vl_DIVO.h* for device-dependent functionality

The client library for VL is */usr/lib32/libvl.so*. The header files for the VL are in */usr/include/dmedia*; the main file is *vl.h*. This file contains the main definition of the VL API and controls that are common across all hardware. Several useful digital media programming examples are in */usr/share/src/dmedia/tools* (such as *capture/avcapture*, *capture/avplayback*, and *vlan/vlan*).

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

For more information on the Video Library and the API usage, see the latest version of the DMPG.

This section explains

VL Concepts

The Video Library defines a basic set of primitives and mechanisms to specify interconnections and controls to achieve the desired setup. The two central concepts for VL are

- *path*: an abstraction for a way of moving data around
- *node*: an endpoint of the path

The basic nodes are a *source* (such as a VTR) and a *drain* (such as memory). Figure 2-1 diagrams the simplest VL path, with one of each of these two nodes.

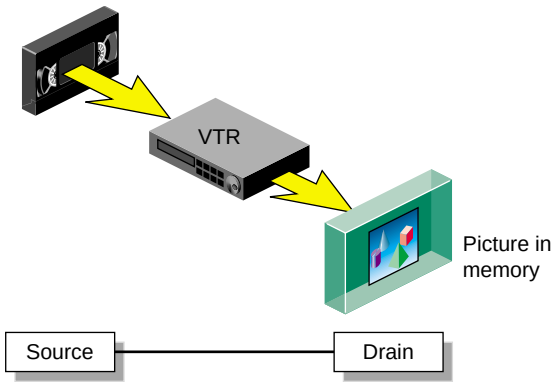

Figure 2-1 Simple VL Path

DIVO/DIVO-DVC nodes are further discussed in “VL Nodes for DIVO and DIVO-DVC” on page 19.

VL Syntax Elements

VL 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()**

VL Object Classes

The VL recognizes these classes of objects:

- *devices*, each including sets of nodes
- *nodes*, which are sources, drains, and internal nodes (as discussed in the preceding section)
- *paths*, connecting sources and drains (as discussed in the preceding section)
- *buffers*, for sending and receiving field/frame data to and from host memory

  DIVO and DIVO-DVC require the use of DMbuffers (digital media buffers) and not the original ring buffer mechanisms (VL buffers) used with earlier SGI video options. The new buffering scheme is much more flexible and versatile than the older VL buffer-based scheme. See Chapter 5 of the DMPG.

  DMbuffers, an abstraction of main memory, allow efficient and API-independent interchange of data between the different digital media libraries. For example, video fields can be captured into DMbuffers via VL and then displayed in graphics using OpenGL. They can also be passed between two processes without the data having to be copied explicitly. Refer to Chapter 5, "Digital Media Buffers," in the *Digital Media Programming Guide* for details.
- *events*, for monitoring video I/O status
- *controls*, or parameters that modify how data flows through nodes; for example:
  - video device parameters, such as sync source
  - video data parameters such as packing, size, and color space

VL controls fall into two categories:

- *device-global* or *device-independent* (prefix VL_), which can be used by several SGI video products

  For details of the device-independent controls, refer to the DMPG.
- *device-dependent* (prefix VL_DIVO_), specific to a particular video device, in this case, DIVO and DIVO-DVC

Both types of VL controls are explained in this chapter with respect to their usage with DIVO and DIVO-DVC.

VL Nodes for DIVO and DIVO-DVC

Use **vlGetNode()** to specify nodes. This call returns the node's handle, which is used when setting controls or setting up paths. Its function prototype is:

```
VLNode vlGetNode(VLServer svr, int type, int kind, int number)
```

In this prototype, variables are as follows:

svr Names the server (as returned by **vlOpenVideo()**).

type Specifies the type of node:

- VL_SRC: source, such as a digital tapedeck connected to an input port

  Note: The DIVO and DIVO-DVC options have only one input.

- VL_DRN: drain, such as system memory
- VL_DEVICE: global control, such as a default source; Table 2-1 summarizes the values for this type

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

kind Specifies the kind of node:

- VL_VIDEO: connection to a video device equipment; for example, a video tapedeck or camera
- VL_MEM: workstation memory

number Number of the node in cases of two or more identical nodes, such as two video source nodes. The default value for all *kind*s is 0.

VL_ANY can also used as a value for *number* to reference the first available node of the specificated *type* and *kind*.

In general, a path on DIVO and DIVO-DVC has a memory node and a video node. The following fragment creates a digital video input source node and a memory drain node, and creates the path:

```
VLServer svr;
VLPath path;
VLNode src;
VLNode drn;
                    /*Set up video source node */
VLControlValue timing,format, val;
src = vlGetNode(svr, VL_SRC, VL_VIDEO, VL_ANY);
                    /*Set up memory drain node */
drn = vlGetNode(svr, VL_DRN, VL_MEM, VL_ANY);
                    /* Create source-to-drain path */
if((path = vlCreatePath(svr, VL_ANY, src, drn)) < 0){
     fprintf(stderr,"%s\n",vlStrError(vlGetErrno()));
     exit(1);
}
                    /* Set up path with shared src and drn node */
vlSetupPaths(svr, (VLPathList)&path, 1, VL_SHARE, VL_SHARE);
```

After nodes are specified, use **vlSetControl()** to specify parameters:

- video nodes: video timing (drain only) and format (for example digital component)
- memory nodes: timing, packing, and color space

Controls for each node are defined in "DIVO/DIVO-DVC Controls" later in this chapter, and are summarized in Table 2-1.

VL Data Transfer Functions

This section summarizes VL data transfer categories, and gives the basic steps of creating an application. For DIVO and DIVO-DVC, VL data transfers always involve memory (video to memory, memory to video) and require setting up a DMbuffer pool.

In the VL programming model, the process of creating a VL application consists of these steps:

1. Open a connection to the video daemon (**vlOpenVideo()**).
2. Specify nodes on the data path (**vlGetNode()**).
3. Create the path (**vlCreatePath()**).
4. Optional step: add more nodes to a path (**vlAddNode()**).
5. Set up the hardware for the path (**vlSetupPaths()**).
6. Specify path-related events to be captured (**vlSelectEvents()**, **vlAddCallback()**).
7. Set input and output parameters (controls) for the nodes on the path (**vlSetControl()**); video format and timing must be specified.
8. Create a dmBuffer pool to hold data for memory transfers (**vlDMGetParams()**, **dmBufferSetPoolDefaults()**, **dmBufferCreatePool()**, **vlGetTransferSize()**).
9. Register the buffer (**vlDMPoolRegister()**,**vlDMPoolDeregister()**).
10. Start the data transfer (**vlBeginTransfer()**).
11. Get the data (**vlDMBufferGetValid()**, **vlDMBufferPutValid()**, **dmBufferAllocate()**, **dmBufferAllocateSize()**, **dmBufferGetPoolState()**, **dmBufferGetPoolFD()**, **dmBufferSetPoolSelectSize()**, **dmBufferMapData()**, **dmBufferFree()**) to manipulate frame data.
12. Handle data stream events (**vlSelectEvents()**, **vlNextEvent(), vlPending()**).
13. Clean up (**vlEndTransfer()**, **vlDMPoolDeregister()**, **vlDestroyPath()**, **vlCloseVideo()**)**.**

Note: Error handling (**vlPerror()**) is accomplished throughout.

DIVO/DIVO-DVC Controls

To determine the available devices (that is, video options in the workstation, such as the DIVO or DIVO-DVC option board) and the nodes available on them, run *vlinfo*. To determine possible controls for each device, enter

```
vlinfo -l
```

Note: VL controls specified as true with **vlSetControl()** are executed immediately. However, it is not guaranteed that they execute at a specific time. For better precision on the execution of these controls, see "Setting Inline Controls" on page 41.

To set controls for DIVO/DIVO-DVC nodes, use **vlSetControl()**. The following example sets video format and timing on a node:

```
timing.intVal = VL_TIMING_525_CCIR601;
format.intVal = VL_FORMAT_RGB;

if (vlSetControl(svr, path, drn, VL_TIMING, &timing) <0)
{
     vlPerror("VlSetControl:TIMING");
     exit(1);
}
if (vlSetControl(svr, path, drn, VL_FORMAT, &format) <0)
{
     vlPerror("VlSetControl:FORMAT");
     exit(1);
 }
```

For details on **vlSetControl()** and **vlGetControl()**, see the latest version of the DMPG.

Tables in this section summarize

- device-global controls for DIVO/DIVO-DVC
- controls for DIVO/DIVO-DVC nodes
- control values and uses

Table 2-1 summarizes supported node controls for the DIVO/DIVO-DVC option.

Table 2-1 DIVO/DIVO-DVC Node Controls

Control	Video Source	Memory Source	Video Drain	Memory Drain
VL_ASPECT		X (read-only)		X (read only)
VL_CAP_TYPE		X		X
VL_COLORSPACE		X		X
VL_COMPRESSION		X		X
VL_DIVO_CLOSED_CAPTION	X		X	
VL_DIVO_FIELD_DISPLAY	X		X	
VL_DIVO_LOOPBACK	X			
VL_FIELD_DOMINANCE	X		X	
VL_FORMAT	X		X	
VL_GPI_OUT_MODE	X		X	
VL_GPI_STATE	X		X	
VL_OFFSET	X (read-only)	X	X (read-only)	X
VL_PACKING		X		X
VL_RICE_COMP_PRECISION		X		X
VL_RICE_COMP_SAMPLING		X		X
VL_SDTI_HEADER		X		X (read-only)
VL_SDTI_MODE		X		X
VL_SIZE	X (read-only)	X	X (read-only)	X
VL_SYNC			X	
VL_SYNC_SOURCE			X	
VL_TIMING	X (read-only)	X	X	X
VL_TRANSFER_TRIGGER	X		X	
VL_ZOOM	X (read-only)	X (read-only)	X (read-only)	X (read-only)

Table 2-2 summarizes the values and uses of controls for the DIVO/DIVO-DVC option.

Table 2-2 Controls for the DIVO and DIVO-DVC Option Boards

Control	Values or Range	Use
VL_ASPECT	Aspect (read-only).	Reads aspect ratio.
VL_CAP_TYPE	Memory nodes: VL_CAPTURE_FIELDS VL_CAPTURE_INTERLEAVED VL_CAPTURE_NONINTERLEAVED	Selects type of frame(s) or field(s) to capture.
VL_COLORSPACE	VL_COLORSPACE_RGB (full-range RGB) VL_COLORSPACE_CCIR601 (compressed-range YUV) VL_COLORSPACE_RP175 (compressed-range RGB) VL_COLORSPACE_YUV (full-range YUV)	Specifies color space of video data in memory.
VL_COMPRESSION	VL_COMPRESSION_NONE VL_COMPRESSION_RICE VL_COMPRESSION_DV (DIVO-DVC only) VL_COMPRESSION_DVCPRO (DIVO-DVC only)	Specifies compression option for video. See *vl.h* for information on compression-specific controls. For example, to access Rice entropy coding, use VL_COMPRESSION_RICE as the compression control for the memory node.
VL_DIVO_CLOSED_CAPTION	VL_DIVO_CLOSED_CAPTION_ON VL_DIVO_CLOSED_CAPTION_OFF	Specifies if closed-caption should be extracted from input or inserted into output.
VL_DIVO_LOOPBACK	VL_DIVO_LOOPBACK_ON VL_DIVO_LOOPBACK_OFF	Specifies if the video source on input should be from the output pipe.
VL_DIVO_FIELD_DISPLAY	VL_DIVO_FIELD_DISPLAY_F1F2 VL_DIVO_FIELD_DISPLAY_F2F1 VL_DIVO_FIELD_DISPLAY_F1F1 VL_DIVO_FIELD_DISPLAY_F2F2	Specifies output field display order. For example, to display only one field of a DVCPRO frame, use VL_DIVO_FIELD_DISPLAY_F1F1. This control is supported only in DV/DVCPRO compression mode on the DIVO-DVC option board.

Table 2-2 (continued) Controls for the DIVO and DIVO-DVC Option Boards

Control	Values or Range	Use
VL_FIELD_DOMINANCE	VL_F1_IS_DOMINANT VL_F2_IS_DOMINANT (Frames that are output are deinterlaced differently depending on the choice of output field dominance. Deinterlacing is specified in the application.)	Identifies frame boundaries in a field sequence; see "Setting Field Dominance."
VL_FORMAT	VL_FORMAT_DIGITAL_COMPONENT_SERIAL VL_FORMAT_DIGITAL_COMPONENT_DUAL_SERIAL	Sets video format in or out: Serial 4:2:2:4 Serial 4:4:4:4
VL_GPI_OUT_MODE	Conditions: VL_GPI_OUT_XFER_START VL_GPI_OUT_XFER_STOP	Specifies when the GPI_OUT line is asserted, to control downstream devices in a studio environment. For more information, see "VL Support for the General-Purpose Interface (GPI)." Asserts GPI_OUT at BeginTransfer. Asserts GPI_OUT at EndTransfer.
VL_GPI_STATE	State: VL_GPI_CLEAR VL_GPI_OFF VL_GPI_ON VL_GPI_PULSE (transition for one field time)	Sets or gets the state of output_gpi lines. For more information, see "VL Support for the General-Purpose Interface (GPI)" on page 36.
VL_OFFSET	Any position within the video raster.	Sets the position within the video raster to stuff bits.
VL_PACKING	Supported packings; see Appendix C, ","for information.	Sets packing format for memory source or drain node.
VL_RICE_COMP_PRECISION	VL_RICE_COMPRESSION_8 VL_RICE_COMPRESSION_10 VL_RICE_COMPRESSION_12 VL_RICE_COMPRESSION_13	Specifies the component size.
VL_RICE_COMP_SAMPLING	VL_RICE_COMPRESSION_422 VL_RICE_COMPRESSION_4224 VL_RICE_COMPRESSION_444 VL_RICE_COMPRESSION_4444	Specifies the sampling resolution.

Table 2-2 (continued) Controls for the DIVO and DIVO-DVC Option Boards

Control	Values or Range	Use
VL_SDTI_HEADER	stringVal	Sets or gets the SDTI header as an array of bytes (stringVal).
VL_SDTI_MODE	VL_SDTI_MODE_OFF VL_SDTI_MODE_ON	Specifies if the memory node data should be processed as SDTI data.
VL_SIZE	Any size of the raster.	Size plus offset or origin should not exceed the raster dimensions.
VL_SYNC	VL_SYNC_INTERNAL VL_SYNC_GENLOCK	Sets sync mode for source or drain; on source, this is set to VL_SYNC_GENLOCK.
VL_SYNC_SOURCE	VL_DIVO_SYNC_HOUSE VL_DIVO_SYNC_DIGITAL_INPUT_LINK_A VL_DIVO_SYNC_DIGITAL_INPUT_LINK_B VL_DIVO_SYNC_DBOARD	Selects the genlock source: VL_DIVO_SYNC_DBOARD is used when the compressed stream provides a sync source.
VL_TIMING	 VL_TIMING_525_CCIR601 VL_TIMING_625_CCIR601	Sets or gets video timing: 13.50 MHz, 720 x 486 13.50 MHz, 720 x 576
VL_TRANSFER_TRIGGER	VL_TRIGGER_NONE VL_TRIGGER_GPI VL_TRIGGER_VITC VL_TRIGGER_MSC	Specifies the conditions under which transfers begin on a path (video nodes only); see “Using VL_TRANSFER_TRIGGER” in this chapter.
VL_ZOOM	Zoom factor (read-only)	Reads zoom factor of video stream.

Compression Through the VL

Compression is handled via enhancements to the VL API; the DIVO and DIVO-DVC boards do not support the Compression Library API.

For DIVO and DIVO-DVC, compression in the VL is supported by adding compression-related controls on memory nodes. The control VL_COMPRESSION specifies the compression. Based on the compression type—Rice, DV (DIVO-DVC only), DVCPRO (DIVO-DVC only), DVCam (DIVO-DVC only), or none—additional controls specify compression-related parameters. Table 2-2 earlier in this chapter summarizes these controls.

Note: Compression for the DIVO-DVC option is different from compression for DIVO; see "DV and DVCPRO Compression for DIVO-DVC" on page 27.

Rice Compression

The DIVO and DIVO-DVC option boards can use Rice compression, a lossless entropy coding mechanism that provides an average compression of 2:1 while retaining bit-accurate data. In some cases, compression can add to the size of the data being transferred, but is limited to a maximum of 1% increase. Tests at SGI show that this compression mechanism can reduce data in ratios of 2:1 to 6:1.

For encoding, specify the controls VL_COMPRESSION, VL_RICE_COMP_PRECISION, and VL_RICE_COMP_SAMPLING. For decoding, the control information is already embedded in the stream; you need specify only the data type, with the VL_COMPRESSION control. For best memory utilization with Rice compression, use variably sized buffers.

Refer to the source examples (*`divo_vidtomem.c`* for encoding and *`divo_memtovid.c`* for decoding) in the directory */usr/share/src/dmedia/video/DIVO*.

DV and DVCPRO Compression for DIVO-DVC

The DIVO-DVC board is capable of additional types of encoding defined in the DV, DVCam, and DVCPRO standards. For DV/DVCam and DVCPRO encoding, specify the control VL_COMPRESSION with the VL_COMPRESSION_DV or VL_COMPRESSION_DVCPRO parameter, respectively. For decoding, the control information is already embedded in the stream; you need specify only the data type, with the VL_COMPRESSION control. DV/DVCam and DVCPRO compression mechanisms can reduce data in the ratio of 5:1.

For 525/60, frame size is 12,000 bytes; for 625/50, frame size is 144,000 bytes. You must specify VL_PACKING as VL_PACKING_DV, as illustrated by the following code fragment.

```
{
   int GvlPacking;
   VLControlValue val;
   int GvlCspace;
   int GvlCapType;
```

```
VL_PACKING_242_8
VL_COLORSPACE_CCIR(0)

GvlCapType = VL_CAPTURE_INTERLEAVED;

   val.intVal = GvlPacking;
   if (vlSetControl(GvlSvr, GvlPath, GvlMemNode, VL_PACKING, &val) < 0)
   vl_error("vlSetControl(VL_PACKING) on memory node");
   if (vlGetControl(GvlSvr, GvlPath, GvlMemNode, VL_PACKING, &val) < 0)
   vl_error("vlGetControl(VL_PACKING) on memory node");
   if (val.intVal != GvlPacking)
   other_error("can't set VL_PACKING on memory node");

   val.intVal = GvlCspace;
   if (vlSetControl(GvlSvr, GvlPath, GvlMemNode, VL_COLORSPACE, &val) <
0)
   vl_error("vlSetControl(VL_COLORSPACE) on memory node");
   if (vlGetControl(GvlSvr, GvlPath, GvlMemNode, VL_COLORSPACE, &val) <
0)
   vl_error("vlGetControl(VL_COLORSPACE) on memory node");
   if (val.intVal != GvlCspace)
   other_error("can't set VL_COLORSPACE on memory node");

   val.intVal = GvlCapType;
   if (vlSetControl(GvlSvr, GvlPath, GvlMemNode, VL_CAP_TYPE, &val) <
0)
   vl_error("vlSetControl(VL_CAP_TYPE) on memory node");
   if (vlGetControl(GvlSvr, GvlPath, GvlMemNode, VL_CAP_TYPE, &val) <
0)
   vl_error("vlGetControl(VL_CAP_TYPE) on memory node");
   if (val.intVal != GvlCapType)
   other_error("can't set VL_CAP_TYPE on memory node");

}
```

Use the VL_DIVO_FIELD_DISPLAY control to specify output field display order. To display only one field of a DVCPRO frame, specify the field. In the following example, VL_DIVO_FIELD_DISPLAY_F2F2 is used:

```
/* display only field # 2 */
 field_display = VL_DIVO_FIELD_DISPLAY_F2F2;
 val.intVal = field_display;
 if (vlSetControl(GvlSvr, GvlPath, GvlVidNode, VL_DIVO_FIELD_DISPLAY,
&val) <0)
```

```
     vl_error("vlSetControl(VL_DIVO_FIELD_DISPLAY) on video node");
 if (vlGetControl(GvlSvr, GvlPath, GvlVidNode, VL_DIVO_FIELD_DISPLAY,
&val) <0)
     vl_error("vlGetControl(VL_DIVO_FIELD_DISPLAY) on video node");
 if (val.intVal != field_display)
     other_error("can't set VL_DIVO_FIELD_DISPLAY on video node");
```

Programming the DIVO/DIVO-DVC Board for SDTI

For the DIVO and DIVO-DVC options, SDTI with embedded DV is handled by means of features in the VL API and hardware support in the input and output pipes. The API includes the following SDTI features:

- The 32-bit packing VL_PACKING_SDTI_DV specifies how the signal is stored in memory. The layout for this packing is diagrammed in "SDTI Packing" on page 99 in Appendix C.
- The VL has two SDTI-related controls on memory nodes:
  - VL_SDTI_MODE enables or disables SDTI mode. When SDTI is enabled and the payload data type is DV, the start lines for each field are changed for 525 timing: start lines are F1=21, F2=284 instead of the default F1=21, F2=283. (For 625 timing, start lines are the same as the defaults, F1=23 and F2=336.)
  - VL_SDTI_HEADER sets or gets the SDTI header as an array of bytes; the data structure is explained in this section.

This section consists of the following:

SDTI Data Structure

For both DIVO and DIVO-DVC, the control VL_SDTI_HDR overlaps the string variable. The SDTI header is defined in *vl_divo.h* as follows:

```
                /*  SDTI Header  */
typedef struct {
  u_char did;            /* data id identifier */
  u_char sdid;           /* sec. data id identifier */
```

```
  u_char dc;              /* data word count */
  u_char lnum[2];         /* linenumber0 and linenumber1 */
  u_char lnumcrc[2];      /* linenumber0 and linenumber1 crc */
  u_char aai_code;        /* appl identifier and code */
  u_char dst_addr[16];    /* destination address */
  u_char src_addr[16];    /* source address */
  u_char btype;           /* block type */
  u_char crcflag;         /* payload crc flag */
  u_char dextflag;        /* data extension flag */
  u_char reserved[4];     /* reserved words */
}


void init_sdti_header_dv( sdti_hdr_t *  sdti_hdr, VLControlValue *val)
{
  int i;

  sdti_hdr->did           = 0x40; /* see smpte 305M spec */
  sdti_hdr->sdid          = 0x01;
  sdti_hdr->dc            = 0x2E;
  sdti_hdr->lnum[0]       = 0x00;
  sdti_hdr->lnum[1]       = 0x00;
  sdti_hdr->lnumcrc[0]    = 0x00;
  sdti_hdr->lnumcrc[1]    = 0x00;
  sdti_hdr->aai_code      = 0x01;

  for(i=0;i<16;i++) sdti_hdr->dst_addr[i] = 0; /* dst_addr[0] is msb */
  for(i=0;i<16;i++) sdti_hdr->src_addr[i] = 0; /* src_addr[0] is msb */

  sdti_hdr->btype         = DIVO_SDTI_BLOCK_TYPE_FIXED; /* fixed block,
no ecc
*/
  sdti_hdr->crcflag       = 0x01;  /* insert CRC */
  sdti_hdr->dextflag      = 0x00;  /* no data extension packets */
  for(i=0;i<4;i++) sdti_hdr->reserved[i] = 0; /* set to zero */

  /* copy sdti header into control string variable */
  bcopy(sdti_hdr,val->stringVal,sizeof(sdti_hdr_t));

}

    /* Set the SDTI Header */
    if (vlSetControl(GvlSvr, GvlPath, GvlMemNode, VL_SDTI_HEADER, &val)
< 0)
      vl_error("vlSetControl(VL_SDTI_HEADER) on memory node");
```

Sending SDTI DV

To multiplex and send SDTI DV, follow these steps:

1. Enable SDTI mode by setting the VL_SDTI_MODE control.
2. Set the color space to VL_COLORSPACE_CCIR601.
3. Set the packing to VL_PACKING_SDTI_DV.
4. Initialize the SDTI header by setting the VL_SDTI_HEADER control. For an example, see init_sdti_header_dv in `/usr/share/src/dmedia/video/DIVO/divo_memtovid.c`
5. Set *ysize* to the active payload region.
6. Multiplex DIF frames and send buffers. For an example, see init_sdti_header_dv in `/usr/share/src/dmedia/video/DIVO/divo_memtovid.c`

Note: The Reed-Solomon error correction code (ECC) is not supported on DIVO or DIVO-DVC.

Receiving SDTI DV

To receive and demultiplex SDTI DV, follow these steps:

1. Enable SDTI mode by setting the VL_SDTI_MODE control.
2. Set the color space to VL_COLORSPACE_CCIR601.
3. Set the packing to VL_PACKING_SDTI_DV.
4. Set *ysize* to the active payload region.
5. Receive field buffers and demultiplex DIF frames. For an example, see init_sdti_header_dv in `/usr/share/src/dmedia/video/DIVO/divo_memtovid.c`

Setting Field Dominance

Field dominance identifies the frame boundaries in a field sequence; that is, it specifies which pair of fields in a field sequence constitute a frame. The control VL_FIELD_DOMINANCE allows you to specify whether an edit occurs on the nominal video field boundary (Field 1, or F1) or on the intervening field boundary (Field 2, or F2).

- F1 dominant: The edit occurs on the nominal video field boundary.
- F2 dominant: The edit occurs on the intervening field boundary.

Whether a field is Field 1 or Field 2 is determined by the setting of bit 9, the F bit, in the XYZ word of the EAV and SAV sequences. The setting means the following:

- For Field 1 (also called the odd field), the F bit is 0.
- For Field 2 (also called the even field), the F bit is 1.

Figure 2-2 shows fields and frames as defined for digital 525-line and 625-line formats for the DIVO/DIVO-DVC option.[1]

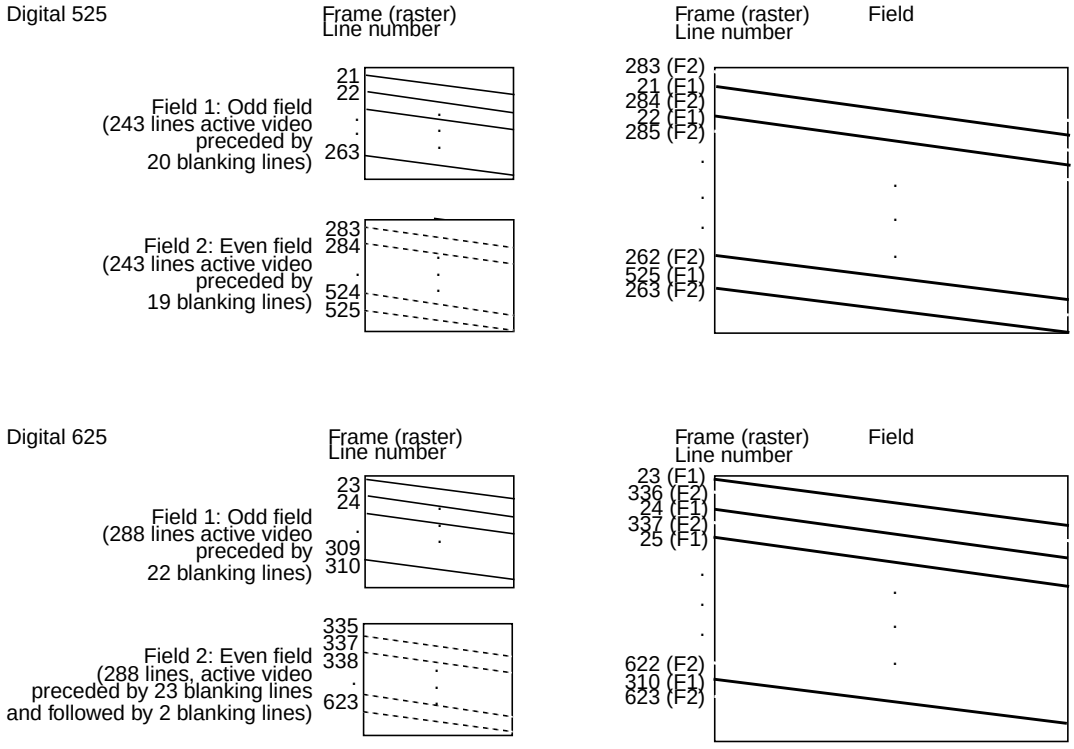

Figure 2-2 DIVO/DIVO-DVC 525-Line and 625-Line Frames and Fields

Editing is usually on Field 1 boundaries, where Field 1 is defined as the first field in the video standard's two-field output sequence. 525-line standards send the second (whole) raster line out to begin the first field, and the first (half) raster line out to begin the second field; 625-line standards send the first (half) raster line out to begin the first field, and the second (whole) raster line to begin the second field.

[1] The line numbers for the DIVO/DIVO-DVC option diverge slightly from those defined in SMPTE 259M.

However, some users may want to edit on F2 boundaries, which fall on the field in between the video standard's frame boundary. To do so, use this control, then program your deck to select F2 edits.

If you use component 525-line format, you might need to vary the field dominance choice, depending on the origin of the input material they are to edit.

To output a set of frames, they must be deinterlaced into fields differently, depending on the choice of output field dominance. For example, when F1 dominance is selected, the field with the topmost line must be the first field to be transferred; when F2 dominance is selected, the field with the topmost line must be the second field to be transferred. Deinterlacing must be specified in the application; the following code fragment contains an example of how to consult the field dominance control to determine deinterleave order.

```
/*
* Set the memory node's timing based upon the video drain's timing,
 * which has been set up by the daemon from the defaults file, or by
 * the user via vcp.
 *
 * When we get around to reading image files, we'll check the file
 * size against the size reported by the VL for this node: if the file
 * size does not match the format's, we'll punt.
 */

int is_525, F1_is_first;
VLControlValue drn timing, dominance;
if (vlGetControl(svr, MEMtoVIDPath, drn, VL_TIMING, &drainTiming) < 0)
{
    vlPerror("GetControl(VL_TIMING) on video drain failed");
    exit(1);
}
if (vlSetControl(svr, MEMtoVIDPath, src, VL_TIMING, &drainTiming) < 0)
{
    vlPerror("SetControl(VL_TIMING) on memory source failed");
    exit(1);
}
```

```
/*
 * Read the video drains's field dominance control setting and timing,
 * then set a variable to indicate which field has the first line, so
 * readimage() will know how to deinterleave frames to fields.
 */
if (vlGetControl(svr, MEMtoVIDPath, drn,
    VL_FIELD_DOMINANCE, &dominance) < 0) {
    vlPerror("GetControl(VL_FIELD_DOMINANCE) on video drain failed");
    exit(1);
}

is_525 = (drainTiming.intVal == VL_TIMING_525_CCIR601));

switch (dominance.intVal) {
    case VL_F1_IS_DOMINANT:
        if (is_525) {
            F1_is_first = 0;
        } else {
            F1_is_first = 1;
        }
        break;
    case VL_F2_IS_DOMINANT:
        if (is_525) {
            F1_is_first = 1;
        } else {
            F1_is_first = 0;
        }
        break;
    }
```

VL Support for the General-Purpose Interface (GPI)

The DIVO and DIVO-DVC option boards have two GPI connectors, each associated with one of the serial digital video ports. Each port has two transmit and two receive channels, lines 0 and 1 in and out.

The VL API supports the GPI as a device-independent interface. It supports GPI triggers in three **vlSetControl()** interfaces. The union VLControlValue supports the controls for

- output triggering, discussed in "Using VL_GPI_OUT_MODE" on page 36
- explicitly setting and querying the GPI lines, discussed in "Using VL_GPI_STATE" on page 37
- setting up triggering to begin transfers on a path, discussed in "Using VL_TRANSFER_TRIGGER" on page 38

Note: See Appendix G, "," and the section "GPI Interface" in Appendix A for extensive hardware information on the GPI interface for the DIVO board only.

Using VL_GPI_OUT_MODE

Use VL_GPI_OUT_MODE to program the gpi_out line. Two conditions are supported for asserting the GPI line: transfer_start and transfer_stop. You can have multiple trigger conditions outstanding.

Figure 2-3 diagrams the effect of VL_GPI_OUT_MODE values.

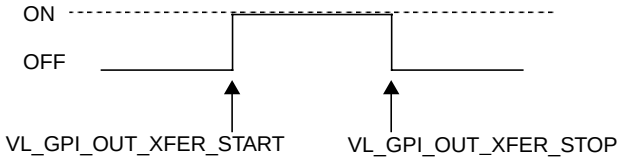

Figure 2-3 Output GPI (OFF Assumed to Be Low)

Note: In this section, OFF is assumed to be low. VL_CLEAR_GPI changes the state of the GPI signal to low.

The following pseudocode segment illustrates a setup for gpi_out line 1 to toggle at the beginning and end of transfer:

```
VLControlValue val;

                          /* make sure the GPI line is high */
val.gpi_state.gpi   = VL_GPI_OUT;
val.gpi_state.instance = <which GPI line>;       /* integer */
val.gpi_state.state =  VL_GPI_OFF;
vlSetControl(svr,path,VL_GPI_STATE,&val);

                          /* transfer start */
val.gpi_out.condition = VL_GPI_OUT_XFER_START;
val.gpi_out.instance  = <which GPI output line >;
val.gpi_out.state     = VL_GPI_ON;

vlSetControl(svr,path,VL_GPI_OUT_MODE,&val);

                          /* transfer stop */
val.gpi_out.condition = VL_GPI_OUT_XFER_STOP;
val.gpi_out.instance  = <which GPI output line >;
val.gpi_out.state     = VL_GPI_OFF;

vlSetControl(svr,path,VL_GPI_OUT_MODE,&val);
```

To clear all outstanding trigger controls on a particular line, use the gpi_state control with the clear flag.

Using VL_GPI_STATE

Use VL_GPI_STATE to query the state of the input GPI lines and to set or get the state of output GPI lines. The states are ON, OFF, PULSE (transition for one field time), and CLEAR.

The following code fragment clears all output triggers on the specified line:

```
val.gpi_state.gpi   = VL_GPI_OUT;
val.gpi_state.instance = <which GPI line>;
val.gpi_state.state =  VL_GPI_CLEAR;

vlSetControl(svr,path,VL_GPI_STATE,&val);
```

To get the GPI state on an input line, use

```
val.gpi_state.gpi = VL_GPI_IN;
val.gpi_state.instance = <which GPI line>;
vlGetControl (svr, path, VL_GPI_STATE, &val);
```

Using VL_TRANSFER_TRIGGER

The VL_TRANSFER_TRIGGER control specifies the conditions under which transfers begin on a path based on external GPI triggers. This control is valid only on video nodes.

The following code illustrates an input GPI-based trigger transfer setup:

```
VLTransferDescriptor desc;
int num_images = 30;   /* 1 second of video */
int trigger = TRUE;/* enable trigger transfers */

if (trigger) {
   val.xfer_trigger.triggerType = VL_TRIGGER_GPI;
   val.xfer_trigger.value.instance = 1;     /* GPI line 1 */
   desc.trigger = VLDeviceEvent;
}
else {   /* reset for subsequent transfers */
    val.xfer_trigger.triggerType = VL_TRIGGER_NONE;
    desc.trigger = VLTriggerImmediate;
}

vlSetControl(svr, path, src, VL_TRANSFER_TRIGGER, &val);

desc.mode = VL_TRANSFER_MODE_DISCRETE;
desc.count = num_images;
desc.delay = 0;

vlBeginTransfer(svr, path, 1, &desc);
```

Note: The trigger type is a persistent device control setting and remains valid until reset. Thus the VL_TRANSFER_TRIGGER control must be set to VL_TRIGGER_NONE if subsequent transfers are of a nontrigger type.

VL Support for Vertical Interval Time Code (VITC)

Over-the-wire electrical time-code signals are used as clocks by which one master device can drive the input or output of data by other slave devices. Each tick of the clock consists of a unique signal that represents a time code.

VITC time code is a standardized part of a 525- or 625-line video signal. The code itself occupies lines in the vertical blanking interval of each field of the video signal. Each VITC code word contains a time code, 32 user bits, an F1/F2 field indicator, and other useful information. People use the user bits to store information such as reel and shot number for indexing footage. Under certain circumstances, the original VITC recorded with footage can go along with that footage as it is edited, so that you can produce an edit list or track assets for a final prototype edit.

The DIVO and DIVO-DVC options always extract VITC information from the fields as they are captured and pass this information to the application as buffer information associated with the dmBuffer. To extract this information, call **vlDMBufferGetVideoInfo()**. In the structure returned, a valid bit (validinfo) indicates if the VITC is valid. Refer to the programming example *divo_vitc.c* in the directory */usr/share/src/dmedia/video/DIVO*.

To insert VITC, fill the DMBufferVideoInfo* structure and call the function **vlDMBufferSetVideoInfo()**.

To parse a VITC image out of a pixel buffer, you can use a VITC parser included in *libdmedia* (see **dmVITC(3dm)**); see */usr/share/src/dmedia/video/vitc.c* for sample code.

DIVO/DIVO-DVC Events

The VL provides several ways of handling data stream events, such as completion or failure of data transfer, vertical retrace event, loss of the path to another client, lack of detectable sync, or dropped fields or frames. The method you use depends on the kind of application you are writing:

- For a strictly VL application, use
  - **vlSelectEvents()** to choose the events to which you want the application to respond
  - **vlCallback()** 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 are 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.

For more information on these functions, see Chapter 4 in the *Digital Media Programming Guide*.

Table 2-3 summarizes events for DIVO/DIVO-DVC. For these options, this table supersedes the table of events in Chapter 14, "VL Event Handling," in the DMPG; DIVO and DIVO-DVC support only the events listed in Table 2-3.

Table 2-3 DIVO/DIVO-DVC Events

Event	Use
VLSyncLost	Sync is not detected
VLStreamStarted	Stream started delivery
VLStreamStopped	Stream stopped delivery
VLSequenceLost	A field/frame was dropped
VLControlChanged	A control on the path has changed
VLTransferComplete	A field/frame transfer has completed
VLTransferFailed	A transfer has failed and DMA is aborted
VLEvenVerticalRetrace	Vertical retrace event for an even field
VLOddVerticalRetrace	Vertical retrace event for an odd field
VLDeviceEvent	A device-specific event
VLTransferError	A transfer error was discovered; field may be invalid

Setting Inline Controls

Because of the asynchronous nature of the implementation, controls executed with **vlSetControl()** have no guarantees as to when they are executed once transfers are in progress. However, in certain situations, it is useful to be able to specify when controls on a path are executed; for example, to play out video clips with different packings or color-space formats without having to stop transfers while a memory-to-video operation is in progress.

Inline controls specify control changes to happen between buffers. For example, if you want to play out two video clips that have different packing formats in memory, the application would set up the path, queue the buffers from the first clip, set up inline controls to match the next clip, and queue the buffers from the second clip. The syntax for usage is

```
int vlSetControlInLine(VLServer svr, VLPath path, VLNode node,
     VLNode refnode, VLControlType control, VLControlValue *val)
```

In this syntax, *refnode* is the reference node, identifying a unique connection in a path with more than two nodes.

Inline controls are generally applied on a memory node being used as the source node. Control changes are queued to the hardware along with the buffers and are executed in order. To change packing control inline, for example, use

```
VLNode mem;
mem = vlGetNode(svr, VL_SRC, VL_MEM, VL_ANY);
val.intval = VL_PACKING_444_12;
vlSetControlInLine(svr, path, mem, mem, VL_PACKING, &val);
```

Capturing Graphics to Video

To capture graphics to video, you can use OpenGL to read pixels into memory. However, the coordinate system differs between video and Open GL; under OpenGL, the origin is at the lower left corner and in video, origin is in the upper left corner. To adjust for this difference, the image must be inverted in memory before it is sent to the DIVO board for output.

Besides sending an image out to the DIVO board, you can use the GVO graphics option to get zero-latency transcoding to CCIR 601-2 digital video.

Reporting

The DMediaInfo structure reports the Unadjusted System Time (UST) and VITC information.

The DIVO and DIVO-DVC options make use of the error events noted in Chapter 4 of the DMPG, as well as VLTransferErrorEvent, which reports nonfatal video transfer errors, including error detection and handling (EDH) errors. The VLTransferComplete and VLSequenceLost events also report the Media Stream Count (MSC) of the field or frame transferred or failed.

Audio Data Conversion

The Digital Media Audio Conversion Library provides data format conversion for applications that do real-time audio capture, playback, and file conversion. This library lets you move data efficiently between any audio producer and any audio consumer, regardless of their native formats.

This chapter describes the Digital Media Audio Conversion Library, its converters, and its use, in these sections:

Digital Media Audio Conversion Library

The Digital Media Audio Conversion library provides a single API for performing memory-to-memory sound compression and format conversion. This library serves as a "wrapper" around the standalone Digital Media Audio Codecs and Digital Media Audio Rate Conversion Library, allowing conversion of audio data without the need to keep track of buffer sizes and other such issues.

In addition to the compression and decompression done by the codec, an audio converter may perform the following transformations:

- audio sampling rate conversion
- conversion between different numerical sample representations, such as unsigned integer and two's complement signed integer
- conversion between big-endian and little-endian byte orders
- conversion between different numbers of interleaved channels, such as mono and stereo
- pulse code modulation (PCM) mapping
- scaling or offsetting samples by arbitrary amounts

Compression, decompression, and audio sampling rate conversion can be accomplished using either the Digital Media Audio Conversion Library or standalone conversion-specific routines (see "DV Audio Compression Library" on page 60). All other conversions listed above can be accomplished only with the Digital Media Audio Conversion Library.

The commonly used codec accessed through the Audio Conversion API is DM_AUDIO_DV, which is DV and DVCPRO audio compression and decompression. See "DV Audio Compression Library" on page 60 for more information about the DV standalone routines. DM_AUDIO_DV is the identification value used with **dmACSetParams()** as described in "Configuring a Converter Instance."

Table 3-1 lists the functions of the Digital Media Audio Conversion Library API. These functions are described more fully in the sections mentioned in the Description column. More details about the specific functions, such as the errors they return, can be found by looking at the man pages also mentioned in the Description column.

Table 3-1 Digital Media Audio Conversion API

Function	Parameters	Description
DMstatus **dmACConvert**	(DMaudioconverter *converter*, void **inbuffer*, void **outbuffer*, int **in_amount*, int **out_amount*)	Convert the audio data format, sampling rate, and compression. See "Converting Data Using a Converter Instance" on page 59 and the dmACConvert(3dm) man page for more details.
DMstatus **dmACCreate**	(DMaudioconverter **converter*)	Create a DMaudioconverter handle to use for audio format conversion. See "Creating a Converter Instance" on page 47 and the dmACCreate(3dm) man page for more details.
DMstatus **dmACDestroy**	(DMaudioconverter *converter*)	Destroy a DMaudioconverter handle used for audio format conversion. See "Destroying a Converter Instance" on page 60 and the dmACDestroy(3dm) man page for more details.
DMstatus **dmACReset**	(DMaudioconverter *converter*)	Reset a DMaudioconverter handle to its default state. See "Configuring a Converter Instance" on page 47 and the dmACReset(3dm) man page for more details.
DMstatus **dmACSetParams**	(DMaudioconverter *converter*, DMparams **sourceparams*, DMparams **destparams*, DMparams **conversionparams*)	Set the DMaudioconverter parameter values. See "Configuring a Converter Instance" on page 47 and the dmACSetParams(3dm) man page for more details.
DMstatus **dmACGetParams**	(DMaudioconverter *converter*, DMparams **sourceparams*, DMparams **destparams*, DMparams **conversionparams*)	Get the DMaudioconverter parameter values. See "Configuring a Converter Instance" on page 47 and the dmACGetParams(3dm) man page for more details.

Using the Audio Conversion API

When an application uses the Digital Media Audio Conversion API to perform conversions, it creates a *converter instance* (also referred to simply as a *converter*). A converter instance includes an input buffer, an output buffer, and state information in the form of parameters that describe the input data, output data, and conversion process.

A converter instance can be viewed as a pipeline. At the input end of the pipeline is data in the unconverted format. At the output end is data in the requested format. The pipeline processing is done by the codec and/or converter. When multiple transformations are required, the converter makes sure that they are done in an order that best maintains the quality of the data.

The next four sections describe the steps an application follows when performing an audio conversion:

- "Creating a Converter Instance" on page 47
- "Configuring a Converter Instance" on page 47
- "Converting Data Using a Converter Instance" on page 59
- "Destroying a Converter Instance" on page 60

To use the digital media conversion libraries to create a converter instance, you must link your application with *libdmedia.so*. To use the Audio Conversion API, include these header files:

```
#include <dmedia/dm_audioconvert.h>
#include <dmedia/dm_audioutil.h>
```

Creating a Converter Instance

To create an audio converter instance, use **dmACCreate()**:

```
DMstatus dmACCreate (DMaudioconverter* converter)
```

This function creates and initializes *converter*, a handle to a DMaudioconverter instance. All of the Audio Conversion Library functions use this handle, which is declared as follows:

```
typedef struct _DMaudioconverter *DMaudioconverter;
```

Note: All of the Audio Conversion Library functions return a DMstatus value of DM_SUCCESS if they succeed, DM_FAILURE if not. After a receiving a DM_FAILURE, your application can call the function **dmGetErrorForPID()** or **dmGetError()** to retrieve an error message and error number. See the DMPG for more information on error handling.

Configuring a Converter Instance

Once a converter instance has been created, it must be configured. An application does this by using DMparams data structures to specify a set of source parameters, a set of destination parameters, and, optionally, a set of conversion parameters. See the DMPG for more information on DMparams.

The Audio Conversion Library lets you:

- Configure an audio converter by setting the source, destination, and conversion parameters with **dmACSetParams()**:

  ```
  DMstatus dmACSetParams (DMaudioconverter converter,
                          DMparams *sourceparams,
                          DMparams *destparams,
                          DMparams *conversionparams)
  ```

  – *converter* is a DMaudioconverter handle created by a previous call to **dmACCreate()**.

  – *sourceparams* and *destparams* point to DMparams structures that describe the formats of the audio data prior to and after conversion.

  – *conversionparams* points to a DMparams structure that contains parameters specific to the conversion process.

destparams and *conversionparams* are optional and may be set to NULL. The output format defaults to the input values, with the exception of compression (defaults to uncompressed) and byte order (defaults to big-endian).

- Retrieve the source, destination, and conversion parameter settings of a configured audio converter with **dmACGetParams()**:

```
DMstatus dmACGetParams (DMaudioconverter converter,
                        DMparams *sourceparams,
                        DMparams *destparams,
                        DMparams *conversionparams)
```

  – *converter* is a DMaudioconverter handle created by a previous call to **dmACCreate()**.

  – *sourceparams* and *destparams* point to DMparams structures that describe the formats of the audio data prior to and after conversion.

  – *conversionparams* points to a DMparams structure that contains parameters specific to the conversion process.

  Any parameter list pointer may be set to NULL if the application does not need to retrieve that parameter set.

- Reset an audio converter's internal state with **dmACReset()**:

  `DMstatus dmACReset (DMaudioconverter` *converter*`)`

  converter is a DMaudioconverter handle created by a previous call to **dmACCreate()**.

  dmACReset() clears out any internal buffers and/or state associated with compression, decompression, and/or rate conversion, and resets all counters, essentially returning the converter to the state it was in immediately following a call to **dmACSetParams()**. This might be desirable in a case where an existing stream of data is interrupted and replaced with a new stream, and you want to be certain that all of the previously processed data was cleared.

The rest of this section consists of the following:

Source and Destination Parameters

The source parameters describe the data to be converted and are contained in the DMparams structure indicated by *sourceparams*. The destination parameters describe the output audio data and are contained in the DMparams structure indicated by *destparams*. Any excess or unrecognized parameters in *sourceparams* or *destparams* are ignored.

Table 3-2 lists the source and destination parameters, their data types, legal values, and any pertinent information about their use. There are no default values for source parameters—they must be specified.

Table 3-2 Source and Destination Parameters

Parameter (Data Type)	Legal Values	Other Info
DM_AUDIO_BYTE_ORDER (DM_TYPE_ENUM)	DM_AUDIO_BIG_ENDIAN DM_AUDIO_LITTLE_ENDIAN	**As source parameter:** No default value. Must be specified. **As destination parameter:** Defaults to DM_AUDIO_BIG_ENDIAN
DM_AUDIO_CHANNELS (DM_TYPE_INT)	The number of audio channels. An integer value greater than 0.	**As source parameter:** No default value. Must be specified. **As destination parameter:** Defaults to source value.
DM_AUDIO_COMPRESSION (DM_TYPE_STRING)	DM_AUDIO_UNCOMPRESSED or DM_AUDIO_DV.	**As source parameter:** No default value. Must be specified. **As destination parameter:** Defaults to DM_AUDIO_UNCOMPRESSED.
DM_AUDIO_FORMAT (DM_TYPE_ENUM)	DM_AUDIO_TWOS_COMPLEMENT DM_AUDIO_UNSIGNED DM_AUDIO_FLOAT DM_AUDIO_DOUBLE	**As source parameter:** No default value. Must be specified. **As destination parameter:** Defaults to source value.

Table 3-2 (continued) Source and Destination Parameters

Parameter (Data Type)	Legal Values	Other Info
DM_AUDIO_RATE (DM_TYPE_FLOAT)	The audio sampling rate in Hz. A float value greater than 0.0.	**As source parameter:** No default value. Must be specified. **As destination parameter:** Defaults to source value.
DM_AUDIO_WIDTH (DM_TYPE_INT)	The width of the data in bits. An integer value between 1 and 32, inclusive.	**As source parameter:** No default value. Must be specified. **As destination parameter:** Defaults to source value.

PCM Mapping Parameters

Table 3-3 lists the four parameters for PCM mapping whose values, although they can be set for source data, are normally specified only for destination data. These parameters are useful when you need to convert integer data to floating point or vice versa. The parameters specify the numeric mapping of one format to another, using one PCM value that corresponds to zero voltage and a differential value that corresponds to full voltage. See the afIntro(3dm) man page for a more detailed discussion of the PCM mapping model.

The function **dmACSetParams()** automatically calculates default input and output PCM parameters from the input and output data format specifications. (For instance, the default values for 16 bit, two's complement data are 0.0 for intercept, 32767.0 for maxclip, -32768.0 for minclip, and 32767.0 for slope.)

Your application needs to set these parameters only if it has special mapping requirements, such as input data with a fixed offset like a DC bias. If your application sets any of these parameters, it must set all of them.

Table 3-3 Parameters for PCM Mapping

Parameter (Date Type)	Description	Other Info
DM_AUDIO_PCM_MAP_INTERCEPT (DM_TYPE_FLOAT)	The zero voltage PCM value.	As source parameter: Rarely used. As destination parameter: Default is calculated from the output data format specifications.
DM_AUDIO_PCM_MAP_MAXCLIP (DM_TYPE_FLOAT)	Clip all PCM values to this maximum value.	As source parameter: Rarely used. As destination parameter: Default is calculated from the output data format specifications.
DM_AUDIO_PCM_MAP_MINCLIP (DM_TYPE_FLOAT)	Clip all PCM values to this minimum value.	As source parameter: Rarely used. As destination parameter: Default is calculated from the output data format specifications.
DM_AUDIO_PCM_MAP_SLOPE (DM_TYPE_FLOAT)	The full voltage PCM value.	As source parameter: Rarely used. As destination parameter: Default is calculated from the output data format specifications.

Compression Parameters

Compression parameters that modify DM_AUDIO_DV and their values are outlined in the following sections:

- "Compression Parameters Common to All Codecs" on page 52
- "DV Audio Parameters" on page 53

When working with codecs, keep these points in mind:

- All format parameters interact with the codec parameters. For instance, some codecs work only at certain rates or channel counts. Refer to the relevant man pages for each codec for more information about using specific parameters.
- No cross-compression is supported; that is, input and output cannot both be compressed. Applications must create two converter instances and handle the intermediate uncompressed buffer themselves.
- The particular codec being used cannot be changed during the lifetime of the converter instance. If this needs to be done, destroy and recreate the converter.
- All other parameters may be changed without recreating the converter. The Digital Media Audio Conversion Library is designed to allow real-time tracking of changes in audio input and output format, such as sampling rate and number of interleaved channels.

Compression Parameters Common to All Codecs

The parameters listed in Table 3-4 apply to all codecs and can be queried using the **dmACGetParams()**.

Ordinarily, when your application is using the Digital Media Audio Conversion library to control the standalone codec routines, these parameters are solely informational—your application uses the buffer length parameters instead (see "Buffer Length Parameters" on page 56). For DV and MPEG1 codecs operating in decode mode, however, these parameters must be set (see "DV Audio Parameters" on page 53). Table 3-4 gives query parameters for all codecs.

Table 3-4 Query Parameters for All Codecs

Parameter (Data Type)	Description
DM_AUDIO_CODEC_FILTER_DELAY (DM_TYPE_INT)	Indicates delay, in sample frames, introduced by compression and decompression processing. This is usually different for compression and decompression.
DM_AUDIO_CODEC_FRAMES_PER_BLOCK (DM_TYPE_INT)	Specifies how many sample frames are contained in each compressed data block.
DM_AUDIO_CODEC_MAX_BYTES_PER_BLOCK (DM_TYPE_INT)	Indicates the maximum number of bytes that will make up a compressed data block.

DV Audio Parameters

The DV Audio codec implements DV and DVCPRO audio compression and decompression.

Table 3-5 lists the codec-specific parameters used with the DV Audio codec. See the dmDVAudioEncode(3dm) man page and *dmedia/dm_audioutil.h* for more information on using these and other source and destination parameters with this codec.

Table 3-5 DV Audio Parameters

Parameter (Data Type)	Legal Values, Default Values	Encoding, Decoding, or Query
DM_AUDIO_MEDIUM (DM_TYPE_ENUM)	DM_AUDIO	Encoding and decoding
DM_DVAUDIO_FORMAT (DM_TYPE_INT)	DM_DVAUDIO_NTSC DM_DVAUDIO_PAL	Encoding only
DM_DVAUDIO_LOCK_MODE (DM__TYPE_INT)	Non-zero if audio is locked according to the DV audio specification, zero if it is not locked (default = locked)	Encoding only
DM_DVAUDIO_TYPE (DM_TYPE_INT)	DM_DVAUDIO_DV DM_DVAUDIO_DVCPRO (default = DM_DVAUDIO_DV)	Encoding only

Table 3-5 (continued) DV Audio Parameters

Parameter (Data Type)	Legal Values, Default Values	Encoding, Decoding, or Query
DM_DVAUDIO_CHANNEL_MODE (DM_TYPE_INT)	DM_DVAUDIO_SD_2CH DM_DVAUDIO_SD_4CH (Not currently supported for encoding)	Query only
DM_DVAUDIO_CHANNEL_POLICY (DM_TYPE_INT) See *dmedia/dm_audioutil.h* for more information about this parameter.	*With DM_DVAUDIO_SD_2CH mode:* DM_DVAUDIO_SD_2CH_STEREO DM_DVAUDIO_SD_2CH_2CH_MONO * DM_DVAUDIO_SD_2CH_MONO *With DM_DVAUDIO_SD_4CH mode:* DM_DVAUDIO_SD_4CH_STEREO_STEREO * DM_DVAUDIO_SD_4CH_STEREO_2CH_MONO * DM_DVAUDIO_SD_4CH_STEREO_1CH_MONO * DM_DVAUDIO_SD_4CH_STEREO ** DM_DVAUDIO_SD_4CH_2CH_MONO_STEREO * DM_DVAUDIO_SD_4CH_4CH_MONO * DM_DVAUDIO_SD_4CH_3CH_MONO_1 * DM_DVAUDIO_SD_4CH_2CH_MONO_1 * DM_DVAUDIO_SD_4CH_1CH_MONO_STEREO * DM_DVAUDIO_SD_4CH_3CH_MONO_2 * DM_DVAUDIO_SD_4CH_2CH_MONO_2 * DM_DVAUDIO_SD_4CH_1CH_MONO * DM_DVAUDIO_SD_4CH_3_1_STEREO * DM_DVAUDIO_SD_4CH_3_0_STEREO_1CH_MONO * DM_DVAUDIO_SD_4CH_3_0_STEREO * DM_DVAUDIO_SD_4CH_2_2_STEREO * * Not currently supported for encoding.	Query only

Note: If you use the Digital Media Audio Conversion Library to control standalone DV Audio decoder functions, your application must call **dmDVAudioHeaderGetParams()** to get the parameter values to pass to **dmDVAudioDecoderSetParams()**. See the dm_dv(3dm) and dmDVAudioHeaderGetParams(3dm) man pages for more details (and sample code) on this process.

Conversion Parameters

The conversion parameters, which modify the codec settings and other aspects of the conversion process, are contained in the DMparams structure indicated by *conversionparams*.

The five categories of conversion parameters are discussed in these subsections:

Processing Mode Parameter

The DM_AUDIO_PROCESS_MODE (DM_TYPE_INT) parameter determines the converter's processing mode. It can also be used to set the processing mode when both the input and output data are uncompressed. Its legal values are DM_AUDIO_PROCESS_PULL and DM_AUDIO_PROCESS_PUSH:

- In *pull* mode, the application requests a fixed number of frames from the converter, and the converter determines how much input data to consume in order to satisfy this request. Decompression requires pull mode, and the application must use the **dmACConvert()** argument *out_amount* (see "Converting Data Using a Converter Instance" on page 59) to specify how many frames of uncompressed data the converter instance should put in the output buffer of **dmACConvert()**.
- In *push* mode, the application gives the converter a fixed number of frames, and the converter produces a (possibly variable) amount of output, depending on the conversion being done. Compression requires push mode, and the application must use the **dmACConvert()** argument *in_amount* (see "Converting Data Using a Converter Instance" on page 59) to specify how many frames of data need to be compressed.

Buffer Length Parameters

The three buffer length parameters are used to determine appropriate sizes for the input or output buffers. Your application must specify them only during compression, decompression, or rate conversion because the input and output buffer lengths are equal at all other times.

When performing compression, decompression, or rate conversion, your application must follow these steps:

1. Set DM_AUDIO_MAX_REQUEST_LEN equal to the maximum amount of data the application will push to or pull from the converter, using **dmParamsSetInt()**. (For information on DMparams, see the *Digital Media Programming Guide*.)
2. Call **dmACSetParams()** with the DMParams list from the call to **dmParamsSetInt()**.
3. Call **dmACGetParams()** to obtain the correct values for DM_AUDIO_MIN_INPUT_LEN and DM_AUDIO_MIN_OUTPUT_LEN.

If the converter is operating in pull mode, DM_AUDIO_MIN_INPUT_LEN specifies the minimum number of frames or bytes the converter requires in the input buffer. If the converter is operating in push mode, DM_AUDIO_MIN_OUTPUT_LEN specifies the minimum number of frames or bytes the converter requires in the output buffer. Your application can use buffers equal to or longer than the specified values. See “Processing Mode Parameter” on page 55 for more information on push and pull modes.

Table 3-6 lists the buffer length parameters and their data types and legal values.

Table 3-6 Buffer Length Parameters

Parameter (Data Type)	Legal Values
DM_AUDIO_MAX_REQUEST_LEN (DM_TYPE_INT)	Integer value greater than 0
DM_AUDIO_MIN_INPUT_LEN (DM_TYPE_INT)	Integer value
DM_AUDIO_MIN_OUTPUT_LEN (DM_TYPE_INT)	Integer value

Dithering Parameter

The DM_AUDIO_DITHER_ALGORITHM parameter (DM_TYPE_INT) is used only when data is converted from a larger to a smaller data type, such as when converting from floating point to 16-bit integer samples. The dithering algorithm is applied to reduce the quantization error distortion inherent in reducing resolution.

The possible values for this parameter are DM_AUDIO_DITHER_NONE (default) and DM_AUDIO_DITHER_LSB_TPDF. The latter specifies the Least Significant Bit—Triangular Probability Density Function described by John Watkinson in *The Art of Digital Audio* (Focal Press, 1994).

Rate Conversion Parameters

Rate conversion parameters affect the rate conversion algorithm and are used only when the input and output sampling rates are not equal (see the dmAudioRateConverterSetParams(3dm) man page for more information). Table 3-7 lists the rate conversion parameters and their data types and legal values.

Table 3-7 Rate Conversion Parameter

Parameter (Data Type)	Legal Values
DM_AUDIO_RC_ALGORITHM (DM_TYPE_STRING)	DM_AUDIO_RC_JITTER_FREE (default) (jitter-free interpolation/decimation) DM_AUDIO_RC_POLYNOMIAL_ORDER_1 (first order polynomial, linear interpolation) DM_AUDIO_RC_POLYNOMIAL_ORDER_3 (third order polynomial)
DM_AUDIO_RC_JITTER_FREE_STOPBAND_ATTENUATION (DM_TYPE_FLOAT) Applies only to JITTER_FREE.	DM_AUDIO_RC_JITTER_FREE_STOPBAND_ ATTENUATION_78_DB DM_AUDIO_RC_JITTER_FREE_STOPBAND_ ATTENUATION_96_DB DM_AUDIO_RC_JITTER_FREE_STOPBAND_ ATTENUATION_120_DB (default = DM_AUDIO_RC_JITTER_FREE_ STOPBAND_ATTENUATION_78_DB)

Table 3-7 (continued) Rate Conversion Parameter

Parameter (Data Type)	Legal Values
DM_AUDIO_RC_JITTER_FREE_TRANSITION_BANDWIDTH (DM_TYPE_FLOAT) Applies only to JITTER_FREE.	DM_AUDIO_RC_JITTER_FREE_TRANSITION_BANDWIDTH_1_PERCENT DM_AUDIO_RC_JITTER_FREE_TRANSITION_BANDWIDTH_10_PERCENT DM_AUDIO_RC_JITTER_FREE_TRANSITION_BANDWIDTH_20_PERCENT
DM_AUDIO_RC_FLUSH_VALUE (DM_TYPE_FLOAT)	The value used to flush the rate converter during dmAC flush mode (to avoid an audible click).

Channel Conversion Parameter

DM_AUDIO_CHANNEL_MATRIX (DM_TYPE_FLOAT_ARRAY), the channel conversion or channel matrix parameter, determines the manner in which the input channels are mapped into the output channels. The matrix is a one-dimensional array representing a two-dimensional array in row-major order, where each row represents an output channel and each column represents an input channel. Its legal values are a DMfloatarray of double-precision floating point numbers. For a detailed explanation, see the afSetChannelMatrix(3dm) man page.

Figure 3-1 illustrates a channel conversion from four input channels to two output channels. Inputs 1 and 2 are split between the outputs.

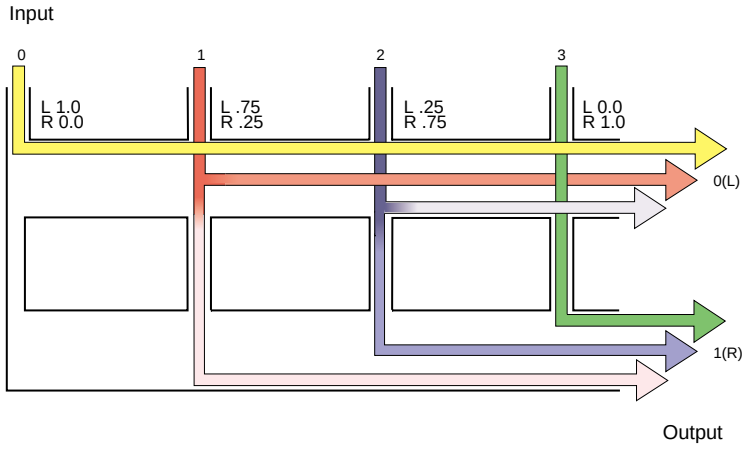

Figure 3-1 Channel Conversion

Converting Data Using a Converter Instance

Use **dmACConvert()** to convert the audio data's format, sampling rate, and compression. This function performs the data format, sampling rate, and compression or decompression specified by the previous call to **dmACSetParams()**:

```
DMstatus dmACConvert (DMaudioconverter converter,
                      void *inbuffer,
                      void *outbuffer,
                      int *in_amount,
                      int *out_amount)
```

- *converter* is a handle to an audio converter previously created with **dmACCreate()** and configured with **dmACSetParams()**.
- *inbuffer* and *outbuffer* point to the buffers that contain the audio data prior to and after conversion. As described in "Configuring a Converter Instance" on page 47 your application may need to determine the number of frames or bytes these buffers hold using the DM_AUDIO_MIN_INPUT_LEN or DM_AUDIO_MIN_OUTPUT_LEN parameters.
- *in_amount* points to an integer containing the number of frames (bytes if the input is compressed) of input data available to the converter instance. This can be any value greater than 0. In pull mode, **dmACConvert()** resets this value to the number of frames (bytes) read from *inbuffer* by the converter. (To review the push and pull modes, see the discussion in "Processing Mode Parameter" on page 55.)
- *out_amount* is a pointer to an integer containing the number of frames (bytes if the output is compressed) of converted data your application is requesting from the converter. In push mode, the initial value is ignored. After processing the data, **dmACConvert()** resets *out_amount* to the number of frames (bytes) actually placed into *outbuffer*. If the conversion involves rate conversion, compression, or decompression, this value can vary significantly from the *in_amount* value. *out_amount* can be zero even when *in_amount* was positive. This is because the converter must often do internal buffering to allow the application to request arbitrary amounts of data.

Note: Flushing the converter is advisable in certain situations. Many conversion operations, such as rate conversion, cause the converter to store portions of the audio signal in internal buffers. When your application has no more data to pass to the converter, it should flush these buffers into the output buffer to avoid losing the end of the audio stream. To flush these buffers, call **dmACConvert()** with *inbuffer* set to NULL until **out_amount* equals 0. This action usually takes only one call.

Destroying a Converter Instance

The Audio Conversion library provides the function **dmACDestroy()** to destroy an audio converter instance:

```
DMstatus dmACDestroy (DMaudioconverter converter)
```

The function frees the memory associated with the DMaudioconverter handle. The handle is not valid after this call returns.

DV Audio Compression Library

The DV Audio codec implements DV and DVCPRO audio compression and decompression. Table 3-8 summarizes the calls. For an example, see `recdvaudio.c` and `playdvaudio.c` in `/usr/share/src/dmedia/dvaudiotest`.

Table 3-8 DV Audio Library API

Function	Parameters	Description
DMstatus **dmDVAudioDecode**	(DMDVaudiodecoder *decoder*, void **inbuf*, void **outbuf*, int **nsamples*)	Do DV and DVCPRO audio decompression. See also dmDVAudioDecode(3dm).
DMstatus **dmDVAudioDecoderCreate**	(DMDVaudiodecoder **decoder*)	Allocate new DMDVaudiodecoder structure. See also dmDVAudioDecoderCreate(3dm).
DMstatus **dmDVAudioDecoderDestroy**	(DMDVaudiodecoder *decoder*)	Deallocate a DMDVaudiodecoder structure. See also dmDVAudioDecoderDestroy(3dm).
DMstatus **dmDVAudioDecoderGetParams**	(DMDVaudiodecoder *decoder*, DMparams **params*)	Get DV audio decoder parameter values. See also dmDVAudioDecoderGetParams(3dm).
DMstatus **dmDVAudioDecoderReset**	(DMDVaudiodecoder *decoder*)	Fill decoder internal buffers with zeros. See also dmDVAudioDecoderReset(3dm).
DMstatus **dmDVAudioDecoderSetParams**	(DMDVaudiodecoder *decoder*, DMparams **params*)	Set DV Audio decoder parameter values. See also dmDVAudioDecoderSetParams(3dm).

Table 3-8 (continued) DV Audio Library API

Function	Parameters	Description
DMstatus **dmDVAudioEncode**	(DMDVaudioencoder *encoder*, void **inbuf*, void **outbuf*, int **nsamples*)	Do DV and DVCPRO audio compression. See also dmDVAudioEncode(3dm).
DMstatus **dmDVAudioEncoderCreate**	(DMDVaudioencoder **encoder*)	Allocate new DMDVaudioencoder structure. See also dmDVAudioEncoderCreate(3dm).
DMstatus **dmDVAudioEncoderDestroy**	(DMDVaudioencoder *encoder*)	Deallocate DMDVaudioencoder structure. See also dmDVAudioEncoderDestroy(3dm).
DMstatus **dmDVAudioEncoderGetFrameSize**	(DMDVaudioencoder *handle*, int **numSampFrame*)	Get number of audio samples required for DIF frame.
DMstatus **dmDVAudioEncoderGetParams**	(DMDVaudioencoder *encoder*, DMparams **params*)	Get DV audio encoder parameter values. See also dmDVAudioEncoderGetParams(3dm).
DMstatus **dmDVAudioEncoderReset**	(DMDVaudioencoder *encoder*)	Fill encoder internal buffers with zeros. See also dmDVAudioEncoderReset(3dm).
DMstatus **dmDVAudioEncoderSetParams**	(DMDVaudioencoder *encoder*, DMparams **params*)	Set DV audio encoder parameter values. See also dmDVAudioEncoderSetParams(3dm).
DMstatus **dmDVAudioHeaderGetParams**	(void **dif*, DMparams **params*, int **numSampFrames*)	Get DV audio decoder parameter values based on the information in a DIF block handed to the routine. See also dmDVAudioHeaderGetParams(3dm).

Audio Rate Conversion Library

The Audio Rate Conversion library enables the sampling rate conversion of single-channel, 32-bit, floating point audio data. Table 3-9 summarizes the functions in this library.

Table 3-9 Audio Rate Conversion Library API

Function	Parameters	Description
DMstatus **dmAudioRateConvert**	(DMaudiorateconverter *handle*, float **inbuf*, float **outbuf*, int *inlen*, int **numout)*	Convert the data sampling rate. See also dmAudioRateConvert(3dm).
DMstatus **dmAudioRateConverterCreate**	(DMaudiorateconverter **converter*)	Allocate a new DMaudiorateconverter structure. See also dmAudioRateConverterCreate(3dm).
DMstatus **dmAudioRateConverterDestroy**	(DMaudiorateconverter *handle*)	Deallocate an DMaudiorateconverter structure. See also dmAudioRateConverterDestroy(3dm).
DMstatus **dmAudioRateConverterGetParams**	(DMaudiorateconverter *handle*, DMparams **params)*	Get the parameter values of a DMaudiorateconverter structure. See also dmAudioRateConverterGetParams(3dm).
DMstatus **dmAudioRateConverterSetParams**	(DMaudiorateconverter *handle*, DMparams **params)*	Set the parameter values of a DMaudiorateconverter structure. See also dmAudioRateConverterSet Params(3dm).
DMstatus **dmAudioRateConverterReset**	(DMaudiorateconverter *handle*, float *resetval)*	Fill the internal buffers of a DMaudiorateconverter structure with a constant value. See also dmAudioRateConverterReset(3dm).

DIVO/DIVO-DVC I/O Panel Connectors

This appendix summarizes hardware specifications for the DIVO and DIVO-DVC option boards, in these sections

DIVO/DIVO-DVC Connectors

Table A-1 summarizes return loss for the **IN LINK A**, **IN LINK B**, and **GEN IN** connectors.

Table A-1 Return Loss for DIVO/DIVO-DVC Video and Genlock Channels

Channel	Value
IN LINK A, IN LINK B	> 15 dB @ 270 MHz
GEN IN	> 40 dB @ 6 MH

Table A-2 summarizes output characteristics for the **OUT LINK A** and **OUT LINK B** connectors.

Table A-2 Characteristics for DIVO/DIVO-DVC Digital Video Out Channels

Characteristic	Value
Amplitude	800 mV +/-10%
Rise/fall time	.75 nsec to 1.5 nsec
Overshoot	<10% p-p
Alignment jitter	<740 ps p-p

Table A-3 summarizes the use of **LINK A** and **LINK B** connectors for 4:2:2:4 mode. If LINK B is not used in 4:2:2:4 format, the resulting format is 4:2:2. The **LINK A** connector carries 10-bit wide UVY information; the **LINK B** connector carries 10-bit alpha. Usage is similar for 10-bit RGBA.

Table A-3 Usage for LINK A and LINK B in 4:2:2:4 Mode

Sample	LINK A	LINK B
0	Cb_0	x
1	Y_0	A_0
2	Cr_0	x
3	Y_1	A_1

Table A-4 summarizes the use of **LINK A** and **LINK B** connectors for 4:4:4:4 mode. The **LINK A** connector carries a 4:2:2 sampled portion of 10-bit wide UVY; the **LINK B** connector carries the remaining 10-bit UV samples and 10-bit alpha. Usage is similar for 10-bit RGBA.

Table A-4 Usage for LINK A and LINK B in 4:4:4:4 Mode

Sample	LINK A	LINK B
0	Cb_0	Cb_1
1	Y_0	A_0
2	Cr_0	Cr_1
3	Y_1	A_1

Genlock

The **GEN OUT** and **GEN IN** connectors make up a passive genlock loopthrough connection. If you attach a cable to one **GEN** connector, you must attach to the other **GEN** connector either a 75-ohm BNC terminator or a cable to other equipment accepting analog sync. If another cable is connected, it must ultimately be terminated.

GPI Interface

For each video pipe, the General Purpose Interface (GPI) provides two channels of input and output trigger signal pairs. This section consists of the following:

GPI Connectors

The panel on the DIVO and DIVO-DVC option boards has two GPI connectors, each associated with one of the serial digital video ports (two transmit and two receive channels each). Figure A-1 points out the GPI connectors on the DIVO/DIVO-DVC panel.

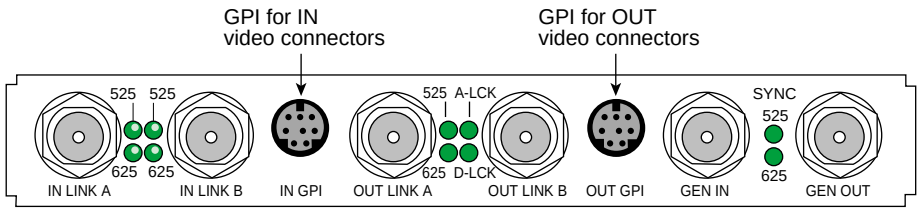

Figure A-1 GPI Connectors

Figure A-2 shows pinouts for the GPI; the information is applicable for both the **IN GPI** and **OUT GPI** connectors. In this figure, CCT denotes contact closure transmit, and CCR denotes contact closure receive.

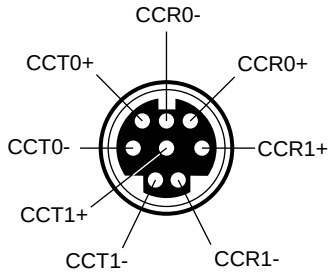

Figure A-2 GPI Pinouts

Each +/- signal pair of the same name applies to one channel of either a receive or transmit optical device. Table A-5 gives the meaning of the pins in Figure A-2.

Table A-5 GPI Pinouts

Pin	Symbol	Name	Channel
8	CCT0+	Contact Closure Transmit +	0
4	CCT0-	Contact Closure Transmit -	0
5	CCT1+	Contact Closure Transmit +	1
2	CCT1-	Contact Closure Transmit -	1
6	CCR0+	Contact Closure Receive +	0
7	CCR0-	Contact Closure Receive -	0
3	CCR1+	Contact Closure Receive +	1
1	CCR1-	Contact Closure Receive -	1

GPI Transmitter

GPI contact closure transmit (CCT) outputs use an optically coupled solid-state relay (SSR) to provide a means of electrical isolation for destination equipment. The GPI transmitter is triggered by setting VL_GPI_STATE (see "Using VL_GPI_STATE" on page 37 in Chapter 2), which forward-biases the internal LED, which in turn drives the output MOSFET, closing the contacts of the SSR.

When the GPI trigger is off, a high resistance exists between the CCT+/- terminals. When the GPI is on (triggered by setting VL_TRANSFER_TRIGGER to VL_TRIGGER_GPI (see "Using VL_TRANSFER_TRIGGER" on page 38 in Chapter 2), a low resistance exists between the terminals.

Figure A-3 and Table A-6 show electrical specifications for the GPI transmitter.

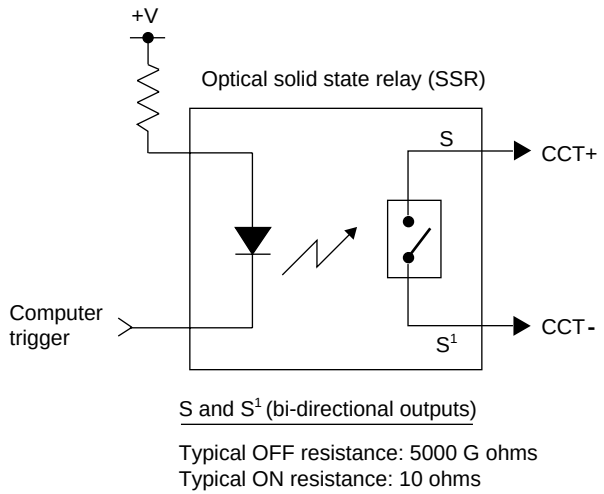

Figure A-3 GPI Transmitter Electrical Specifications

Table A-6 GPI Transmitter Electrical Specifications

Parameter	Value
On resistance	10 ohms typical, 15 ohms maximum
Off resistance	5000 G ohms
Current limit	360 mA typical, 460 mA maximum
Output capacitance	60 pF
Continuous DC load current	180 mA
Output power dissipation	600 mW
Isolation voltage	3750 V rms

The GPI transmitter can be interfaced to the destination equipment by tying the CCT- terminal to GND and using the CCT+ terminal as a current sink. The input device can consist of a logic device with active pullup, an optoisolator LED with series-limiting resistor, or relay primary with series-limiting resistor.

The GPI transmitter's logic sense can be swapped (inverted) by tying the CCT+ terminal to the logic power supply (VCC) of the destination equipment and using the CCT- terminal to drive the input of the receiving device.

GPI Receiver

GPI contact closure receive (CCR) inputs use an optical isolator device to provide a means of electrical isolation from source equipment. The device consists of a bidirectional input LED optically coupled to a bipolar transistor. A voltage pulse applied across the CCR+/- pins causes the LED to become forward-biased and to produce a GPI trigger to the computer.

Figure A-4 shows electrical specifications for the GPI receiver.

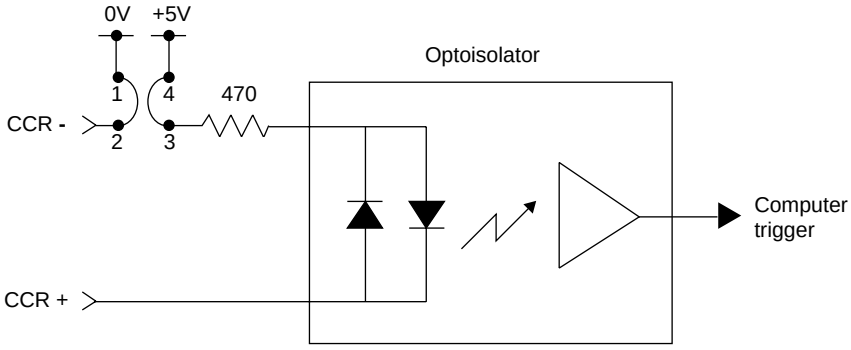

Figure A-4 GPI Receiver (Switch Closure) electrical Specifications

Table A-7 summarizes electrical specifications for the GPI receiver optoisolator.

Table A-7 GPI Receiver Input Optoisolator Electrical Specifications

Parameter	Value
Forward voltage (V_F)	1.55 V, 1.2 V typical (IF = 10 mA)
Continuous forward current (I_F)	30 mA
Peak forward current	1000 mA (10 us duration, 1% DC)
Reverse current (I_R)	0.1 uA, 100 uA maximum (V_R = 6 V)
Isolation surge voltage (V_{10})	2500 VAC_{RMS} (t = 1 min)

The +5 V power supply and ground of the DIVO/DIVO-DVC board are not electrically isolated from the chassis of the source equipment. The GPI receiver can be interfaced to the source equipment by tying the CCR+ and CCR- terminals across the output terminals of an optoisolator, solid-state relay, or any device that acts like a single-pole contact switch.

The GPI receiver is set to switch closure mode, which creates a digital pulse. A GPI trigger is generated as long as the source switch is closed.

Note: Polarity of the CCR+/- signals must be observed for the source equipment.

Setting Up DIVO and DIVO-DVC for Your Video Hardware

This appendix illustrates how to attach video equipment to connectors on the DIVO/DIVO-DVC I/O panel and how to use the video control panel *vcp* to set the option to match your installation.

This appendix explains

Figure B-1 shows connectors on the DIVO/DIVO-DVC I/O panel.

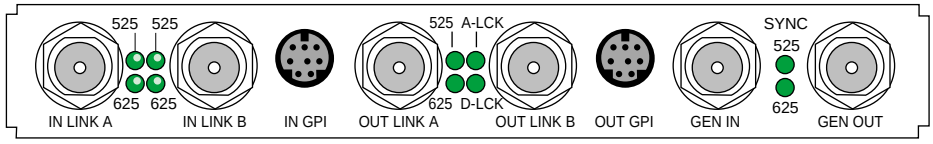

Figure B-1 DIVO/DIVO-DVC Ports

Setting Up Digital Source Video

The DIVO/DIVO-DVC panel has two 10-bit digital video input ports (**IN LINK A** and **IN LINK B**) for equipment that complies with the CCIR 601 standard. The ports can be configured for 4:4:4:4 or 4:2:2:4 dual-link mode; for 4:2:2 single-link mode, ignore the alpha:

- In 4:4:4:4 mode, Link A carries Y plus the U and V from even-numbered sample points; Link B carries alpha plus the U and V from odd-numbered sample points.
- In 4:2:2:4 mode, Link A carries Y plus the U and V from all sample points; Link B carries alpha only.

To set up the option for a digital video source, follow these steps:

1. Connect a video device to IN LINK A and, if you are using a device for alpha key data, also **IN LINK B**. If you use only one input, it must be **IN LINK A**.
2. Call up the panel:

   ```
   /usr/sbin/vcp
   ```

3. In the Inputs(s): DIVO Digital Video Source section of the control panel *vcp* for the channel(s) you are using, select the format that matches your equipment, as shown in Figure B-2.

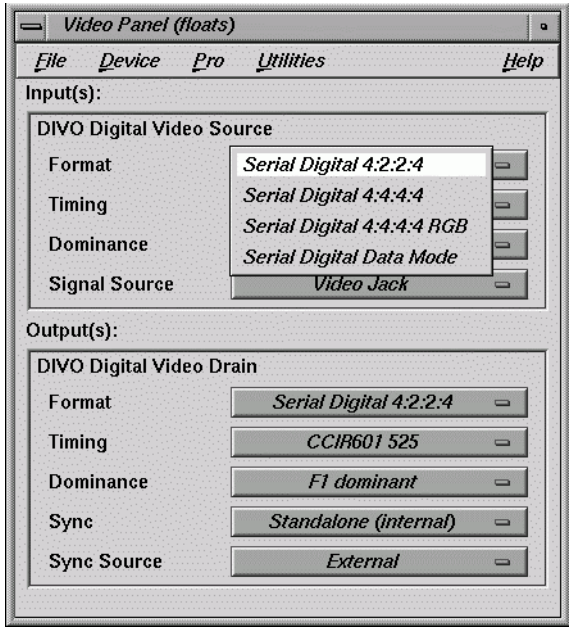

Figure B-2 Selecting Digital Input Video Format in vcp

4. In the Digital Video Source portion of the panel for the channel(s) you are using, select the timing that matches your equipment: CCIR 525 or CCIR 625.

Setting Up the Output (Drain)

To set up the digital video output, follow these steps:

1. Connect the video equipment to **OUT LINK A** and, if you are using a device for alpha key data, also **OUT LINK B**. If you use only one output, it must be **OUT LINK A**.
2. If necessary, call up the panel (**`/usr/sbin/vcp`**).
3. In the Output(s): DIVO Video Drain section of the control panel, select the format that matches your equipment, as shown in Figure B-3.

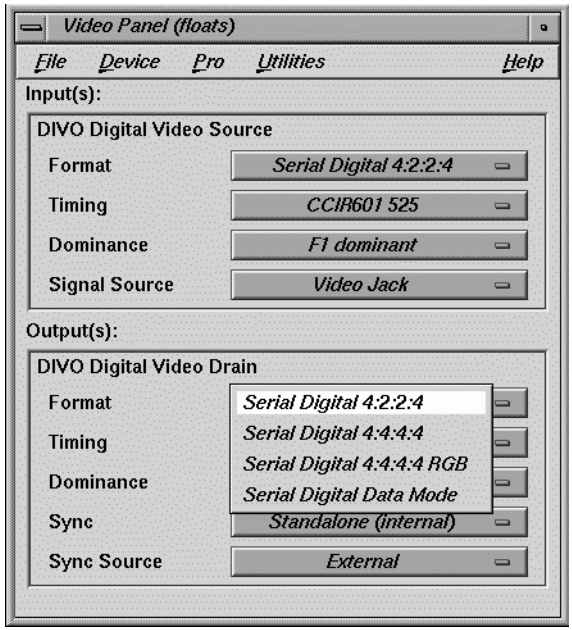

Figure B-3 Selecting Video Drain Format

4. Select the timing that matches your equipment: CCIR 525 or CCIR 625.
5. To set field dominance, at the “Input Timing” menu item select “F1 dominant” for the edit to occur on the nominal video field boundary, or “F2 dominant” for the edit to occur on the intervening field boundary. See “Setting Field Dominance” in Chapter 2 for more information on field dominance.

Setting Up Sync

This section explains

Setting Up Internal Sync

In the Output(s): DIVO Digital Video Drain section of the control panel, select the Sync format that matches your equipment:

- standalone (not synced to another device): select Standalone (internal)
- output sync to an external source connected to the genlock in: select Genlock

These two choices toggle, as shown in Figure B-4.

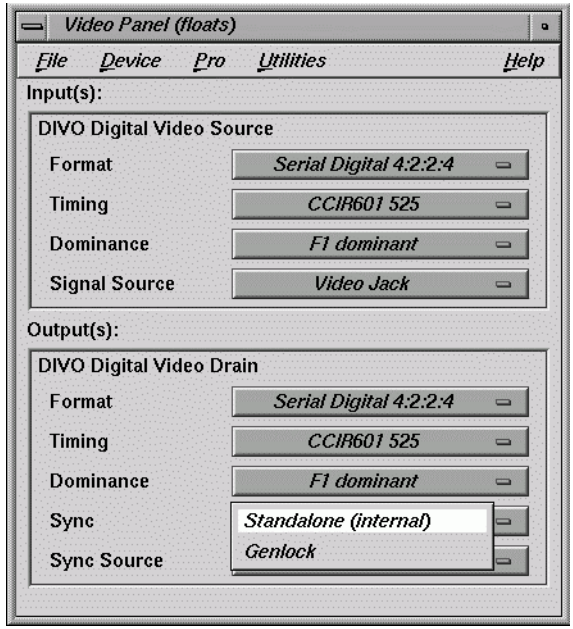

Figure B-4 Setting Standalone or Genlock Sync

Setting Up External Sync

To set up the /DIVO-DVC option for an external sync source, follow these steps:

1. Connect the sync source equipment to one of the following connectors:
   - the **GEN IN** BNC on the I/O panel, as diagrammed in Figure B-5.

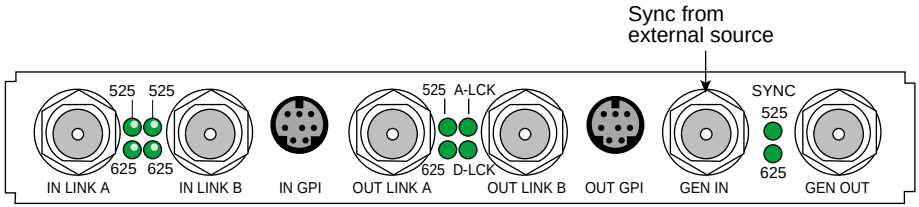

Figure B-5 GEN IN Port on the DIVO/DIVO-DVC I/O Panel

   - **IN LINK A** or **IN LINK B** (if the device is not already cabled to this connector)
2. If you are using the same signal for other equipment, attach a BNC cable to the **GEN OUT** BNC to loop the signal through the board. Make sure the final element in the chain is terminated.

   If DIVO/DIVO-DVC is the last element in the sync chain, make sure a terminator is attached to the **GEN OUT** BNC.
3. If necessary, call up the panel (**`/usr/sbin/vcp`**).
4. Select the appropriate setting in Output(s): Sync Source:
   - select External for a device connected to **GEN IN**
   - select Digital Input Link A or B if you are syncing to the device attached to **IN LINK A** or **IN LINK B**, respectively

Saving Settings

Once you have set values in *vcp* to match your installation, save them; they are written to */usr/etc/video/videod.defaults*. Select "Restore Settings" on the video control panel File menu to load the values in this file to *vcp*.

The last settings saved are automatically loaded every time the system is reinitialized. If the panel is running, current settings are in effect.

Note: You do not need to open the panel to put its settings into effect.

You can also use File menu choices to restore the factory defaults and close the panel.

Pixel Packings and Color Spaces

This appendix explains

DIVO/DIVO-DVC Pixel Packings

This section presents each packing used by the DIVO/DIVO-DVC hardware, 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, 4:4:4:4 or 4:2:2:4), 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, unless specified otherwise in the description of the packing

A color space

- determines the color in each component by specifying the color set (see Table C-2)
- specifies a canonical minimum and maximum value for each component, either full range or compressed range (headroom range); see “Color Spaces” on page 109 for an explanation

In most SGI 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. For the VL of DIVO/DIVO-DVC 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_444_8 (Figure C-1), indicates 3-byte quantities packed together in memory; the values are not padded out to 32-bit boundaries.

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_444_8

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 as appropriate. (See "Color Spaces" later in this appendix for more information.) 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 appendix 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.

- An x ("don't care") in a bit means the following:
  - Readers may get any garbage in this bit.
  - Writers can leave this bit as garbage.
- A 0 means the following:
  - Readers may assume this bit is zero.
  - Writers can leave this bit as garbage.
- An s indicates a padding bit that is a sign extension bit. For the DIVO or DIVO-DVC option, this convention applies only to the more significant bits in 12-bit and 13-bit packings with rightward orientation; that is, VL_PACKING_4444_12_in_16_R and VL_PACKING_4444_13_in_16_R.

- A p indicates a padding bit in the least significant bits of a left-justified 10-, 12-, or 13-bit word, such as VL_PACKING_R242_10_in_16_L or VL_PACKING_4444_13_in_16_L:
  - Readers can assume that the bits are replicated from the component found in the same word: With bits numbered starting with 0 for the least significant, there are n contiguous p bits to the right of the component. The p bits contain a copy of bits [9,9-n+1] of the component.
  - Writers can leave the p bits as garbage.

The DIVO or DIVO-DVC device can natively transfer data of all the packings shown in this appendix in real time.

Packings and Library Tokens

Following each packing diagram are comments and library tokens for that packing, listing, where applicable, the color set (RGBA or VYUA) and the library (VL, OpenGL, and DM) for each library token.

- 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" in this appendix for more information.
- For most packings, two indications are given for VL:
  - VL, new style, includes the packing control value and a color-space control value; for example, VL_PACKING_4_8 + VL_COLORSPACE_{CCIR,YUV}. For DIVO or DIVO-DVC, you set packing and color space separately for memory nodes. In contrast to Sirius Video, VL_COLORSPACE replaces VL_FORMAT on DIVO/DIVO-DVC memory nodes. The new definitions provide a more flexible way to specify memory layout of pixels and their color spaces.
  - VL, old style (for example, VL_PACKING_Y_8_P) is included for reference; these tokens are still recognized in case you are using programs for earlier SGI video options that include these. The old style is not recommended for new development.

Packing Naming Conventions

In packing tokens, the following applies:

- _L or _R appended to the end of a token with padding (0 bits) indicates 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).
- X before the numerical part of the token at the end of a token indicates a component order other than the standard (RGBA or ABGR, VYUA or AUYV); for example, VL_PACKING_X4444_5551, which uses ARGB order.
- 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 their 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 DIVO/DIVO-DVC 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 DIVO/DIVO-DVC Packings

Packing	Bits	Color Space
VL_PACKING_4_8	8	VYUA monochrome/luma only
VL_PACKING_R444_332	8	RGBA
VL_PACKING_444_332	8	RGBA
VL_PACKING_242_8	16	VYUA
VL_PACKING_R242_8	16	VYUA
VL_PACKING_X4444_5551	16	RGBA
VL_PACKING_444_5_6_5	16	RGBA
VL_PACKING_242_10	20	VYUA
VL_PACKING_R242_10	20	VYUA
VL_PACKING_444_8	24	RGBA/VYUA

Table C-1 (continued) DIVO/DIVO-DVC Packings

Packing	Bits	Color Space
VL_PACKING_R444_8	24	RGBA/VYUA
VL_PACKING_4444_6	24	RGBA/VYUA
VL_PACKING_4444_8	32	RGBA/VYUA
VL_PACKING_R4444_8	32	RGBA/VYUA
VL_PACKING_R0444_8	32	RGBA/VYUA
VL_PACKING_0444_8	32	RGBA/VYUA
VL_PACKING_4444_10_10_10_2	32	RGBA/VYUA
VL_PACKING_2424_10_10_10_2Z	32	VYUA
VL_PACKING_R2424_10_10_10_2Z	32	VYUA
VL_PACKING_242_10_in_16_L	32	VYUA
VL_PACKING_242_10_in_16_R	32	VYUA
VL_PACKING_R242_10_in_16_L	32	VYUA
VL_PACKING_R242_10_in_16_R	32	VYUA
VL_PACKING_DV	32	Y only
VL_PACKING_SDTI_DV	32	Cb/Y/Cr/Y
VL_PACKING_444_12	36	RGBA/VYUA (signed)
VL_PACKING_4444_12	48	RGBA/VYUA (signed)
VL_PACKING_444_10_in_16_L	48	RGBA/VYUA
VL_PACKING_4444_10_in_16_L	64	RGBA/VYUA
VL_PACKING_4444_10_in_16_R	64	RGBA/VYUA
VL_PACKING_4444_12_in_16_L	64	RGBA (signed)
VL_PACKING_4444_12_in_16_R	64	RGBA (signed)
VL_PACKING_4444_13_in_16_L	64	RGBA (signed)
VL_PACKING_4444_13_in_16_R	64	RGBA (signed)

The packings are explained in these categories:

8-Bit Pixel Packings

Figure C-2 shows the VL_PACKING_4_8, an 8-bit packing useful for VYUA monochrome/luma only.

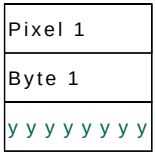

Figure C-2 VL_PACKING_4_8

This packing is

- VL_PACKING_4_8 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style
- VL_PACKING_Y_8_P in the VL, old style
- GL_LUMINANCE GL_UNSIGNED_BYTE in OpenGL
- DM_IMAGE_PACKING_LUMINANCE in DM

Figure C-3 shows VL_PACKING_R444_332, an 8-bit packing in the RGBA color space.

Figure C-3 VL_PACKING_R444_332

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

Figure C-4 shows VL_PACKING_444_332, an 8-bit RGBA packing.

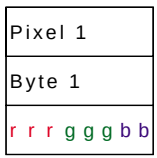

Figure C-4 VL_PACKING_444_332

This packing is

- VL_PACKING_444_332 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
- GL_RGB GL_UNSIGNED_BYTE_3_3_2_EXT in OpenGL
- DM_IMAGE_PACKING_RGB332 in DM

16-Bit Pixel Packings

Figure C-5 shows VL_PACKING_242_8, a 16-bit VYUA packing.

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

Figure C-5 VL_PACKING_242_8

Note: Cr and Cb are more accurate terms than V and U; however, this appendix uses the letters V and U in the illustrations of packings for typographical convenience.

This rarely used packing is VL_PACKING_242_8 + VL_COLORSPACE_{CCIR,YUV} in the VL. It samples chroma and luma in a 4:2:2 pattern. See "Sampling Patterns," later in this appendix.

Figure C-6 shows VL_PACKING_R242_8, a 16-bit 4:2:2 VYUA packing. The most commonly used 4:2:2 packing, it is used by other SGI video hardware as well as DIVO or DIVO-DVC hardware.

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-6 VL_PACKING_R242_8

This packing is

- VL_PACKING_R242_8 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style
- VL_PACKING_YVYU_422_8 in the VL, old style
- GL_YCRCB_422_SGIX GL_UNSIGNED_BYTE in OpenGL
- DM_IMAGE_PACKING_CbYCrY in DM

Figure C-7 shows VL_PACKING_X4444_5551, a 16-bit RGBA packing that corresponds to the QuickTime file 16-bit uncompressed format with alpha.

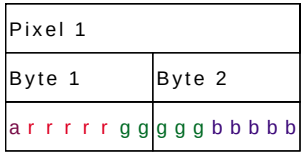

Figure C-7 VL_PACKING_X4444_5551

This packing is

- VL_PACKING_X4444_5551 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
- VL_PACKING_ARGB_1555 in the VL, old style

DM_IMAGE_PACKING_XRGB1555 in DM (even though the upper bit is really alpha)

Figure C-8 shows VL_PACKING_444_5_6_5, a 16-bit RGBA packing.

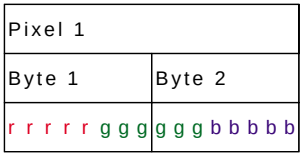

Figure C-8 VL_PACKING_444_5_6_5

This packing is VL_PACKING_444_5_6_5 + VL_COLORSPACE_{RGB,RP175} in the VL, new style.

20-Bit Pixel Packings

Figure C-9 shows VL_PACKING_242_10, a 20-bit VYUA packing.

Pixels 1-2				
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
v v v v v v v v	v v y y y y y y	y y y y u u u u	u u u u u u y y	y y y y y y y y
left				right

Figure C-9 VL_PACKING_242_10

This packing is VL_PACKING_242_10 + VL_COLORSPACE {CCIR,YUV}.

Figure C-10 shows VL_PACKING_R242_10, a 20-bit VYUA packing.

Pixels 1-2				
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
u u u u u u u u	u u y y y y y y	y y y y v v v v	v v v v v v y y	y y y y y y y y
left				right

Figure C-10 VL_PACKING_R242_10

This packing is VL_PACKING_R242_10 + VL_COLORSPACE {CCIR,YUV}.

24-Bit Pixel Packings

Figure C-11 shows VL_PACKING_444_8, a 24-bit RGBA/VYUA packing.

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-11 VL_PACKING_444_8

This packing is

- RGBA:
  - GL_RGB GL_UNSIGNED_BYTE in OpenGL
  - VL_PACKING_444_8 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
  - VL_PACKING_BGR_8_P in the VL, old style
  - GL_RGB GL_UNSIGNED_BYTE in OpenGL
  - DM_IMAGE_PACKING_RGB in DM
- VYUA:
  - VL_PACKING_444_8 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
  - VL_PACKING_UYV_8_P in the VL, old style

Figure C-12 shows VL_PACKING_R444_8, a 24-bit RGBA/VYUA packing.

Pixel 1		
Byte 1	Byte 2	Byte 3
b b b b b b b b	g g g g g g g g	r r r r r r r r
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-12 VL_PACKING_R444_8

This packing is

- RGBA:
  - VL_PACKING_R444_8 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
  - VL_PACKING_RGB_8_P in the VL, old style
  - DM_IMAGE_PACKING_BGR in DM
- VYUA:
  - VL_PACKING_R444_8 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style
  - DM_IMAGE_PACKING_CbYCr in DM

Figure C-13 shows VL_PACKING_4444_6, a 24-bit DIVO/DIVO-DVC-only packing with 6 bits per pixel.

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

Figure C-13 VL_PACKING_4444_6

This packing is

- RGBA: VL_PACKING_4444_6 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
- VYUA: VL_PACKING_4444_6 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style

32-Bit Pixel Packings

This section explains

OpenGL-Like 32-Bit Pixel Packing

Figure C-14 shows VL_PACKING_4444_8, an OpenGL-like 32-bit packing. This packing, supported by many SGI video products, is the most commonly used OpenGL packing.

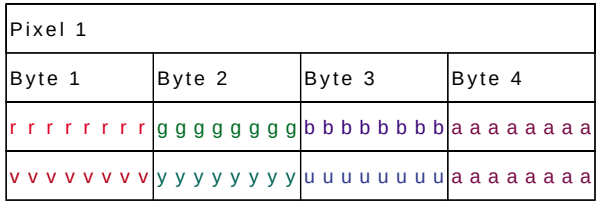

Figure C-14 VL_PACKING_4444_8

This packing is

- RGBA:
  - VL_PACKING_4444_8 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
  - VL_PACKING_ABGR_8 in the VL, old style
  - GL_RGBA GL_UNSIGNED_BYTE in OpenGL (the default)
  - DM_IMAGE_PACKING_RGBA in DM
- VYUA:
  - VL_PACKING_4444_8 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style
  - VL_PACKING_AUYV_4444_8 or VL_PACKING_AUYV_8 in the VL, old style

IRIS GL-Like 32-Bit Pixel Packings

Figure C-15 shows VL_PACKING_R4444_8, an IRIS GL-like 32-bit packing. This packing, supported by many SGI video products, 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-15 VL_PACKING_R4444_8

This packing is

- RGBA:
  - VL_PACKING_R4444_8 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
  - VL_PACKING_RGBA_8 in the VL, old style
  - GL_ABGR_EXT GL_UNSIGNED_BYTE in OpenGL
  - DM_IMAGE_PACKING_ABGR in DM
- VYUA:
  - VL_PACKING_R4444_8 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style
  - VL_PACKING_YUVA_4444_8 in the VL, old style

Figure C-16 shows VL_PACKING_R0444_8, an IRIS GL-like 32-bit packing. This packing is supported by many SGI 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-16 VL_PACKING_R0444_8

- RGBA:
  - VL_PACKING_R0444_8 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
  - VL_PACKING_RGB_8 in the VL, old 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.
- VYUA:
  - VL_PACKING_R0444_8 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style
  - VL_PACKING_YUV_444_8 in the VL, old style

32-Bit Pixel Packing for QuickTime

Figure C-17 shows VL_PACKING_0444_8, a 32-bit packing used for QuickTime files (uncompressed format without alpha).

Pixel 1			
Byte 1	Byte 2	Byte 3	Byte 4
x x x x x x x x	r r r r r r r r	g g g g g g g g	b b b b b b b b
x x x x x x x x	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-17 VL_PACKING_0444_8

This packing is

- RGBA:
  - VL_PACKING_0444_8 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
  - DM_IMAGE_PACKING_XRGB in DM
- VYUA: VL_PACKING_0444_8 + VL_COLORSPACE_{CCIR,YUV}

4:4:4:4 10_10_10_2 32-Bit Pixel Packing

Figure C-18 shows VL_PACKING_4444_10_10_10_2, the 32-bit 4:4:4:4 10_10_10_2 packing.

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

Figure C-18 VL_PACKING_4444_10_10_10_2

This packing is

- RGBA:
  - VL_PACKING_4444_10_10_10_2 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
  - VL_PACKING_A_2_BGR_10 in the VL, old style
  - GL_RGBA GL_UNSIGNED_INT_10_10_10_2_EXT in OpenGL
- VYUA:
  - VL_PACKING_4444_10_10_10_2 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style
  - VL_PACKING_A_2_UYV_10 in the VL, old style

4:2:2:4 10_10_10_2 32-Bit Pixel Packings

Figure C-19 shows VL_PACKING_2424_10_10_10_2Z, the 4:2:2:4 10_10_10_2 32-bit VYUA packing. Only DIVO and DIVO-DVC use this packing.

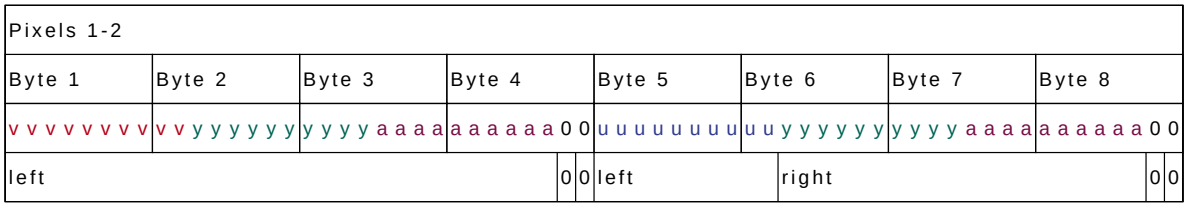

Figure C-19 VL_PACKING_2424_10_10_10_2Z

This packing is

- 4:2:2:4 sampling; see “Sampling Patterns” on page 105 in this appendix
- VL_PACKING_2424_10_10_10_2Z + VL_COLORSPACE_{CCIR,YUV} in the VL, new style

Figure C-20 shows VL_PACKING_R2424_10_10_10_2Z, an alternate 4:2:2:4 10_10_10_2 32-bit packing.

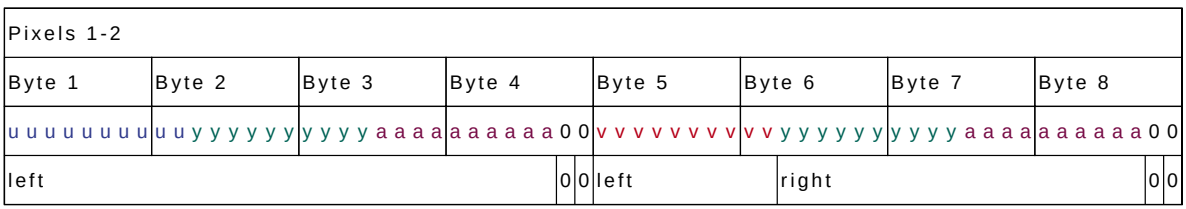

Figure C-20 VL_PACKING_R2424_10_10_10_2Z

This packing is

- 4:2:2:4 sampling; see “Sampling Patterns” on page 105 in this appendix
- VL_PACKING_R2424_10_10_10_2Z + VL_COLORSPACE_{CCIR,YUV} in the VL, new style
- VL_PACKING_AYU_AYV_10 in the VL, old style

4:2:2 10_in_16 32-Bit Pixel Packings

The diagrams of packings that use 4:2:2 sampling show the location (left and right) of each component sample. Only DIVO and DIVO-DVC use this packing.

Figure C-21 shows VL_PACKING_242_10_in_16_L, a DIVO/DIVO-DVC-only 4:2:2 10_in_16 32-bit VYUA packing. For an explanation of the p bit, see “Packing Diagram Conventions” on page 80.

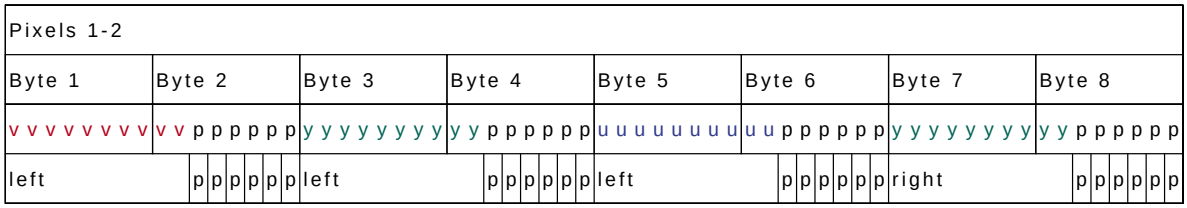

Figure C-21 VL_PACKING_242_10_in_16_L

This packing is

- 4:2:2 sampling; see “Sampling Patterns” on page 105 in this appendix
- VL_PACKING_242_10_in_16_L + VL_COLORSPACE_{CCIR,YUV} in the VL, new style

Figure C-22 shows VL_PACKING_242_10_in_16_R, a DIVO/DIVO-DVC-only 4:2:2 10_in_16 32-bit VYUA packing.

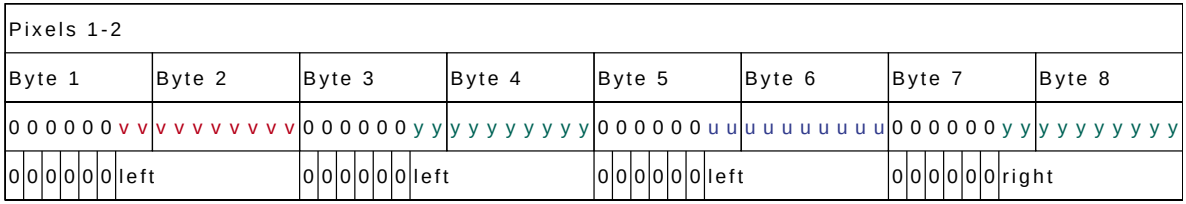

Figure C-22 VL_PACKING_242_10_in_16_R

This packing is

- 4:2:2 sampling; see “Sampling Patterns” on page 105 in this appendix
- VL_PACKING_242_10_in_16_R + VL_COLORSPACE_{CCIR,YUV} in the VL, new style

Figure C-23 shows VL_PACKING_R242_10_in_16_L, a 4:2:2 10_in_16 32-bit VYUA packing. This packing is supported by several SGI video products. For an explanation of the p bit, see “Packing Diagram Conventions” on page 80.

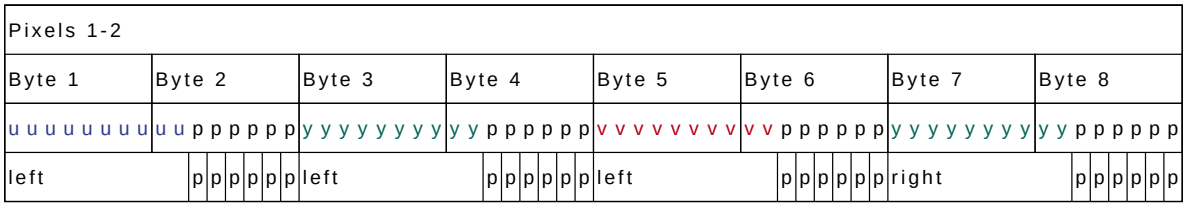

Figure C-23 VL_PACKING_R242_10_in_16_L

This packing is

- 4:2:2 sampling; see “Sampling Patterns” on page 105 in this appendix
- VL_PACKING_R242_10_in_16_L + VL_COLORSPACE_{CCIR,YUV} in the VL, new style
- VL_PACKING_YVYU_422_10 in the VL, old style

Figure C-24 shows VL_PACKING_R242_10_in_16_R, a DIVO/DIVO-DVC-only 4:2:2 10_in_16 32-bit VYUA packing.

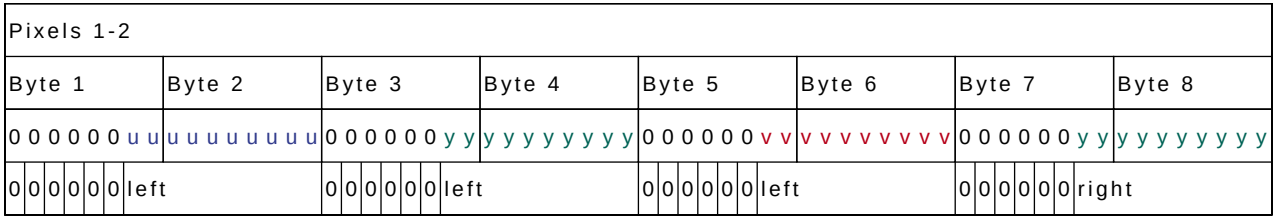

Figure C-24 VL_PACKING_R242_10_in_16_R

This packing is

- VYUA VL_PACKING_R242_10_in_16_R + VL_COLORSPACE_{CCIR,YUV}
- 4:2:2 sampling; see “Sampling Patterns” on page 105 in this appendix

SDTI Packing

Figure C-25 shows the SDTI packing VL_PACKING_SDTI_DV.

Pixel 1		Pixel 2	
Byte 1	Byte 2	Byte 3	Byte 4
Cb7 Cb6 Cb5 Cb4 Cb3 Cb2 Cb1 Cb0	Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0	Cr7 Cr6 Cr5 Cr4 Cr3 Cr2 Cr1 Cr0	Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0

Figure C-25 VL_PACKING_SDTI_DV

This packing

- is VL_PACKING_ SDTI_DV + VL_COLOR_SPACE_CCIR601
- uses 4:2:2 sampling; see “Sampling Patterns” on page 105 in this appendix

36-Bit Pixel Packing

Figure C-26 shows VL_PACKING_444_12, the 36-bit packing, which has 12 bits per component. Only DIVO and DIVO-DVC use this packing.

Pixel 1				Pixel 2				
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9
r r r r r r r r	r r r r g g g g	g g g g g g g g	b b b b b b b b	b b b b r r r r	r r r r r r r r	g g g g g g g g	g g g g b b b b	b b b b b b b b
v v v v v v v v	v v v v y y y y	y y y y y y y y	u u u u u u u u	u u u u v v v v	v v v v v v v v	y y y y y y y y	y y y y u u u u	u u u u u u u u

Figure C-26 VL_PACKING_444_12

This packing is

- RGBA: VL_PACKING_444_12 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
- VYUA: VL_PACKING_444_12 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style

The components in this packing are signed:

- For RGB and RP-175 color spaces, the components are often negative.
- For the YUV color space, they are negative when the incoming video signal has values outside of the nominal range.
- For the CCIR601 color space, they are always positive.

48-Bit Pixel Packings

Figure C-27 shows VL_PACKING_4444_12, a 48-bit packing, with 12 bits per component. Only DIVO and DIVO-DVC use this packing.

Pixel 1					
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
r r r r r r r r	r r r r g g g g	g g g g g g g g	b b b b b b b b	b b b b a a a a	a a a a a a a a
v v v v v v v v	v v v v y y y y	y y y y y y y y	u u u u u u u u	u u u u a a a a	a a a a a a a a

Figure C-27 VL_PACKING_4444_12

This packing is

- RGBA: VL_PACKING_4444_12 + VL_COLORSPACE_{RGB,RP175} in the VL, new style
- VYUA: VL_PACKING_4444_12 + VL_COLORSPACE_{CCIR,YUV} in the VL, new style

The components in this packing are signed:

- For RGB and RP-175 color spaces, the components are often negative.
- For the YUV color space, they are negative when the incoming video signal has values outside of the nominal range.
- For the CCIR601 color space, they are always positive.

Figure C-28 shows VL_PACKING_444_10_in_16_L, a 48-bit packing, with 10 bits per component and no alpha. For an explanation of the p bit, see “Packing Diagram Conventions” on page 80.

Pixel 1					
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
r r r r r r r r	r r p p p p p p	g g g g g g g g	g g p p p p p p	b b b b b b b b	b b p p p p p p
v v v v v v v v	v v p p p p p p	y y y y y y y y	y y p p p p p p	u u u u u u u u	u u p p p p p p

Figure C-28 VL_PACKING_444_10_in_16_L

This packing is

- RGBA: VL_PACKING_444_10_in_16_L + VL_COLORSPACE_{RGB,RP175} in the VL, new style
- VYUA: VL_PACKING_444_10_in_16_L, + VL_COLORSPACE_{CCIR,YUV} in the VL, new style

64-Bit Pixel Packings

Figure C-29 shows VL_PACKING_4444_10_in_16_L. For an explanation of the p bit, see “Packing Diagram Conventions” on page 80.

Pixel 1							
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
r r r r r r r r	r r p p p p p p	g g g g g g g g	g g p p p p p p	b b b b b b b b	b b p p p p p p	a a a a a a a a	a a p p p p p p
v v v v v v v v	v v p p p p p p	y y y y y y y y	y y p p p p p p	u u u u u u u u	u u p p p p p p	a a a a a a a a	a a p p p p p p

Figure C-29 VL_PACKING_4444_10_in_16_L

This packing is

- RGBA:
  - VL_PACKING_4444_10_in_16_L + VL_COLORSPACE_{RGB,RP175} in the VL, new style
  - VL_PACKING_ABGR_10 in the VL, old style
- VYUA:
  - VL_PACKING_4444_10_in_16_L + VL_COLORSPACE_{CCIR,YUV} in the VL, new style
  - VL_PACKING_AUYV_4444_10 in the VL, old style

Figure C-30 shows VL_PACKING_4444_10_in_16_R.

Pixel 1							
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
0 0 0 0 0 0 r r	r r r r r r r r	0 0 0 0 0 0 g g	g g g g g g g g	0 0 0 0 0 0 b b	b b b b b b b b	0 0 0 0 0 0 a a	a a a a a a a a
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 0 u u	u u u u u u u u	0 0 0 0 0 0 a a	a a a a a a a a

Figure C-30 VL_PACKING_4444_10_in_16_R

This packing is

- RGBA: VL_PACKING_4444_10_in_16_R + VL_COLORSPACE_{RGB,RP175}
- VYUA: VL_PACKING_4444_10_in_16_R + VL_COLORSPACE_{CCIR,YUV}

Figure C-31 shows VL_PACKING_4444_12_in_16_L, a 64-bit RGBA packing. For an explanation of the p bit, see “Packing Diagram Conventions” on page 80.

Pixel 1							
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
r r r r r r r r	r r r r p p p p	g g g g g g g g	g g g g p p p p	b b b b b b b b	b b b b p p p p	a a a a a a a a	a a a a p p p p

Figure C-31 VL_PACKING_4444_12_in_16_L

This packing is VL_PACKING_4444_12_in_16_L + VL_COLORSPACE_{RGB,RP175} in the VL, new style.

The components in this packing are signed:

- For RGB and RP-175 color spaces, they are often negative.
- For the YUV color space, they are negative when the incoming video signal has values outside of the nominal range.
- For the CCIR601 color space, they are always positive.

Figure C-32 shows VL_PACKING_4444_12_in_16_R, a 64-bit RGBA packing for use with extended RGB components.

Pixel 1							
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
s s s s r r r r	r r r r r r r r	s s s s g g g g	g g g g g g g g	s s s s b b b b	b b b b b b b b	s s s s a a a a	a a a a a a a a

Figure C-32 VL_PACKING_4444_12_in_16_R

This packing is VL_PACKING_4444_12_in_16_R + VL_COLORSPACE_{RGB,RP175} in the VL, new style.

Note: The components in this packing are signed, with positive and negative values varying by color space, as explained for VL_PACKING_4444_12_in_16_L. The s in the more significant bits in Figure C-32 indicates a sign extension padding bit.

Figure C-33 shows VL_PACKING_4444_13_in_16_L, a 64-bit RGBA packing for use with extended RGB components. For an explanation of the p bit, see “Packing Diagram Conventions” on page 80.

Pixel 1							
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
r r r r r r r r	r r r r r p p p	g g g g g g g g	g g g g g p p p	b b b b b b b b	b b b b b p p p	a a a a a a a a	a a a a a p p p

Figure C-33 VL_PACKING_4444_13_in_16_L

This packing is VL_PACKING_4444_13_in_16_L + VL_COLORSPACE_{RGB,RP175} in the VL, new style.

The components in this packing are signed, with positive and negative values varying by color space, as explained for VL_PACKING_4444_12_in_16_L.

Figure C-34 shows VL_PACKING_4444_13_in_16_R, a 64-bit packing for use with extended RGB components.

Pixel 1							
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
s s s r r r r r	r r r r r r r r	s s s g g g g g	g g g g g g g g	s s s b b b b b	b b b b b b b b	s s s a a a a a	a a a a a a a a

Figure C-34 VL_PACKING_4444_13_in_16_R

This packing is VL_PACKING_4444_13_in_16_R + VL_COLORSPACE_{RGB,RP175} in the VL, new style.

Note: The components in this packing are signed, with positive and negative values varying by color space, as explained for VL_PACKING_4444_12_in_16_L. The s in the more significant bits in Figure C-34 indicates a sign extension padding bit.

Sampling Patterns

Sampling patterns are described in these sections:

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

Some of the diagrams in the "DIVO/DIVO-DVC Pixel Packings" section 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-35 diagrams this sampling pattern.

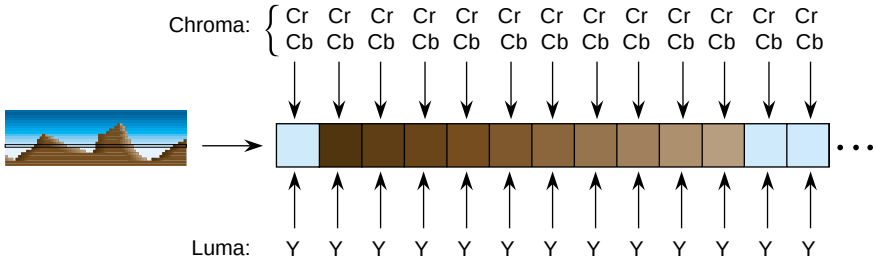

Figure C-35 4:4:4 Sampling

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 Ys) but only one chroma sample (one sample of Cr and Cb, which together determine the chroma), as shown in Figure C-36.

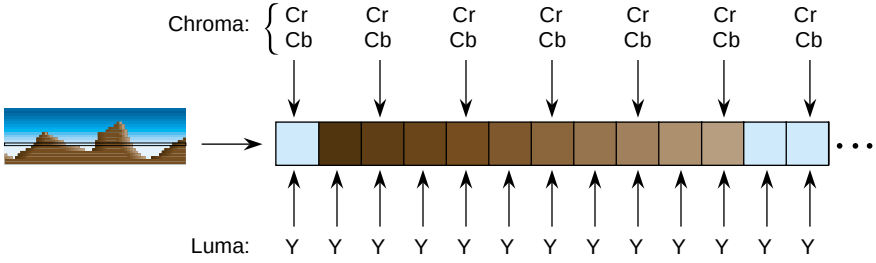

Figure C-36 4:2:2 Sampling

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 "DIVO/DIVO-DVC Pixel Packings" section of this appendix show the 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.

4:1:1 Sampling (DIVO-DVC Only)

The packings shown in diagrams that indicate 4:1:1 sampling make sense only in the VYUA color spaces. For every four pixels, there are four luma samples (four Ys) but only one chroma sample (one sample of Cr and Cb, which together determine the chroma), as shown in Figure C-37.

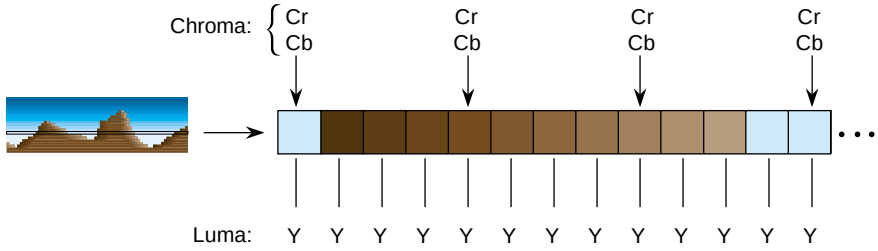

Figure C-37 4:1:1 Sampling

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 DIVO-DVC board uses 4:1:1 sampling for 525/60 (NTSC) DV, NTSC DVCPRO, and 625/50 (PAL) DVCPRO compression.

4:2:0 Sampling (DIVO-DVC Only)

The packings shown in diagrams that indicate 4:2:0 sampling make sense only in the VYUA color spaces. For every two pixels, there are two luma samples (two Ys) but only one chroma sample (on alternate lines, one sample of Cr and one sample of Cb), as shown in Figure C-38.

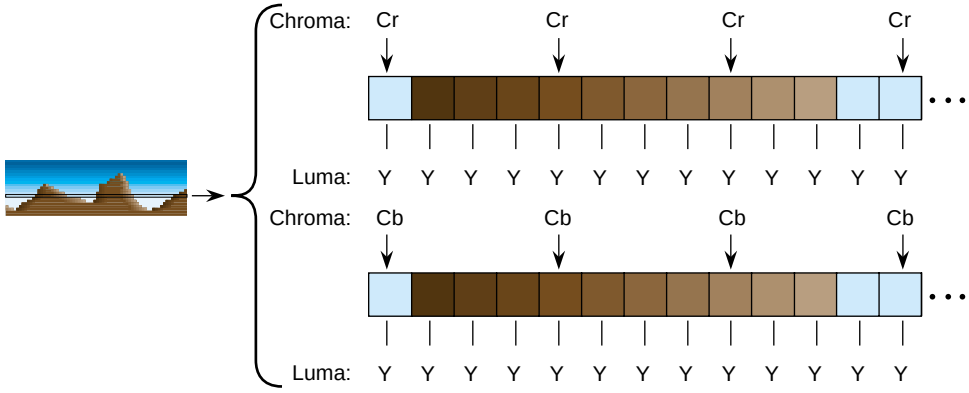

Figure C-38 4:2:0 Sampling

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 DIVO-DVC board uses 4:2:0 sampling for PAL (625/50) DV compression.

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 an unsigned 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
- compressed (headroom) range, which provides numerical headroom, which is often useful when processing video images:
  - Cr and Cb: [(2^n)/16, 15*(2^n)/16]
  - Y, A, R, G, and B: [(2^n)/16, 235*(2^n)/256]

Compressed 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, B: [64, 940]

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 real 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 compressed-range; you must determine this from context

Using the traditional 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 that comes with the DIVO or DIVO-DVC option (for IRIX 6.4 and later) 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-2.

Table C-2 VL_COLORSPACE Options

Color Set	Full-Range Components	Compressed-Range Components
RGBA	VL_COLORSPACE_RGB	VL_COLORSPACE_RP175
VYUA	VL_COLORSPACE_YUV	VL_COLORSPACE_CCIR

The option VL_COLORSPACE_NONE is useful when you want to treat CCIR 601 digital video as a raw 10-bit data stream (as in SDDI).

The DIVO and DIVO-DVC options perform color-space conversion if the color space implied by VL_FORMAT on the video node disagrees with that implied by VL_COLORSPACE on the memory node.

Color-Space Conversions

The DIVO and DIVO-DVC boards support four native color spaces—RGB, YUV, CCIR, and RP-175 compressed RGB. The choice of color space is determined by the format control for video sources and drains and by the color-space controls for memory sources and drains. If the color space selected for memory sources and drains matches that used by the current video format, no color-space conversions are performed. When DIVO/DIVO-DVC performs color-space conversions, extreme care is taken to assure the correctness and precision of the result.

Understanding the capabilities of DIVO/DIVO-DVC 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

The appendix concludes with examples.

DIVO/DIVO-DVC Color Spaces

The DIVO/DIVO-DVC option uses a minimum of ten bits of precision for each color component at all steps of its internal pipeline. Representations for the four native internal color representations are explained separately in this section.

RGB

RGB is the color space used by the graphics subsystem. 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]. Table D-1 summarizes the clamping range for each resulting RGB component for various conversions.

Table D-1 Clamping Ranges for RGB Component Conversions

When converting to...	Each resulting RGB component is clamped to the range
10-bit RGB	[0..1023]
8-bit RGB	[0..255]
12-bit signed RGB	[-2048..2047]
13-bit signed RGB	[-4096..4095]

DIVO/DIVO-DVC uses this color space only at the memory interface.

Note: You should not normally use 4:2:2 coding with RGB data.

YUV

The YUV color space is obtained from RGB by the matrix transformation in the following equation.

$$\begin{bmatrix} 0.500 & -0.419 & -0.081 \\ 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 512 \\ 0 \\ 512 \end{bmatrix} = \begin{bmatrix} V \\ Y \\ U \end{bmatrix}$$

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].

DIVO/DIVO-DVC uses this color space only at the memory interface. With proper filtering, 4:2:2 coding can be used.

CCIR

The CCIR color space is obtained from RGB by the matrix transformation in the follwing equation.

$$\begin{bmatrix} 0.500 & -0.419 & -0.081 \\ 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \end{bmatrix} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix} \times \begin{bmatrix} \frac{896}{1023} \\ \frac{876}{1023} \\ \frac{896}{1023} \end{bmatrix} + \begin{bmatrix} 512 \\ 64 \\ 512 \end{bmatrix} = \begin{bmatrix} Cr \\ Y \\ Cb \end{bmatrix}$$

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. The memory interface can use this color space. With proper filtering, 4:2:2 coding can be used.

RP-175 Compressed RGB

The RP-175 color space is obtained from RGB by the matrix transformation in the following equation.

$$\frac{876}{1023} \times \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 64 \\ 64 \\ 64 \end{bmatrix} = \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix}$$

When converting to 10-bit R‘G‘B‘, the R‘, G‘, and B‘ values are clamped to the range [0..1023]. Black has the R‘G‘B‘ value [64,64,64]. White has the value [940,940,940]. Other clamping ranges are the same as the standard RGB case.

This color space is used by the component digital RGB format. The memory interface can use this color space. You should not use 4:2:2 coding with this color space.

Mathematical Operations Performed During Conversions

DIVO/DIVO-DVC 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: Data from a D1 deck should be stored as a 8-bit 4:2:2 in the CCIR color space. Data from a dual-link telecine could be stored as 4:4:4 10-bit RP-175 RGB. 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 DIVO/DIVO-DVC option to perform real-time color space and 4:2:2 ⇔ 4:4:4 conversions.

Conversions are performed only when absolutely required. Each incoming or outgoing stream can be converted from its current color space to any other color space.

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 DIVO/DIVO-DVC

The DIVO/DIVO-DVC board stores colors with a minimum of 10 bits of precision at all steps in its pipeline. When performing color space conversions, the data is converted to 13-bit signed values before being passed to the matrix multipliers. The matrix multipliers have 13-bit coefficients and 26-bit accumulators. The most significant 14 bits of the matrix-multiplication result are passed on to additional hardware, which applies any needed offsets and then clamps to the proper range.

SGI, has verified both through simulation and hardware testing that the maximal error for two conversions (RGB to CCIR to RGB) is two 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.

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 decouple chrominance and luminance information.

Since RGB represents visible colors, it is contained inside the YUV and CCIR spaces. The CCIR and RP-175 color spaces also have 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.

Any time a conversion operation is performed between CCIR and one of the other color spaces, 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 imply modification of the original signal and generally result in a loss of saturation.

During conversion 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.

During conversion from RGB to YUV or CCIR, clamping never occurs, because all RGB colors are representable in those color spaces.

During conversion from CCIR or YUV to RGB or RP-175 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 don't align with those in the other.

DIVO/DIVO-DVC also supports signed RGB representations with 12 or 13 significant bits. If one of these representations is used, the entire CCIR color space is representable and no clamping will occur. Application software must specifically select this mode and handle the (12/13)-bit data to gain this benefit.

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.

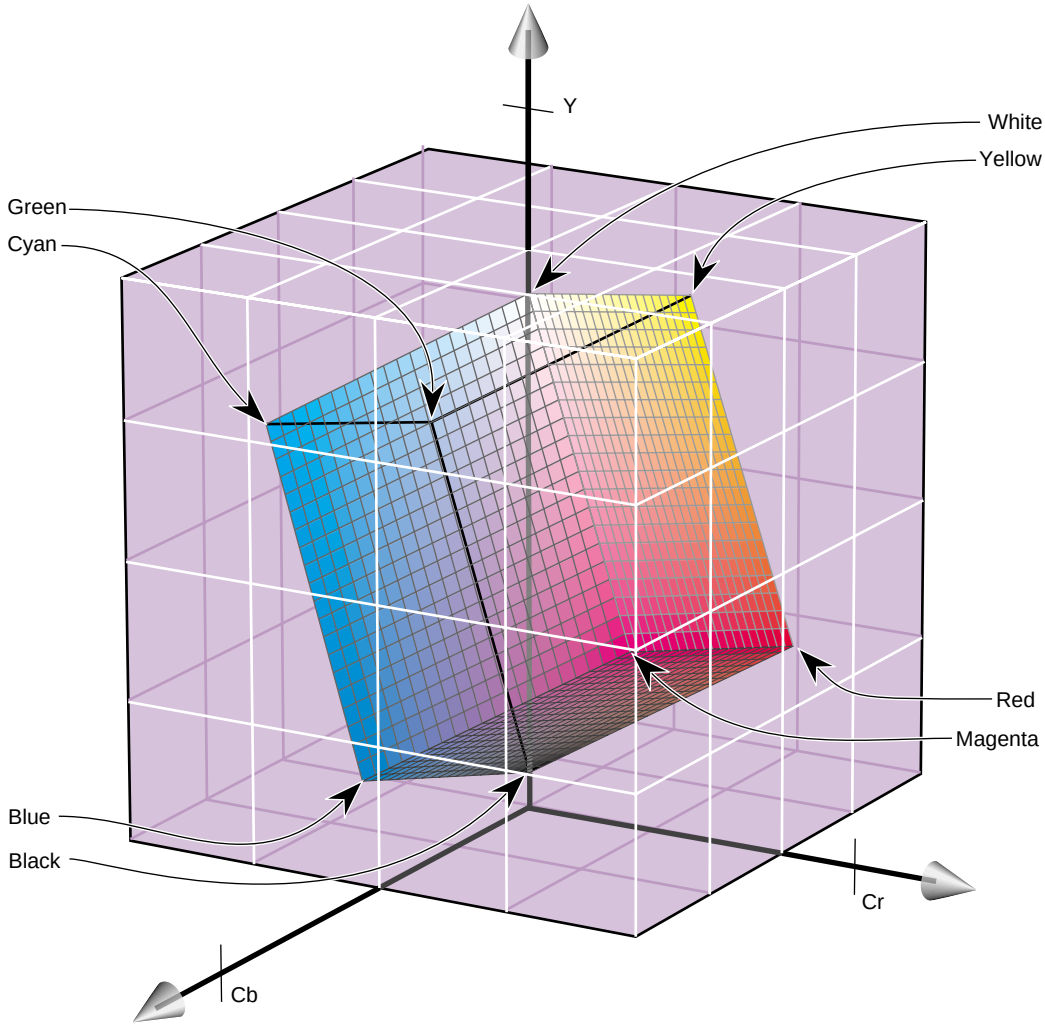

Figure D-1 RGB Cube in CCIR Space

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.

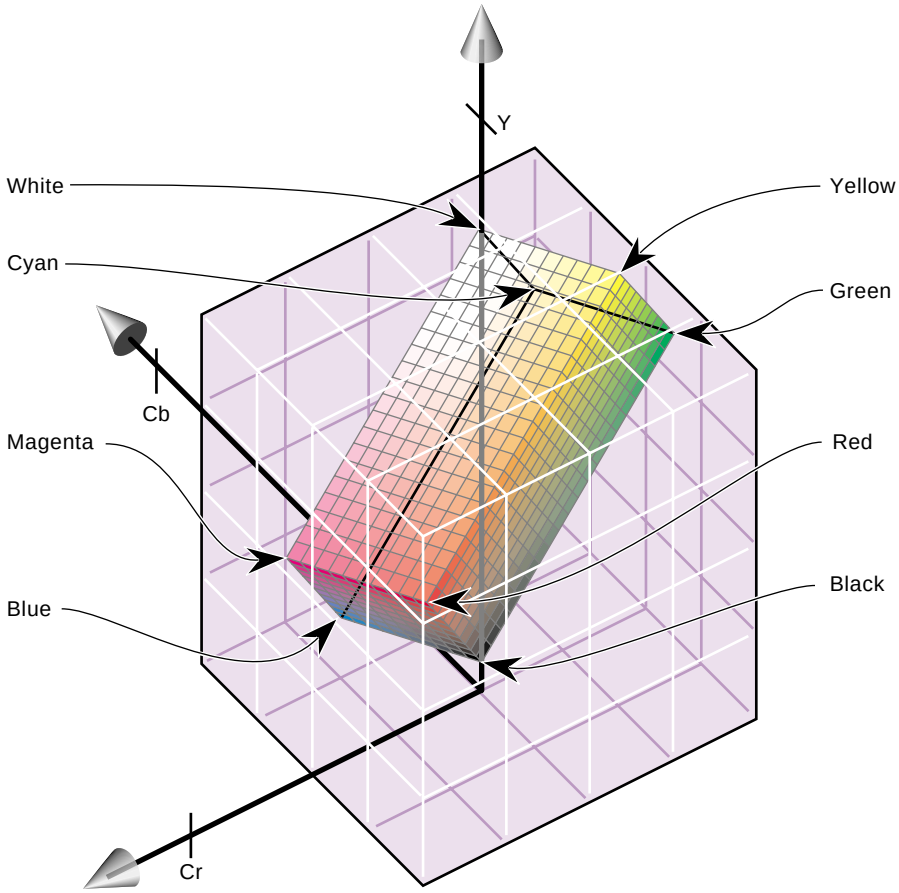

Figure D-2 Color Cube With Luminance/Chrominance Ramp Vector

Example Color Conversions

This section includes example graphs that display the results of converting from CCIR to eight or ten bit RGB and back. They show the same type of result you would see if you passed a digital signal through DIVO/DIVO-DVC using the *soft_ee* program with RGB as the color space and an eight or ten-bit data-packing. If you use CCIR as the memory color space or use a data-packing with 12/13-bit signed representations, the output will look exactly like the input. If the memory color space matches the video color space, the output will be a bit-perfect copy of the input.

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.

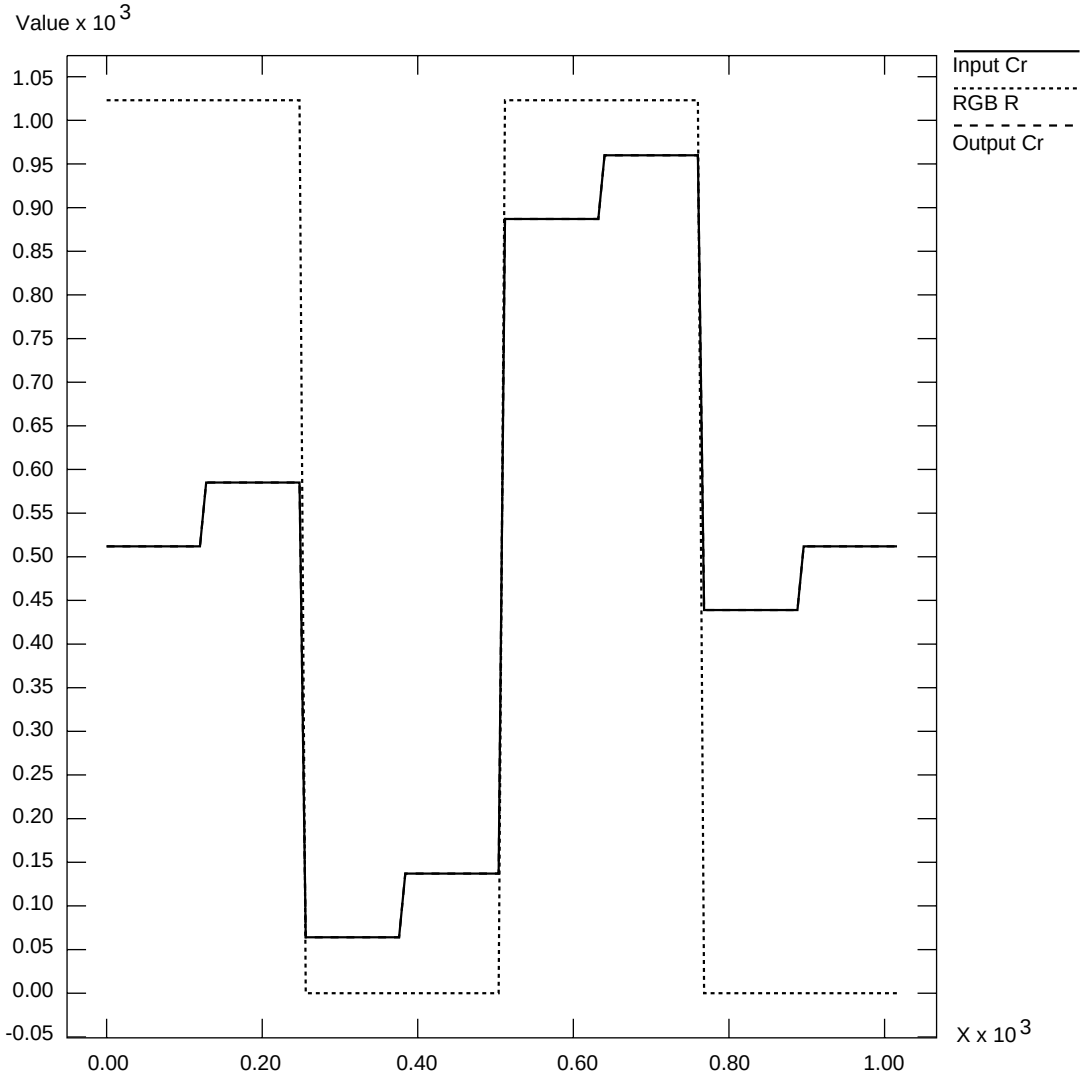

Figure D-3 100% Color Bars: Cr/R

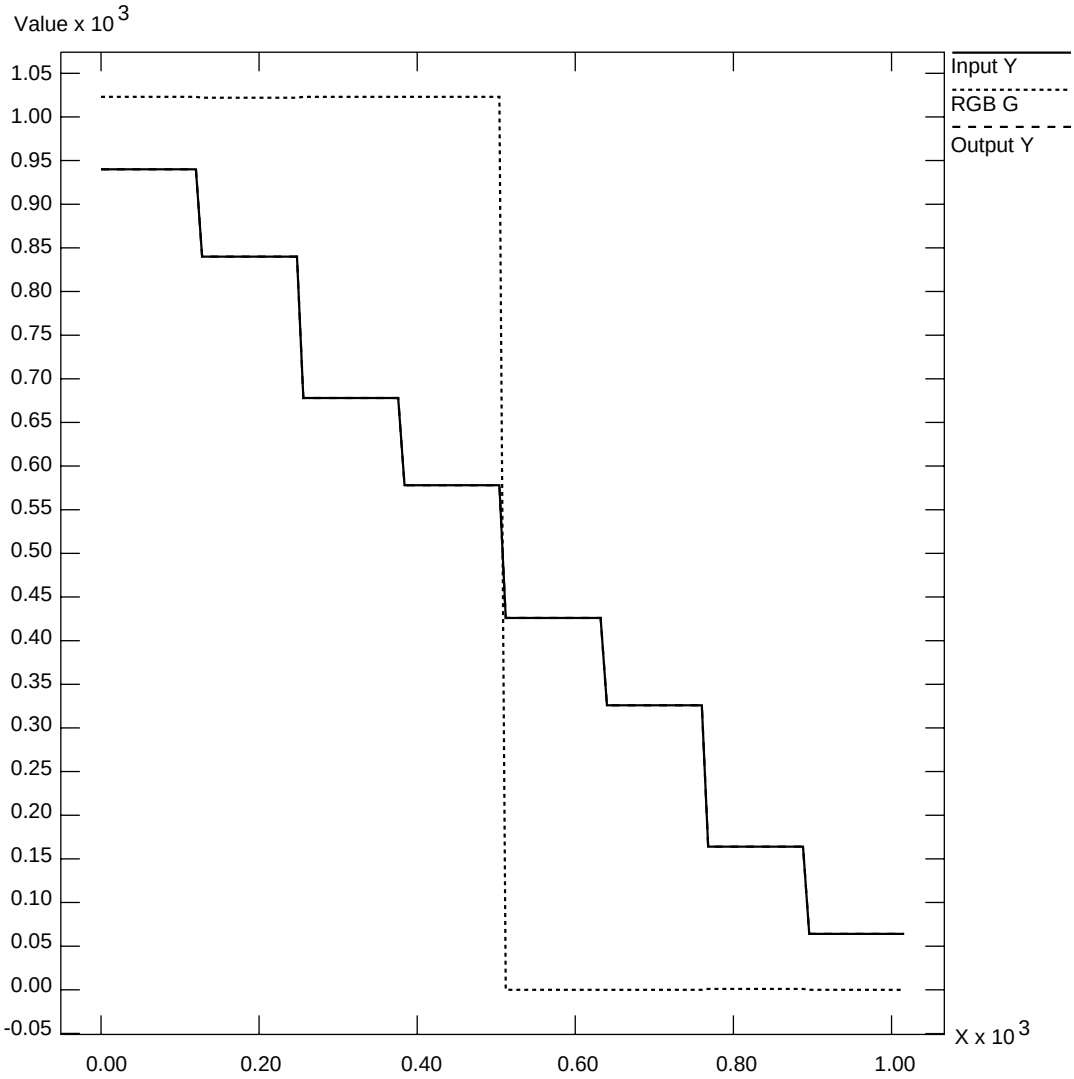

Figure D-4 100% Color Bars: Y/G

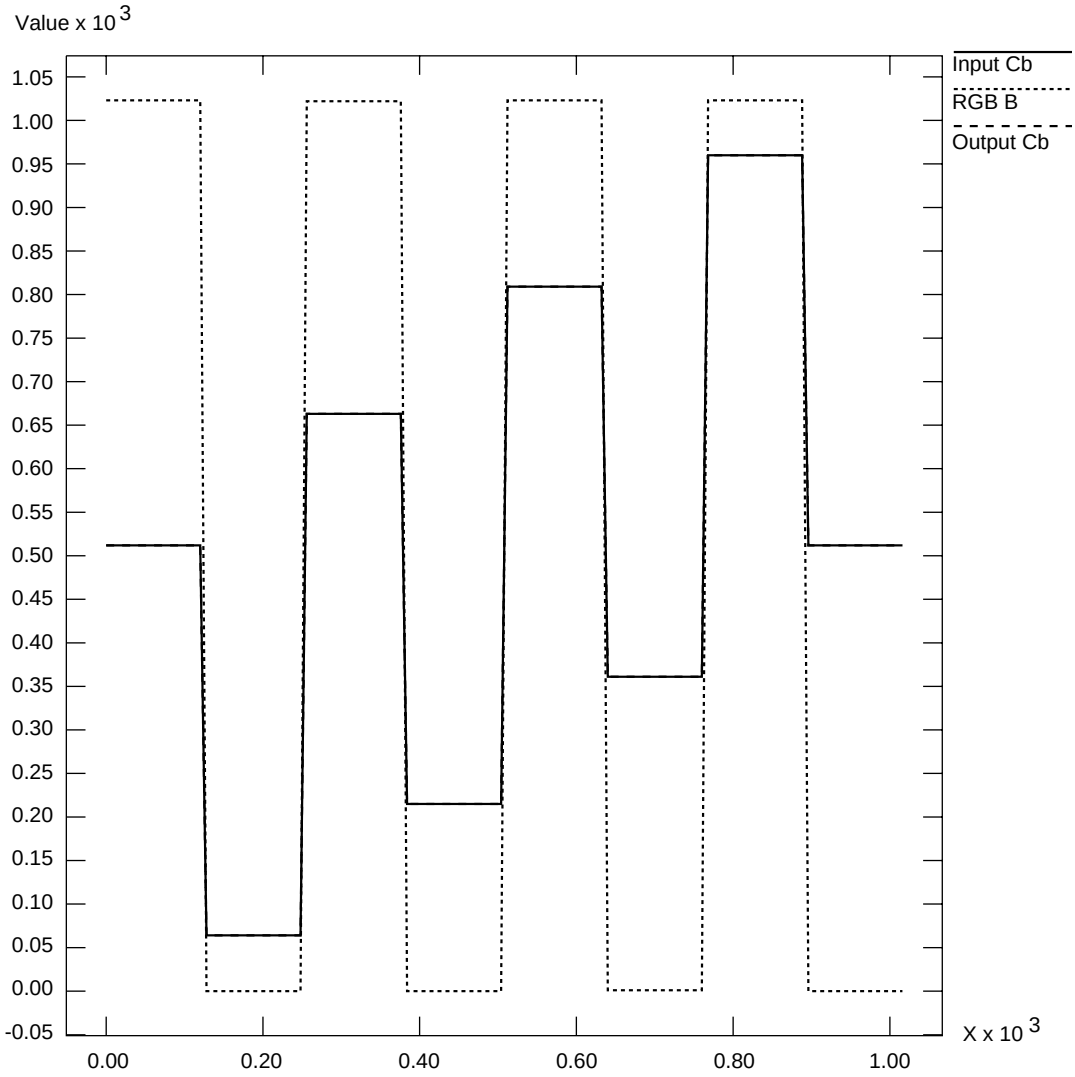

Figure D-5 100% Color Bars: Cb/B

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.

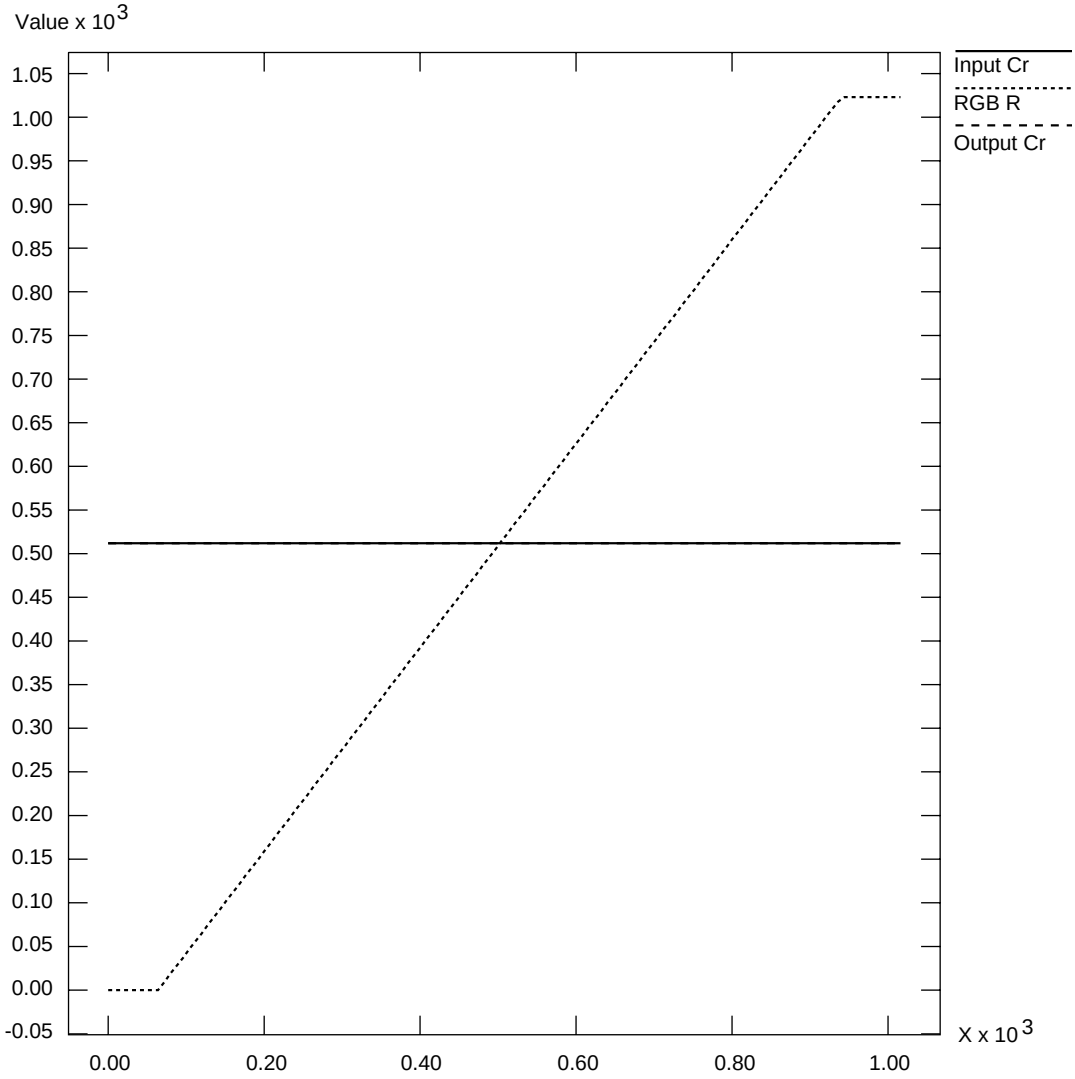

Figure D-6 Luminance Ramp: Cr/R

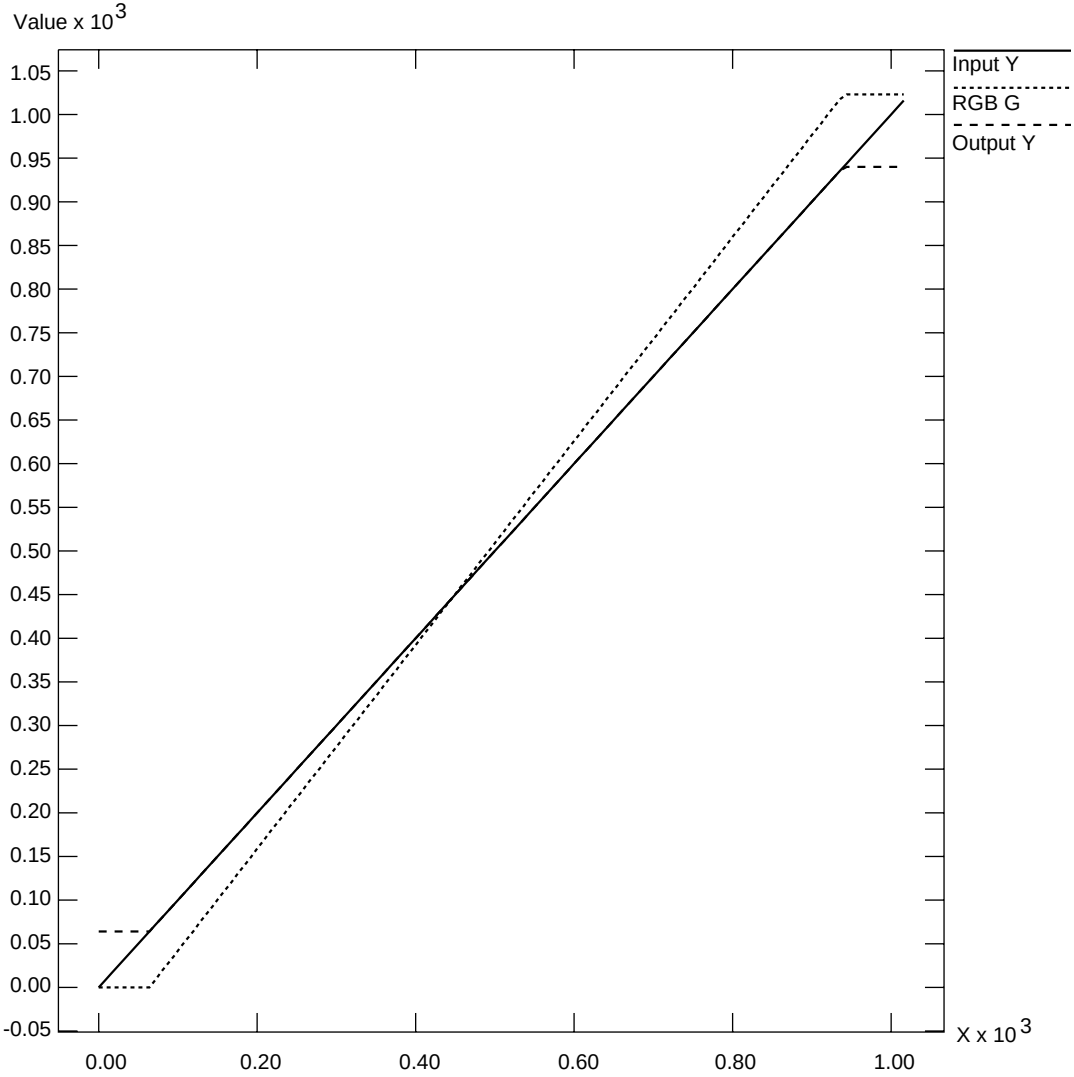

Figure D-7 Luminance Ramp: Y/G

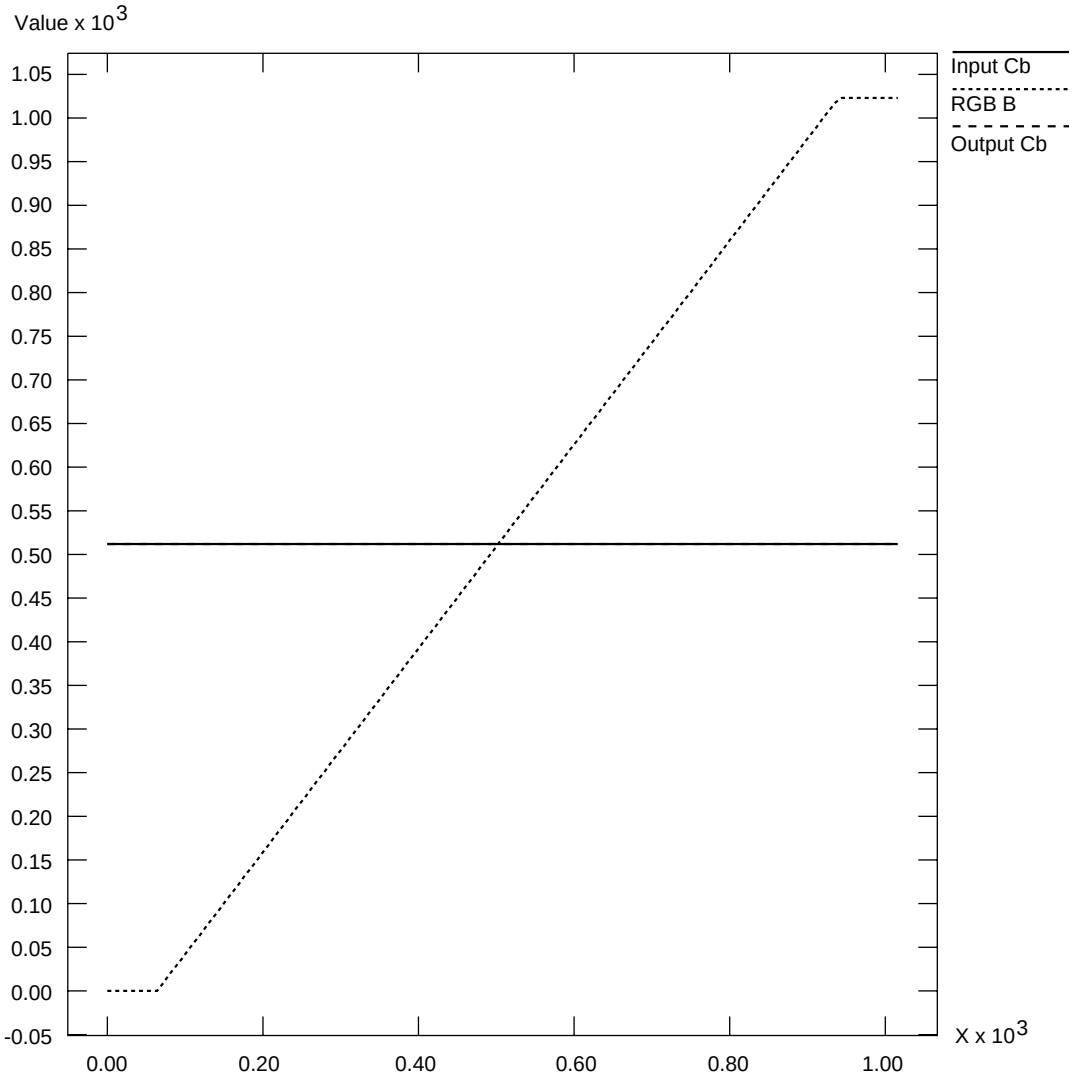

Figure D-8 Luminance Ramp: Cb/B

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.

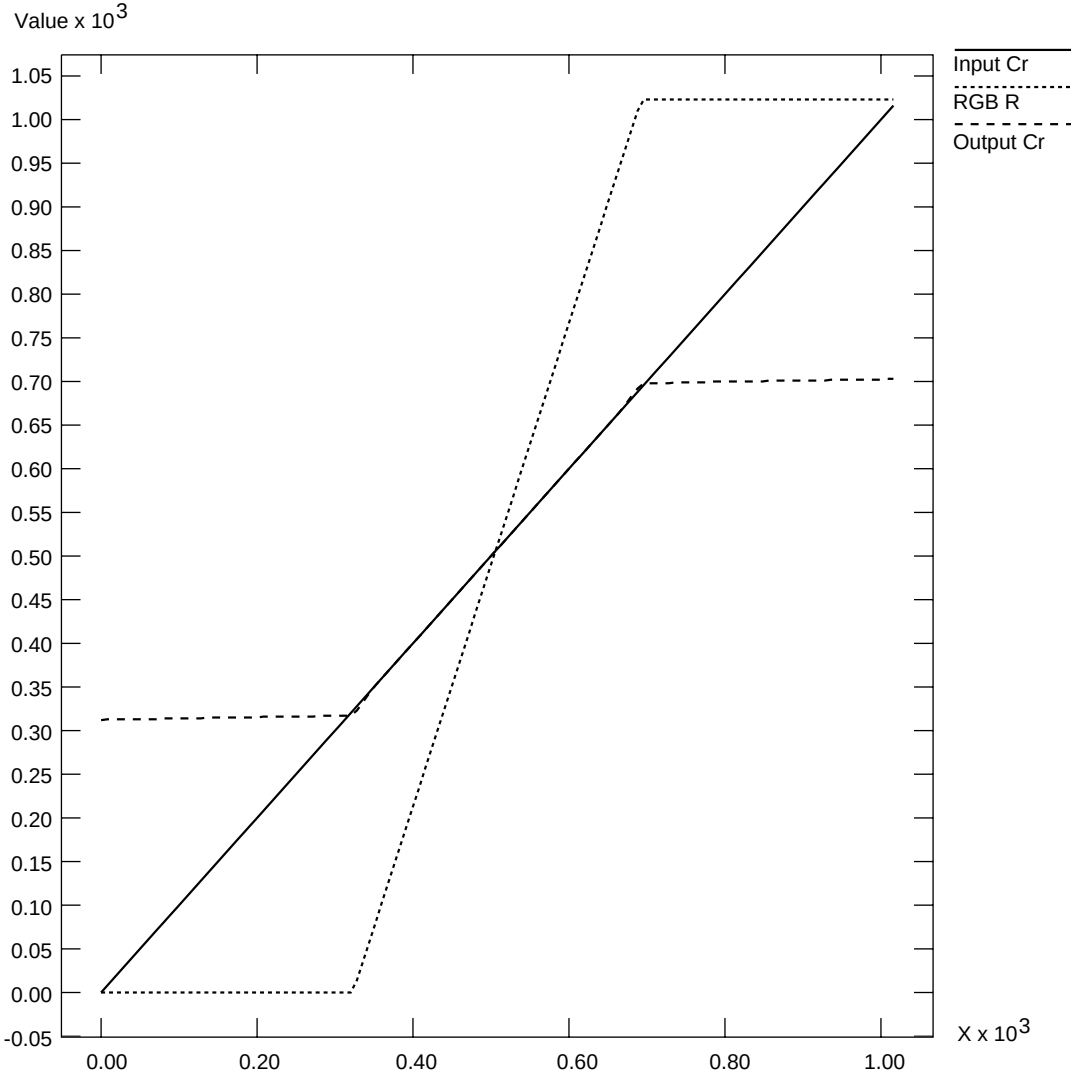

Figure D-9 Chroma/Luma Ramp: Cr/R

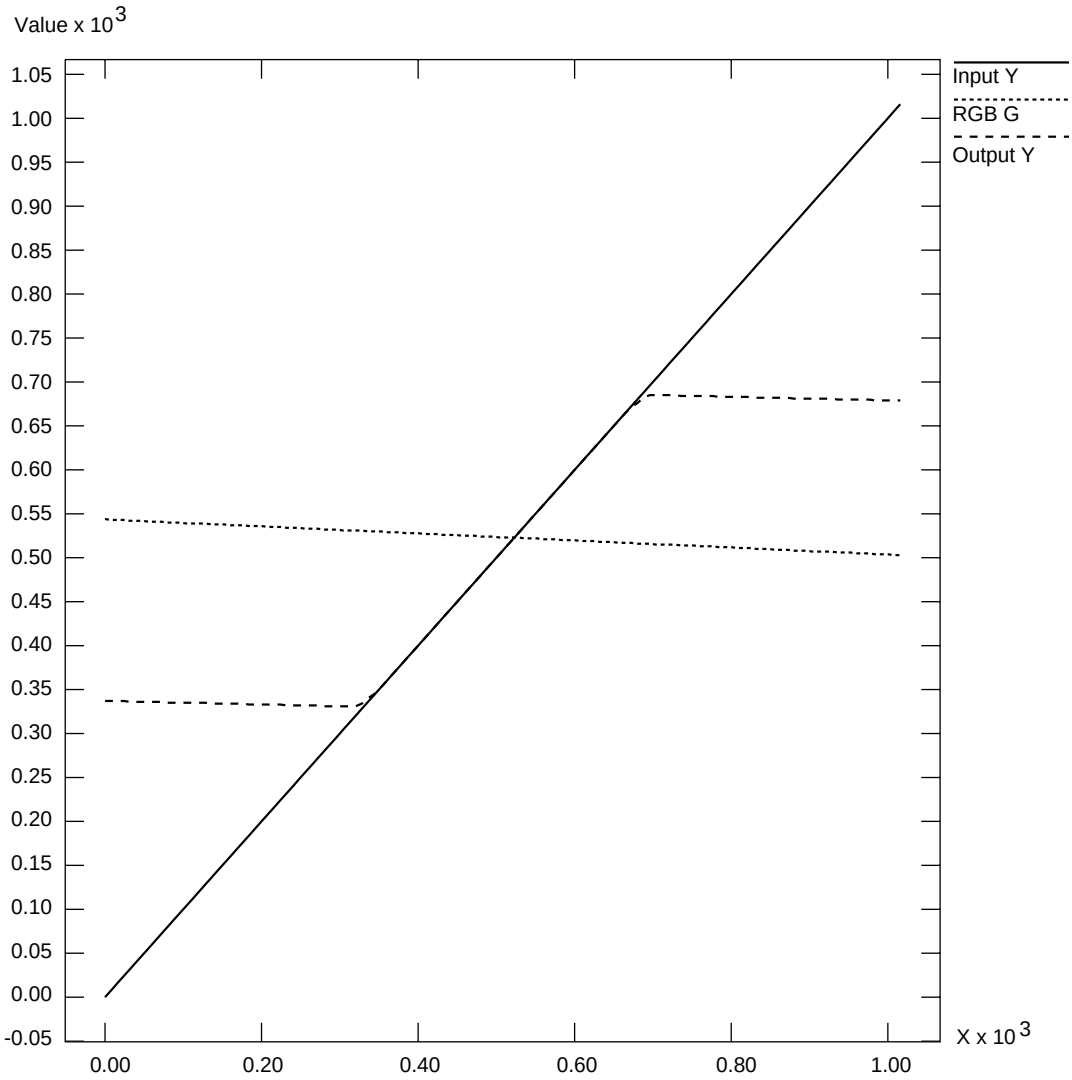

Figure D-10 Chroma/Luma Ramp: Y/G

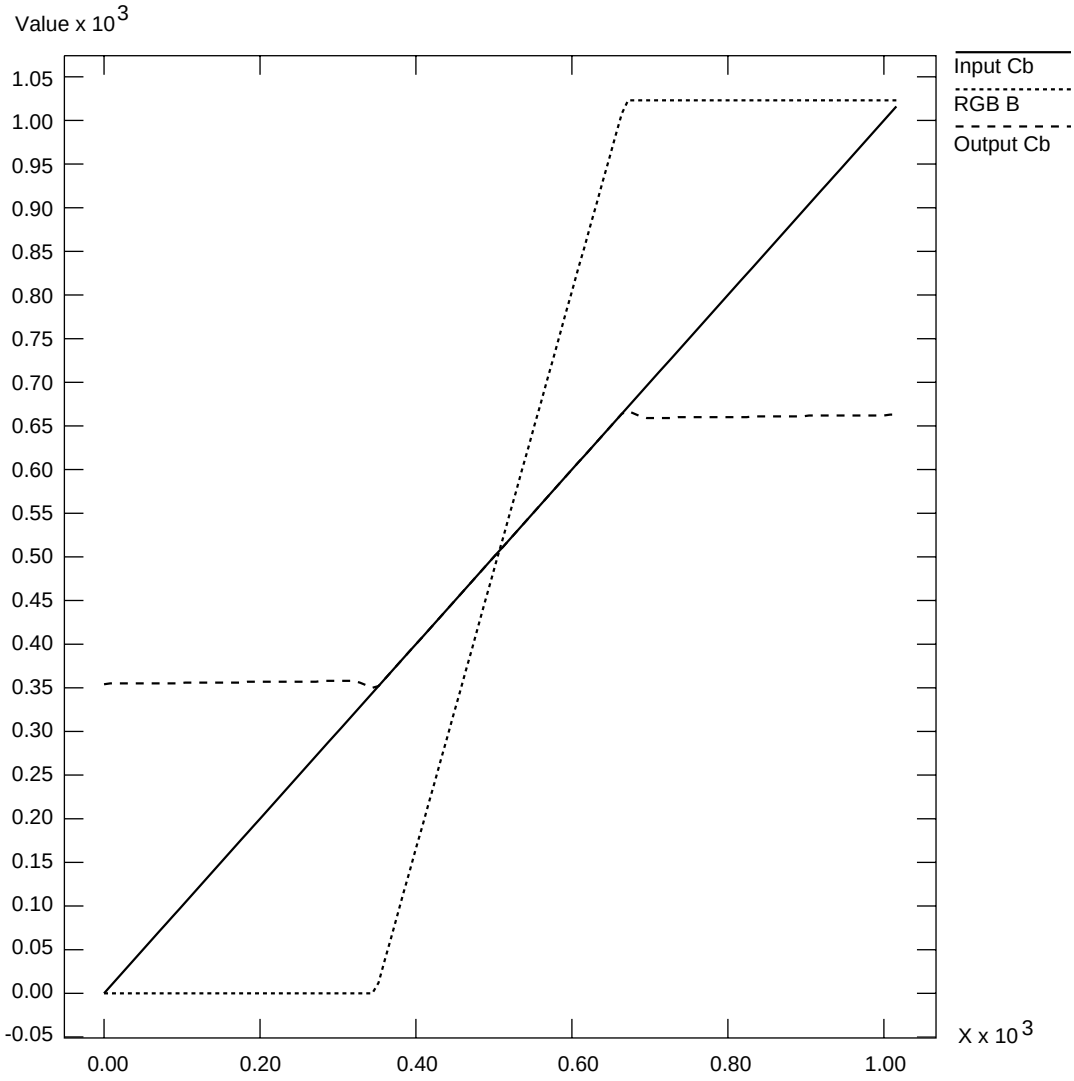

Figure D-11 Chroma/Luma Ramp: Cb/B

Programming Methods for Real-Time Digital Media Recording and Playback

This appendix explains the following real-time disk I/O concepts:

The example source for the utilities discussed in this appendix can be found in `/usr/share/src/dmedia/tools`. The code examples are written to Digital Media buffers (DMbuffers), a real-time data transport facility. See the *Digital Media Programming Guide* (document number 007-1799-060 or later, hereafter referred to as the DMPG) for more details. The emphasis here is not on how data is acquired from or transported to the video device, but rather on how data is moved to disk in real time.

The DMPG covers basic digital media programming concepts; two simple programming examples in `/usr/share/src/dmedia/video/DIVO, divo_vidtomem.c` and divo_`memtovid.c`, illustrate how video data is copied into and out of the DMbuffers for the simpler non-real-time case. At an abstract level, high-bandwidth throughput is simple; the work is in the details, as explained in this appendix.

Direct I/O

The most efficient way to move data on and off a disk device is to use the XFS filesystem with direct I/O mode and large data transfer sizes. If large transfer sizes cannot be achieved, you can combine memory pages from noncontiguous locations using writev(2) or readv(2). Finally, you can use asynchronous I/O to queue multiple I/O requests to the kernel without waiting for blocked calls to return. Other real-time software features and products, such as REACT, can be used to assure low-latency interrupts and high-priority scheduling, but are not absolutely necessary for digital media applications.

Normally, when a disk file is opened with no status flags specified, a call to write(2) for that file returns as soon as the data has been copied to a buffer managed by the device driver (see open(2)). The actual disk write may not take place until considerable time has passed. A common pool of disk buffers is used for all disk files.

Disk buffering is integrated with the virtual memory paging mechanism. A daemon executes periodically and initiates output of buffered blocks according to the age of the data and the needs of the system. You can force the writing of all pending output for a file by calling fsync(2) or by opening the file and specifying the O_SYNC flag. However, the process blocks until the data has been written to disk, and all output data must still be copied from the buffer in the user address space to a buffer in the kernel address space. See Chapter 8, “Optimizing Disk I/O for a Real-Time Program,” in the *REACT Real Time Programmer's Guide* for details.

If you use the O_DIRECT flag, writes to the file take place directly from your program's buffer, and the data is not copied to a buffer in the kernel first. Because the filesystem cache is bypassed, your application must manage buffer alignment and block size specification. To use O_DIRECT, you must transfer data in quantities that are multiples of the filesystem block size. The following code shows how to query the filesystem block size and system DMA transfer size limit.

```
struct dioattr da;
struct stat fileStat;
char *ioFileName = “videodata”;
int ioBlockSize, ioMaxXferSize;

ioFileFD = open(ioFileName,O_DIRECT | O_RDWR | O_CREAT | O_TRUNC,0644);
if (ioFileFD < 0)
    return(DM_FAILURE);
if (fcntl(ioFileFD, F_DIOINFO, &da) < 0)
    return(DM_FAILURE);
ioBlockSize = da.d_miniosz;
ioMaxXferSize = da.d_maxiosz;
```

The two important constraints of direct I/O with XFS are memory address alignment and buffer length. Direct I/O requires all memory addresses to be page-aligned. XFS requires buffers to be allocated as a multiple of the filesystem block size, *ioBlockSize*. DMbuffers are guaranteed to be page-aligned, but to ensure that the buffers are properly padded, you must set the buffer size, *bytesPerXfer,* to the size of the image data you will transfer rounded up to the nearest multiple of *ioBlockSize*.

```
VLServer vlServer;
VLPath vlPath;
DMparams * paramsList;
int dmBufferPoolSize = 30;  /* 1 second of video */
int vlBytesPerImage = vlGetTransferSize(vlServer, vlPath);
int ioBlocksPerImage = (vlBytesPerImage+ioBlockSize - 1) / ioBlockSize;
int bytesPerXfer = ioBlocksPerImage * ioBlockSize;

if (dmBufferSetPoolDefaults(paramsList,dmBufferPoolSize,bytesPerXfer,
    DM_TRUE, DM_TRUE) == DM_FAILURE) {
    fprintf(stderr, “error setting pool defaults\n”);
    return(DM_FAILURE);
}
```

All SGI systems have a configurable maximum DMA transfer size (see systune(1M)). This value should be compared with the user’s I/O request size.

```
if (bytesPerXfer > ioMaxXferSize) {
    fprintf(“DMA request size is too small. Reconfigure with
             systune()\n”);
    return(DM_FAILURE);

}
```

Scatter/Gather I/O

As shown in DMPG Chapter 5, “Digital Media Buffers,” and in the example programs *divo_vidtomem.c* and *divo_memtovid.c,* video data is generally transported to or from DMbuffers one image at a time using standard write and read functions that specify the number of bytes and a pointer to a buffer. However, large reads and writes can usually increase I/O performance. This technique reduces the number of transactions performed between the application, operating system, and I/O device, and can allow the device to optimize some of its activities. These advantages are particularly true with disk arrays.

Since the DMbuffer’s memory pages are not guaranteed to be contiguous, standard reads or writes cannot be made across multiple buffers. The readv(2) and writev(2) interfaces allow an application to provide a list of I/O vectors, which are data structures consisting of an address and byte-count pair. Because the list of vectors is submitted to the operating system as a unit, it can be treated as a single large I/O request. Using **readv()** and **writev()** with direct I/O is particularly efficient.

The restrictions on buffer alignment and block size for **readv()**/**writev()** are similar to those of direct I/O. The address for each I/O vector must be page-aligned, but the length of each I/O vector must be a multiple of the system page size, rather than the filesystem block size, as is the case with direct I/O. Thus, the easy solution is to always use the larger of the two values, page size or filesystem block size. This requirement wastes some space, but is necessary to maintain functionality and performance. This calculation must be performed before dmBufferSetPoolDefaults(3dm) is called.

```
int ioAlignment, ioBlockSize;
ioAlignment = getpagesize();
if (ioAlignment > ioBlockSize)
    ioBlockSize = ioAlignment;
```

The maximum allowable number of I/O vectors can be queried with sysconf(3C).

```
int ioVecCount=2; /* set default to two images */
long ioVecCountMax;
/* check for range */
    ioVecCountMax = sysconf(_SC_IOV_MAX);
    if (ioVecCount > ioVecCountMax) {
        ioVecCount = ioVecCountMax;
        fprintf(stderr, “cannot create more than %d I/O vectors\n”,
            ioVecCountMax);
    }
    else if (ioVecCount <= 0)
        ioVecCount = 2;
```

The aggregate size of all the I/O vectors cannot exceed the maximum DMA transfer size, so you must check for this condition and adjust the number of I/O vectors if necessary:

```
int ioVecCount=2; /* set default to two images */
if (bytesPerXfer * ioVecCount > ioMaxXferSize)
   ioVecCount = ioMaxXferSize/bytesPerXfer;
```

When you work with video data using **readv()**/**writev()**, it is much easier to manage frames or an even number of fields with one I/O vector per field or frame. Most SGI video devices can support either field or frame mode, which is selected with the VL_CAPTURE_TYPE device control (see Chapter 4, "Video I/O Concepts" in the *Digital Media Programming Guide*). Hereafter, the term video image refers to a video data quantum: field or frame, depending on how the hardware is set up. The restriction of working on frame or even field boundaries is also relevant to the data file format, which is discussed at the end of this appendix.

The following code fragment illustrates writing to disk. Upon the successful capture of a video image, the VLTransferComplete event is placed on the event queue. A pointer to a valid DMbuffer is returned by vlDMBufferGetValid(3dm); then the actual video data is mapped into user space. Data is not written to disk until there are enough video images to complete an I/O vector.

```
case VLTransferComplete:

   /* loop until we get a valid buffer */
   while (((retval = vlDMBufferGetValid(vlServer, vlPath, vlDrnNode,
    &dmBuffers[dmbuffer_index])) != VLSuccess) && (vlErrno == VLAgain))
                       sginap(1);
    if (retval == VLSuccess) {
        /* map data to I/O vectors */
        (videoData+iov_index)->iov_base =
                dmBufferMapData(dmBuffers[dmbuffer_index]);
        (videoData+iov_index)->iov_len  = bytesPerXfer;

        /* increment the buffer index for the next image */
        dmbuffer_index = (dmbuffer_index+1) % dmBufferPoolSize;

        /* write data to disk when we have enough I/O vectors */
        if (!(++iov_index % ioVecCount)) {
            first_index = vlXferCount - iov_index + 1
            dataOffset = (off64_t) vlXferCount *
                         (off64_t) bytesPerXfer;
            /* seek to the correct position in the file, must use
             * lseek64() as the 64-bit offset value is necessary for
```

```
                 * XFS filesystems larger than 2 gigabytes
                 */
                if (lseek64(ioFileFD, dataOffset, SEEK_SET) != dataOffset)
                    return(DM_FAILURE);

                /* write the I/O vector to disk */
                if (writev(ioFileFD, videoData, ioVecCount) < 0)
                    return(DM_FAILURE);

                /* the dmbuffers are managed as a ring buffer,
                 * dmbuffer_free_index points to the next free buffer */
                for (i=0, dmbuffer_free_index = (dmbuffer_index - 1);
                        i < iov_index; i++, dmbuffer_free_index--) {
                    if (dmbuffer_free_index < 0)
                        dmbuffer_free_index = dmbuffer_max_index;

                    dmBufferFree(dmBuffers[dmbuffer_free_index]);
                }

                /* write the QuickTime movie offset data */
                if (mvFormat == MV_FORMAT_QT) {
                    last_index = first_index + iov_index;
                    if (write_qt_offset_data() == DM_FAILURE)
                        return(DM_FAILURE);
                }

                /* reset the I/O vector index */
                iov_index = 0;
            }
            vlXferCount++;
        }
        else {
            fprintf(stderr, "cannot get a valid DM buffer: %s\n",
                    vlStrError(vlErrno));
        }
        break;
```

The example for reading data from disk can be found in */usr/share/src/dmedia/tools*.

Multiprocessing

Some aspects of digital media programming lend themselves to a multiprocessing programming model. On a multiprocessor system, the various tasks of moving multiple streams of video and audio data on and off disk, serial I/O control of external video equipment and input devices, processing of video data, or the transport of video data in and out of the graphics framebuffer can be assigned to different processors. New processes must be created with all virtual space attributes (shared memory, mapped files, data space) shared. The following fragment illustrates how to create a process to perform video recording.

```
if ((video_recorder_pid = sproc(video_recorder, PR_SADDR|PR_SFDS))<0){
    perror(“video_recorder”);
    exit(DM_FAILURE);
}
```

If you use multiprocessing, note the following caveats:

- When VL calls are made, VL objects such as VLServer, VLPath, VLNode, and so on, are passed through the kernel to the video driver. However, you cannot create any VL objects without first creating a VLServer, from which everything else is instanced.
- In a process share group, only one VL call whose arguments derive from a VLServer can execute at a time. This requirement applies even to VL calls that do not explicitly take a VLServer as an argument (for example, vlBufferAdvise(3dm)).
- You can use objects derived from a given VLServer in any number of threads as long as you use a locking scheme, such as usnewsema(3P) or pthread_mutex_init(3P), to make the use in each thread mutually exclusive of a use in any of the other threads.

The VL error state, returned by vlGetErrno(3dm), is global to a share group, not per VLServer. If a VL call using one VLServer in one thread executes simultaneously with a VL call using another VLServer in another thread, both calls try to set the error state returned by **vlGetErrno()**. This call should be global only to the thread, not to the entire process share group.

Asynchronous I/O

Asynchronous I/O allows an application to process multiple read or write requests simultaneously. On SGI platforms, asynchronous I/O is available through the *aio* facility. This facility, based on sproc(2)'ed processes, provides all of the benefits of multiprocessing for free. Because multiple I/O requests might be outstanding, when you use asynchronous I/O, the round-trip delay between making a request, having it serviced, and issuing another request is removed. Any process-scheduling delay between these steps is also eliminated.

Because asynchronous I/O operations complete out of sequence, the application must keep track of the order in which data appears in the DMbuffers. DMbuffers are contained in a DMbufferPool; the pool itself is unordered and buffers can be obtained and returned to the pool in any order. Ordering is achieved by a first-in-first-out queue and maintained only while the buffers reside in the queue. The application is free to impose any processing order once buffers are dequeued.

File Formats

Each time a DMbuffer is written to disk, an offset must be recorded for the QuickTime file.

```
MVid theMovie;
MVid mvImageTrack;
off64_t mvFieldGap= bytesPerXfer - vlBytesPerImage;
MVtimescale mvImageTimeScale=MV_IMAGE_TIME_SCALE_NTSC;
int mvFrameTime = 1001;  /* for NTSC */
off64_t meta_data_offset;
int mv_frame_index;
MVframe mv_dummy_offset;
int i;

mvInsertTrackDataAtOffset(
        mvImageTrack,
        1,
        (MVtime) (i * mvFrameTime),
        (MVtime)  mvFrameTime,
        mvImageTimeScale,
        (off64_t) meta_data_offset,
        vlBytesPerImage,
        MV_FRAMETYPE_KEY,
        0)
```

```
/* get the index for the libmovie data corresponding to this field.
 * this is necessary in order to set the gap and field sizes for the
 * fields in the frame.*/
mvGetTrackDataIndexAtTime(
        mvImageTrack,
        (MVtime) (i * mvFrameTime),
        mvImageTimeScale,
        &mv_frame_index,
        &mv_dummy_offset)

/* tell libmovie the field gap and sizes for each field in the frame */
mvSetTrackDataFieldInfo(
        mvImageTrack,
        mv_frame_index,
        vlBytesPerImage,      /* absolute size of field 1 */
        mvFieldGap,           /* gap between fields */
        vlBytesPerImage)      /* absolute size of field 2 */
```

When data recording completes, the following function must be called to close the QuickTime file properly.

```
write_qt_file_header(void)
{
    int flags;

    /* if direct I/O mode is enabled, disable it because the
     * movie library does not do direct I/O
     */
    if (ioFileFD) {
        fsync(ioFileFD);
        flags = fcntl(ioFileFD, F_GETFL);
        flags &= ~FDIRECT;
        if (fcntl(ioFileFD, F_SETFL, flags) < 0) {
            fprintf(stderr,“unable to reset direct I/O file status\n”);
            return(DM_FAILURE);
        }
    }

    if (mvClose(theMovie) == DM_FAILURE) {
        fprintf(stderr, “unable to write movie file header %s\n”,
                    mvGetErrorStr(mvGetErrno()));
        return(DM_FAILURE);
    }
}
```

DV and DVCPRO Standards

DVCPRO is based on the original DV consumer-based video format. This appendix gives an overview of both the DV and DVCPRO digital video formats. It also compares various available video formats in the industry. This appendix consists of these sections:

DV Standard

DV is a ubiquitous video format developed by the HD Digital VCR Consortium, which is composed of more than 50 companies, including Sony, Matsushita, Philips, Thomson, Toshiba, Hitachi, JVC, Sanyo, Sharp, and Mitsubishi. See the "Specifications of Consumer-Use Digital VCRs Using 6.3 mm Magnetic tape" (distributed to Consortium members) for complete details of DV.

The DV initial specification was introduced on July 1, 1993, and now covers both Standard Definition (SD) and High Definition (HD). Table F-1 shows some SD specifications; Table F-2 shows audio recording method specifications.

Table F-1 DV Specifications: General

Specification	Value
Sampling frequency	Y: 13.5 MHz Cb, Cr: 6.75 MHz
Quantization	8-bit
Compression ratio	5:1
Video rate	25 Mbps
Recording time	Standard DV: 270 minutes Mini DV: 60 minutes
Digital compression method	Discrete Cosine Transform (DCT)

Table F-2 DV Specifications: Audio Recording Method

Specification	Value			
PCM Digital	2CH	2CH	2CH	4CH
Sampling frequency	48 KHz	44.1 KHz	32 KHz	32 KHz
Quantization	16-bit	16-bit	16-bit	12-bit

DV Sampling

DV supports 4:1:1 for 525/60 and 4:2:0 for 625/50. The bit rate is kept constant and is dependent on luminance sampling structure, quantization depth, ratio of luminance to color difference samples, number of frames per second, and the compression ratio (5:1 for DV).

Note: For sampling pattern illustrations and descriptions, see “Sampling Patterns” on page 105 in Appendix C.

DV Compression

Figure F-1 summarizes the compression process used in DV.

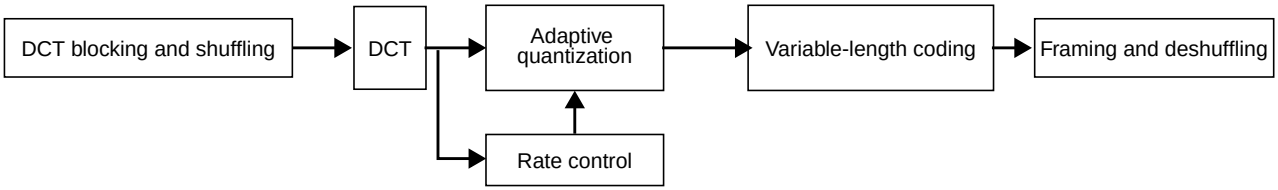

Figure F-1 DV Compression

The steps performed by DV are as follows:

1. Pixels are assembled into appropriate blocks.
2. DCT is applied to each block.
3. Macroblocks of six DCT blocks are formed. These macroblocks are then selected at various places of the picture (hence the shuffling process) and categorized into classes of importance.

   This method better preserves the fine details of the picture throughout the entire frame.
4. DV applies the popular compression method Discrete Cosine Transform (DCT) to each macroblock.

   The assembling process combines four luminance blocks with two chrominance blocks (4:1:1 or 4:2:0) to form a macroblock. Five macroblocks are selected from five different regions in a picture to form a video segment. DCT is applied to each macroblock.

   This mathematical formula transforms each block's pixel signal levels into frequency domain coefficients. The coefficients are quantized at different levels to achieve the best compression without affecting quality. The rate control process makes sure a constant bit rate is achieved by varying the quantization scale.
5. Variable-length coding (VLC, or Huffman encoding) is applied to shorten the data length. Deshuffling puts VLC-coded data into 77-byte packets. Highly detailed macroblocks requiring more data space might use other less-detailed macroblocks to store information.

Overall, this compression process differs from MPEG or M-JPEG compression in these ways:

- DV compresses a picture using only intraframe, and each frame is compressed to a fixed size. This method makes editing easier.
- Intraframe avoids all motion artifacts associated with interframe coding, a trade-off for its higher bandwidth requirement and a lower compression ratio.

On one hand, considerable effort was expended to make MPEG codec nonsymmetrical: the encoder is significantly more complex than the decoder, making distribution appropriate. On the other hand, DV is designed to be symmetrical, compact, and cost-effective. The same chipset can perform both compression and decompression.

For tape requirements, DV uses 1/4-inch wide tapes at a track pitch of 10 µm. Each track has four primary data areas:

- Insert and track information (ITI): this section includes location information, track pitch, servo information, and the application ID of a track (APT). The APT indicates the data structure of that particular track.
- Audio: this section contains audio information, audio auxiliary data, and an APT.
- Video: compressed video data, video auxiliary data, and an APT are stored in this section.
- SubCode: SMPTE/EBU time code, absolute track number, and APT are stored here.

Additional space (editing guard bands) is also included between each of the areas discussed above.

A 525/60 system has 10 tracks per frame; a 625/50 system has 12 tracks per frame. Each track corresponds to a horizontal band of the picture.

For audio, DV allows several modes:

- The 48-KHz mode captures two channels of AES/EBU professional digital audio using 16-bit linear samples.
- Two channels are also possible for 44.1 KHz and 32 KHz, also using 16-bit linear samples.
- A four-channel mode is available for 32 KHz, using 12-bit nonlinear samples. The audio is divided into frames, which are grouped onto video tracks. For example, in 525/60, channel 1 audio is grouped on tracks 0-4, and channel 2 audio is grouped on tracks 5-9.

DVCPRO Standard

Developed by Panasonic, DVCPRO is a nonproprietary video format standard for the production, broadcast, and distribution of digital television video. It is also known as SMPTE D-7; see that documentation for full details of the DVCPRO video format.

DVCPRO is built on the DV format (6.35 mm), but is simpler and thus more suitable for the professional and broadcast markets. Instead of metal evaporated tape, metal particle tape is used. Track pitch is increased from 10 microns (DV) to 18 microns. Tape speed is increased from ~18.8 mm per second to ~33.8 mm per second.

As a subset of DV, DVCPRO uses the same video compression method as DV for 525/60. The video signal is sampled at 13.5 MHz for luminance (Y) and 6.75 MHz for color differences (Cb and Cr). Then the sampled video data is compressed via DCT, 8-bit quantization, and VLC.

Audio is recorded via two 16-bit audio channels at 48 KHz, synchronous with video. In addition, another cue audio track is included to allow intelligible audio during variable play speeds. A control track is also added to help minimize tape preroll and more accurate frame editing.

For 625/50, DVCPRO uses a 4:1:1 sampling structure instead of 4:2:0 as used in DV. The DIVO-DVC option board supports 4:1:1 sampling for both 525/60 and 625/50, and supports 4:2:0 625/50 for DV video format. VITC carries additional information and is recorded in the subcode and video auxiliary regions, like DV.

Despite its tape-oriented technology, DVCPRO, like DV, is an open format designed for use with nonlinear editors and server-based systems. As with DV, DVCPRO has a 4X transfer speed record-and-play capability.

DVCPRO for 525/60 is backward-compatible with DVC; you can play 525/60 DVC tape on a DVCPRO tape deck.

At 25 Mbps and using a 4:1:1 sampling structure, which yields a 5:1 compression ratio, the DIVO-DVC board can accomplish a wide range of work flexibly and cost-effectively.

For information on DVCPRO, see

- Proposed SMPTE standard 305M for Television - Serial Data Transport Interface
- Proposed SMPTE Standard 306M for Television Digital Recording — 6.35-mm Type D-7 Component Format — Video Compression at 25 Mb/s — 525/60 and 625/50
- Proposed SMPTE Standard SMPTE xxxx for Television Data Stream Format for the Exchange of DV Based Audio, Data and Compressed Video over a Serial Data Transport Interface (SDTI)

SMPTE's web site is http://www.smpte.org/.

DV Technology Comparison

Table F-3 compares DVT options.

Table F-3 DV Technology Comparison

Feature	DV (Sony, JVC, Panasonic)	DVCam (Sony)	DVCPRO (Panasonic)	DIGITAL-S (JVC)	DVCPRO50 (Panasonic)
Bit rate (Mbps)	25	25	25	50	50
525/60 subsampling	411	411	411	422	422
625/50 subsampling	420	420	411	422	422
525/60 frame size	720 x 480	720 x 480	720 x 480	720 x 480	720 x 487.5
625/50 frame size	720 x 576	720 x 576	720 x 576	720 x 576	720 x 583.5
16:9 display	Not on all models	Yes	Yes	Yes	Yes
Extended play (12.5 Mpbs)	Yes	No	No	No	No
Audio frequency (KHz)	32, 44.1, 48	32, 48 (44.1 nonpro mode)	48	48	48
Audio mode	Locked and unlocked	Unlocked/locked	Locked	Locked	Locked
Audio channels	2	2 and 4	2	2	4
Aux packs	Main	Main, timecode, take, mark in/out, reel, scene, ok	Main, timecode	Main, timecode	Main, timecode

Table F-3 (continued) DV Technology Comparison

Feature	DV (Sony, JVC, Panasonic)	DVCam (Sony)	DVCPRO (Panasonic)	DIGITAL-S (JVC)	DVCPRO50 (Panasonic)
Tape type	ME	ME	MP	MP	MP
Track pitch (µm)	10	15	18 (plays 10+15)	20	18 (plays 10+15)
Tape speed (min/sec)	18.8	29.193	33.8	57.737	525: 67.640 625:67.708

GPI Interface (DIVO Option Only)

For each video pipe of the DIVO option board, the General Purpose Interface (GPI) provides two channels of input and output trigger signal pairs. This appendix explains

Note: This appendix is pertinent to the DIVO option board only; the information in it does not apply to the DIVO-DVC option board.

GPI Headers (DIVO Option Board Only)

The DIVO board has two GPI headers. Each GPI header (row of four pins) configures one of four receiver channels: two channels for GPI in and two channels for GPI out. Figure G-1 shows the location of the jumper pins on the DIVO option board.

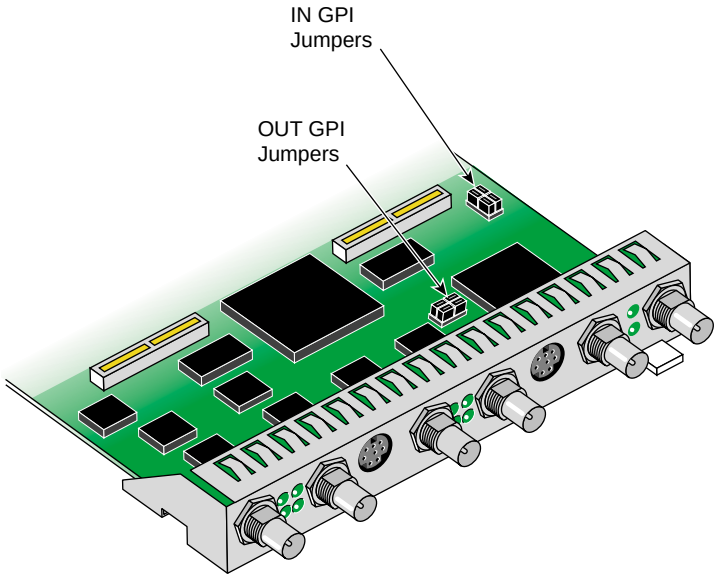

Figure G-1 GPI Jumper Locations (Factory Setting), DIVO Option Only

Note: This information on GPI jumpers pertains to the DIVO option board only; it does not apply to the DIVO-DVC option board.

Note that the jumpers for the **OUT GPI** connector are near the **OUT GPI** mini-DIN connector, while the jumpers for the **IN GPI** connector are relatively far away from the **IN GPI** connector.

For the factory setting of switch closure mode, two jumpers are factory-installed, shorting pins 1-2 and pins 3-4. These jumpers need not be moved unless you wish to use current sense mode. You can choose to mix the modes for the various channels. This reconfiguration is typically performed by a SGI System Service Engineer when the DIVO

board is installed in the chassis. Switch closure and current sense modes are explained in “GPI Receiver, Switch Closure Mode, and Current Sense Mode” on page 152.

Figure G-2 shows GPI headers and jumpering. The printed circuit board (PCB) reference designators are included to aid identification of the header associated with each GPI receiver channel.

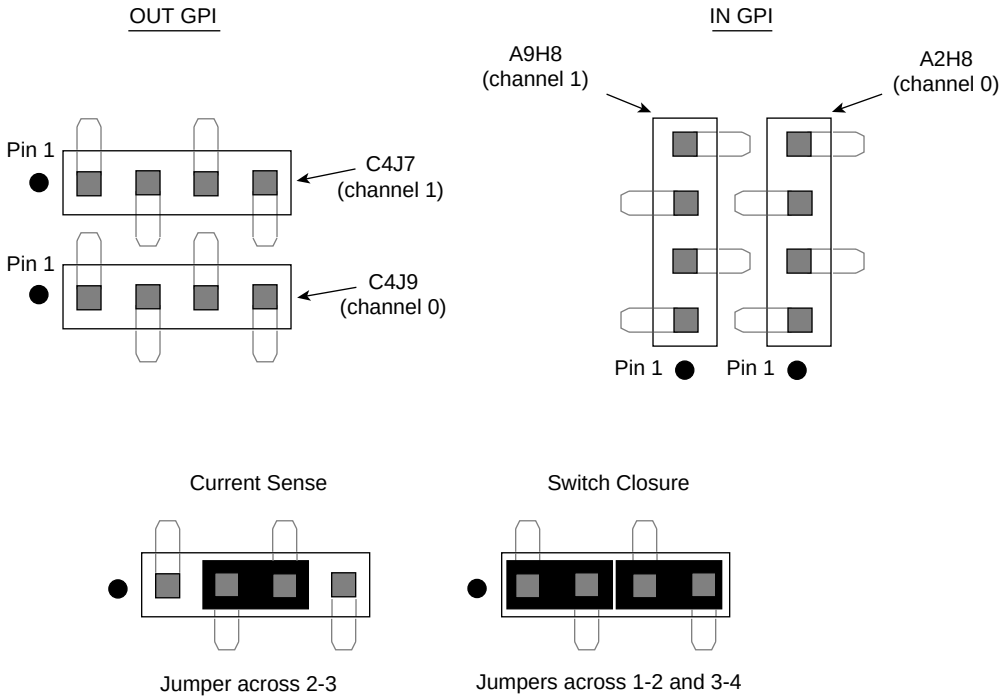

Figure G-2 Example GPI Interface (DIVO Option Only)

Note: This information on GPI jumpers pertains to the DIVO option board only; it does not apply to the DIVO-DVC option board.

For information on VL controls for configuring the GPI ports, see “VL Support for the General-Purpose Interface (GPI)” in Chapter 2. For information on the GPI connectors and transmitter, see “GPI Interface” in Appendix A.

GPI Receiver, Switch Closure Mode, and Current Sense Mode

GPI contact closure receive (CCR) inputs use an optical isolator device to provide a means of electrical isolation from source equipment. The device consists of a bidirectional input LED optically coupled to a bipolar transistor. A voltage pulse applied across the CCR+/- pins causes the LED to become forward-biased and to produce a GPI trigger to the computer.

Table G-1 summarizes electrical specifications for the GPI receiver optoisolator.

Table G-1 GPI Receiver Input Optoisolator

Parameter	Value
Forward voltage (V_F)	1.55 V, 1.2 V typical (IF = 10 mA)
Continuous forward current (I_F)	30 mA
Peak forward current	1000 mA (10 us duration, 1% DC)
Reverse current (I_R)	0.1 uA, 100 uA maximum (V_R = 6 V)
Isolation surge voltage (V_{10})	2500 VAC_{RMS} (t = 1 min)

Depending on jumpering (see "GPI Headers (DIVO Option Board Only)" on page 150), the DIVO board GPI receiver can be set to switch closure or current sense mode, as discussed in these sections:

- "GPI Receiver Switch Closure Mode (Factory Setting)" on page 153
- "GPI Receiver Current Sense Mode (DIVO Option Board Only)" on page 154

Note: Setting the board for current sense mode is possible only for the DIVO option board, and only at the time of installation. Current sense mode does not apply to the DIVO-DVC option board.

GPI Receiver Switch Closure Mode (Factory Setting)

Figure G-3 shows switch closure jumpering, which creates a digital pulse.

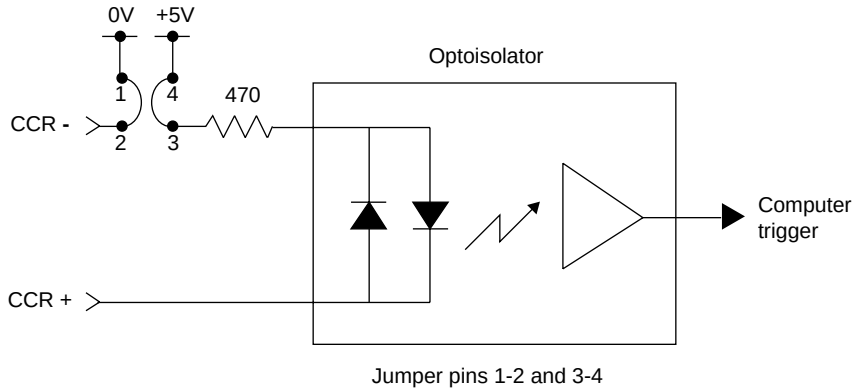

Figure G-3 Jumpering for GPI Switch Closure (Factory Setting)

In switch closure mode, the +5 V power supply and ground of the DIVO board are not electrically isolated from the chassis of the source equipment.

For switch closure mode, the GPI receiver can be interfaced to the source equipment by tying the CCR+ and CCR- terminals across the output terminals of an optoisolator, solid-state relay, or any device that acts like a single-pole contact switch. A GPI trigger is generated as long as the source switch is closed.

Note: Polarity of the CCR+/- signals must be observed for the source equipment in switch closure mode.

GPI Receiver Current Sense Mode (DIVO Option Board Only)

Figure G-4 shows current sense jumpering.

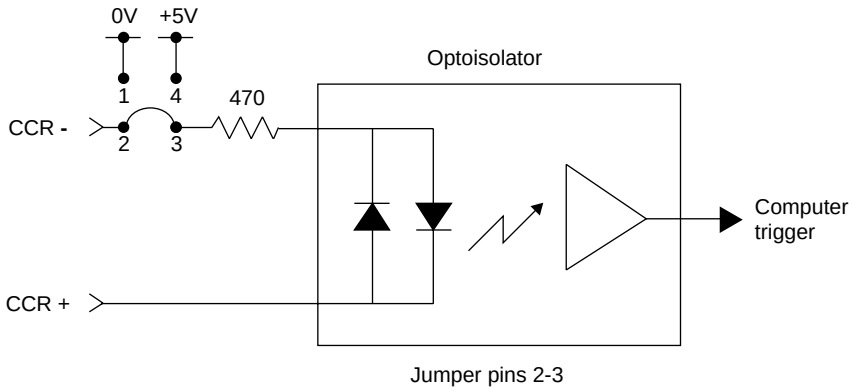

Figure G-4 Jumpering for GPI Current Sense Mode, DIVO Option Only

Note: Setting the board for current sense mode is possible only for the DIVO option board, and only at the time of installation. Current sense mode does not apply to the DIVO-DVC option board.

In current sense mode, the DIVO board is electrically isolated from the chassis of the source equipment.

For current sense mode, the CCR+ and CCR- signals can be interfaced by tying the CCR+ terminal to the output of a TTL or CMOS logic device, and by tying the CCR- terminal to GND of the source equipment. Whenever the logic device is sourcing current (driving a logic high), a GPI trigger is generated.

In current sense mode, the logic sense can be swapped (inverted) by moving the CCR- signal from GND to the logic power supply (typically VCC) of the source equipment. The CCR+ signal remains connected to the output of the logic device; however, in this configuration an open collector type device can be used. Whenever the logic device is sinking current (driving a logic low), a GPI trigger is generated.

Index

D

N

O

P

R

S

T

U

V

X

Y

Z