OCTANE™ Compression Programmer's Guide

Document Number 007-3514-001

CONTRIBUTORS

Written by Carolyn Curtis
Illustrated by Carolyn Curtis
Production by Kirsten Johnson
Engineering contributions by Nat Gurumwoorthy, Gregory Poist, Matthew Hall, Chuck Jerian, and Howard Chartock
St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

© 1997, Silicon Graphics, Inc.— All Rights Reserved
The contents of this document may not be copied or duplicated in any form, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

RESTRICTED RIGHTS LEGEND
Use, duplication, or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and/or in similar or successor clauses in the FAR, or in the DOD or NASA FAR Supplement. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94043-1389.

Silicon Graphics, the Silicon Graphics logo, OpenGL, Indigo, and IRIS are registered trademarks and OCTANE, Origin, Indigo^2, Indigo^2 IMPACT, Cosmo Compress, and Galileo Video are trademarks of Silicon Graphics, Inc. MIPS and R3000 are registered trademarks of MIPS Technologies, Inc. X Window System is a trademark of Massachusetts Institute of Technology.

OCTANE™ Compression Programmer's Guide
Document Number 007-3514-001

Contents

List of Figures

List of Tables

About This Guide

The OCTANE™ Compression motion JPEG option card from Silicon Graphics® provides two independent channels of full-resolution, full-motion, real-time video compression or decompression for the OCTANE desktop workstation.

OCTANE Compression fully utilizes all calls and controls in the Silicon Graphics Compression Library (CL), and works with other Silicon Graphics programming libraries, such as the Video Library (VL).

This guide explains features of the CL and VL for OCTANE Compression and gives step-by-step instructions for creating programs using CL, VL, or both that make use of OCTANE Compression board capabilities.

Audience

This guide is written for the sophisticated user with a background in C programming who wishes to develop programs for OCTANE Compression capabilities, with or without interaction with its on-board video capability or the OCTANE Digital Video option.

Structure of This Document

This guide contains the following chapters and appendix:

- Chapter 1, "OCTANE Compression Features and Capabilities," explains how the OCTANE Compression board works with the Compression Library and presents features of the CL.
- Chapter 2, "Programming With the Compression Library," presents the CL's three interfaces for compressing and decompressing image, audio, and video data.

- Chapter 3, “Programming With the Video Library,” explains how to open a connection to the video daemon and set up a data path, how to set data transfer parameters, how to display video data onscreen, how to transfer video data, and how to end data transfer by presenting an annotated sample program that displays live video input in a graphics window.
- Chapter 4, “Using the Compression Library With OCTANE Compression,” explains how to add OCTANE Compression support to an application, use OCTANE Compression image formats, get compressed image information, specify memory-to-memory compression and decompression, and how to compress and decompress video through external connections to analog video or OCTANE Compression.
- Chapter 5, “Using Video Library Controls,” explains VL control type and values, VL control fraction ranges, VL control classes, and VL control groupings.
- Chapter 6, “Using Compression Library Parameters,” describes the Compression Library parameters and summarizes how to use them.
- Chapter 7, “Using Compression Library Algorithms,” explains how to query and use Compression Library algorithms.
- Chapter 8, “Differences Between OCTANE Compression and Earlier Silicon Graphics Compression Options,” explains hardware and software differences between the two options for those porting applications from these earlier products.
- Appendix A, “Video Library Controls and Compression Library Parameters for the OCTANE Compression Option,” summarizes the VL controls and CL parameters for OCTANE Compression.

An index completes this guide.

Other Documents

The following online documents are also included with the OCTANE Compression option:

- *OCTANE Digital Video and OCTANE Compression Installation Guide* (007-3466-001)
- *Digital Media Programming Guide* (007-1799-060 or later)

Conventions

These type conventions and symbols are used in this guide:

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

Chapter 1

OCTANE Compression Features and Capabilities

The OCTANE Compression motion JPEG option card from Silicon Graphics provides two independent channels for compression and decompression for the OCTANE desktop workstation. Besides compressing and decompressing still images, OCTANE Compression enables an OCTANE workstation to input and output compressed video and record it to disk or videotape. When OCTANE Digital Video is also installed in the workstation, you can input and output CCIR 601 digital video.

In this chapter:

- "OCTANE Compression Features" presents specific features of the hardware.
- "OCTANE Compression and Video Options" summarizes how the compression board and the OCTANE Digital Video option interact.
- "OCTANE Compression and the Compression Library" introduces the CL.
- "OCTANE Compression and the Video Library" introduces the VL.

OCTANE Compression Features

Designed to work with the OCTANE Digital Video option, OCTANE Compression overcomes the obstacle presented by the colossal data streams that video sources generate. Thus, OCTANE Compression is a powerful tool for video production, digital video distribution, motion video analysis, and video-based training. OCTANE Compression is an integral part of the digital studio that combines leading computer graphics, image processing, digital video, and high-quality video in an efficient desktop environment.

Note: The OCTANE Compression option does not perform audio compression.

For applications that demand broadcast quality, OCTANE Compression with OCTANE Digital Video allows compressed digital video streams to be used as elements in sophisticated effects such as real-time keying, blending, and video texture mapping. The option provides an ideal environment for broadcast-quality nonlinear editing, spot playback, and still storage.

OCTANE Compression is an integral part of the Silicon Studio solution for film and video production, which integrates 2D and 3D graphics, image processing, digital audio, and high-quality video in a single environment.

OCTANE Compression features include

- the ability to encode or decode the board's two channels in any combination
- capture and playback of full-resolution, full-motion video to and from memory or disk in real time:
  - 60 fields, or 30 frames, per second compression and decompression of full-resolution NTSC video
  - 50 fields, or 25 frames, per second compression and decompression of full-resolution PAL video
  - single-frame compression and decompression
- composite or S-Video capture, and playback with genlock capability
- compression ratios as low as 2:1
- during real-time compression, scaling of full-size fields by half in the horizontal or vertical direction or both
- real-time color-space conversion in memory-to-memory decompression or uncompressed video capture or playback modes
- during decompressing to main memory or uncompressed video capture, image scaling for flexible viewing of video clips and for processing transitions and effects
- compatibility with all OCTANE graphics solutions
- data formats: 8-bit per component 4:2:2 YUV, XBGR, or RGBX (32 bits per pixel, 8 bits per component)
- capture of uncompressed data to memory; playback of uncompressed data from memory

OCTANE Compression has these modes of operation:

- capturing uncompressed video from the base analog input or optional OCTANE Digital Video option into a memory buffer
- playing back of uncompressed video from a memory buffer to the base analog output or optional OCTANE Digital Video option
- compressing video from the base analog input or optional OCTANE Digital Video option into a memory buffer
- decompressing video from a buffer to the base analog output or optional OCTANE Digital Video option
- compressing an image stored in memory into another area of memory
- decompressing a stored compressed image and placing it into another area of memory

Because of the high data rates produced by video sources, your priorities might alternate between image quality on the one hand and storage size and transmission bandwidth on the other. OCTANE Compression adjusts to your needs with a wide range of compression ratios under complete software control.

OCTANE Compression works with the Compression Library, a complete API for compressing single images, video-streaming applications, and more.

OCTANE Compression and Video Options

The OCTANE Compression option can be used as a simple analog capture and playback device for video, or with the OCTANE Digital Video option for capture and playback and for CCIR 601 digital video.

The OCTANE Compression option's real-time compression and decompression lets you perform nonlinear editing and real-time playback from disk of special effects, composites, and animations. The OCTANE Compression option uses JPEG, the ideal compression algorithm for postproduction processes because it preserves individual video frames.

OCTANE Compression and the Compression Library

The Silicon Graphics Compression Library (CL) was designed to exploit the full capabilities of the OCTANE Compression option:

- compression ratios
- data formats
- in conjunction with the Video Library, capture and playback to and from video destinations
- digital movie recording, editing, and playback

The CL provides three interfaces, for successively more complex compression: a still image API for single images, a sequential access API for video-streaming applications, and a buffered interface. Chapter 2, "Programming With the Compression Library," explains these interfaces in detail.

The CL works with other Silicon Graphics Digital Media libraries—Audio Library (AL) and Movie Library (ML)—as well as the Video Library (VL).

Note: Although the CL supports audio compression, the OCTANE Compression board does not.

OCTANE Compression and the Video Library

The Video Library provides a software interface to the OCTANE Compression board, which lets applications

- capture live video in system memory
- encode graphics to video in real time
- produce full-rate video output

The Video Library (VL) is a collection of device-independent and device-dependent C language calls for Silicon Graphics workstations equipped with video options. The VL provides generic video tools, including simple tools for importing and exporting digital data to and from Silicon Graphics systems or third-party video devices that adhere to the Silicon Graphics architectural model for video devices.

Chapter 3, "Programming With the Video Library," explains the basics of using the VL to create video programs for OCTANE Compression.

Note: See page 10 for information on the order of operation between CL and VL calls.

Chapter 2

Programming With the Compression Library

This chapter describes how to use the Compression Library API to compress and decompress image and video data. The CL provides three interfaces for successively more complex compression:

- still image API for single images
- sequential access API for video-streaming applications where the input is live, or where there is no control over playback and the amount of compressed data for each frame is known in advance
- buffered interface that includes the calls of the sequential interface, plus buffer-management routines to access compressed data and uncompressed framebuffers

Note: Using the CL with video options is explained in detail in Chapter 4, "Using the Compression Library With OCTANE Compression."

In this chapter:

- "Error Handling" describes the CL error-handling facility.
- "Opening a Compression Session" explains the steps required for starting a session.
- "Using the Still Image Interface" explains how to compress still images with a single call.
- "Using the Sequential Frame Interface" explains how to compress or decompress sequential data using a compressor or decompressor.
- "Using the Buffering Interface" explains how to use CL buffers.

Table 2-1 lists calls explained in this chapter.

Table 2-1 Compression Library Calls

Compression and Decompression	Buffers	Miscellaneous
clCompress()	clCreateBuf()	clSetErrorHandler()
clDecompress()	clDestroyBuf()	clQuerySchemeFromName()
clOpenCompressor()	clQueryBufferHdl()	clQueryScheme()
clOpenDecompressor()	clQueryHandle()	clGetParams()
clCloseCompressor()	clQueryFree()	clSetParams()
clCloseDecompressor()	clUpdateHead()	clReadHeader()
clCompressImage()	clUpdateTail()	clQueryMaxHeaderSize()
clDecompressImage()	clDoneUpdatingHead()	
	clQueryValid()	
	clQuery()	
	clUpdate()	

Error Handling

In the CL, file I/O is handled by the caller. The CL has an error handler that prints error messages to *stderr*. Most CL routines return a negative error code upon failure.

You can override the default error-handling routine and establish an alternate compression error-handling routine using **clSetErrorHandler().**

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

```
CL_ErrFunc clSetErrorHandler(CL_ErrFunc efunc)
```

where

efunc is a pointer to an error handling routine declared as

```
void ErrorFunc(CLhandle handle, int code, const char*
fmt,...)
```

The returned value is a pointer to the previous error-handling routine.

The code fragment in Example 2-1 demonstrates how to silence error reporting for a section of code.

Example 2-1 Using a Custom Error-Handling Routine

```
#include <cl.h>
...
CL_ErrFunc originalErrorHandler;
void SilentCLError(CLhandle handle, int errorCode,
   const char* fmt, ...)
{
/* ignore all CL errors */
}

...
originalErrorHandler = clSetErrorHandler(silentCLError);
/* cl errors here will go unnoticed */

...
clSetErrorHandler(originalErrorHandler);
/* back to normal reporting of CL errors */
...
```

Note: If an application attempts to decompress data that is not valid JPEG data, the decompressor can hang.

Opening a Compression Session

Unlike the Cosmo Compress™ option, the OCTANE Compression option does not have a predefined scheme value; that is, no scheme pound define is specified for OCTANE Compression. Instead, applications use **clQuerySchemeFromName()** to query the CL whether a scheme with the name *impact* is available in the system.

If the scheme is available, the return from this function specifies the scheme identifier to pass to the CL routines. As other schemes are added to the Compression Library on a specific workstation, the actual value assigned to OCTANE Compression can change.

Example 2-2 Querying the Scheme Name

```
#include <cl.h>
int scheme;
CLhandle clHandle;

scheme = clQuerySchemeFromName (CL_ALG_VIDEO, "impact");
if (scheme < 0) {
    fprintf(stderr, "compression scheme ;'impact' is not configured\n");
    return;

}
clOpenCompressor (scheme, &clHandle);
```

In modes where the CL and VL interact to control the OCTANE Compression hardware, applications must follow an ordering of when events are requested. For all operations involving a CL_EXTERNAL_DEVICE, the order of startup is:

1. **vlBeginTransfer()**
2. **clCompress()** or **clDecompress()**

The call to **clCompress()** or **clDecompress()** actually starts the device operating. If the **vlBeginTransfer()** is initiated after the CL operation, indeterminate data is captured or the first fields of output are lost.

Using the Still Image Interface

Table 2-2 lists the calls explained in this section.

Table 2-2 Still Image Interface Calls

Compression	Decompression	Miscellaneous
clCompressImage()	clDecompressImage()	clQueryMaxHeaderSize()

The single image method is designed to make still image compression as simple as possible. The still image interface consists of two calls, one for compression and one for decompression. No interframe compression/decompression, such as the method that takes advantage of similarities between frames in MPEG, is possible with this interface.

A simple interface exists for compressing or decompressing still images with a single call. To compress a still image, use **clCompressImage()**, which compresses the data from the specified *frameBuffer*, stores the compressed image in *compressedData*, and stores its resulting size in *compressedBufferSize*.

Pass to **clCompressImage()** the compression scheme; the width, height, and format of the image; the desired compression ratio; pointers to reference the buffer containing the image and the buffer that is to store the compressed data; and a pointer to return the size of the compressed data.

You should allocate a buffer large enough to store the compressed data. In most cases, a buffer the size of the source image plus the maximum header size, which you can get by calling **clQueryMaxHeaderSize()**, is sufficient. When calculating the data storage of the source image, you can use the CL macro **CL_BytesPerPixel()** to determine the number of bytes per pixel for certain packing formats.

The function prototypes for the compress and decompress image routines are

```
int clCompressImage(int compressionScheme, int width,
       int height, int originalFormat, float compressionRatio,
       void *frameBuffer, int *compressedBufferSizePtr,
       void *compressedData)

int clDecompressImage(int decompressionScheme, int width,
       int height, int originalFormat,int compressedBufferSize,
       void *compressedData, void *frameBuffer)
```

where

compressionScheme
: is the compression or decompression scheme to use.

width
: is the width of the image.

height
: is the height of the image.

originalFormat is the format of the original image to (de)compress. For video, use

- CL_RGB
- CL_RGBX
- CL_RGBA
- CL_RGB332
- CL_GRAYSCALE
- CL_YUV
- CL_YUV422
- CL_YUV422DC

compressionRatio
is the target compression ratio. The resulting quality depends on the value of this parameter and on the algorithm that is used. Use 0.0 to specify a nominal value. The nominal values for some of the algorithms are

- MVC1 = 5.3:1
- JPEG = 15.0:1
- MPEG = 48.0:1

frameBuffer is a pointer to the framebuffer that contains the uncompressed image data.

compressedBufferSizePtr
is a pointer to the size, in bytes, of the compressed data buffer. If it is specified as a nonzero value, the size indicates the maximum size of the compressed data buffer. The value pointed to is overwritten by **clCompressImage()** when it returns the actual size of the compressed data.

compressedBufferSize
is the size of the compressed data in bytes.

compressedBuffer
is a pointer to the compressed data buffer.

Use **clDecompressImage()** to decompress an image. **clDecompressImage()** decompresses the data that is stored in *compressedBuffer*, whose size is *compressedBufferSize,* and stores the resulting image in *frameBuffer.*

The values of the state parameters used with the other compression library calls have no effect on these routines, but their defaults do. The arguments *width*, *height*, *originalFormat*, and *compressionRatio* function the same as the state parameters by the same names but are given as direct arguments to facilitate the single-command interface.

Example 2-3 demonstrates how to compress and decompress a color image using the JPEG algorithm. The image is 320 pixels wide by 240 pixels high and its data is in the RGBX format.

Example 2-3 Compressing and Decompressing a Single Frame

```
/* Compress and decompress a 320 by 240 RGBX image with JPEG */
int frameIndex, compressedBufferSize, maxCompressedBufferSize;
int *compressedBuffer, frameBuffer[320][240];

/* malloc a big enough buffer */
maxCompressedBufferSize = 320 * 240 * CL_BytesPerPixel(CL_RGBX)
                                + clQueryMaxHeaderSize(CL_JPEG);
compressedBuffer = (int *)malloc(maxCompressedBufferSize);

/* Compress and decompress it */
clCompressImage(CL_JPEG, 320, 240, CL_RGBX, 15.0,
    frameBuffer, &compressedBufferSize, compressedBuffer);
clDecompressImage(CL_JPEG, 320, 240, CL_RGBX,
    compressedBufferSize, compressedBuffer, frameBuffer);
```

Note: If an application attempts to decompress data that is not valid JPEG data, the decompressor can hang.

Using the Sequential Frame Interface

Table 2-3 lists the calls explained in this section.

Table 2-3 Sequential Frame Interface Calls

Compression	Decompression	Miscellaneous
clOpenCompressor()	clOpenDeompressor()	clGetParams()
clCloseCompressor()	clCloseDecompressor()	clSetParams()
clCompress()	clDecompress()	clQueryScheme()
clCompressImage()	clDecompressImage()	clReadHeader()
		clQueryMaxHeaderSize()

The sequential interface is designed for video-streaming applications where the input is live, or where there is no control over playback and the amount of compressed data for each frame is known in advance; in fact, an error is reported if insufficient data is passed.

This interface is more complex, requiring a series of compress or decompress calls to be encapsulated within an open-close block. Each compressor or decompressor keeps state information appropriate to the selected compression algorithm in parameters that you can query and set.

This section describes how to work with sequential frames of video data. See “Using the Buffering Interface” for a description of how to work with nonsequential data, or for situations where the decompression rate is different from the compression rate. See Chapter 5 of the *Digital Media Programming Guide* (007-1799-060 or later) for a complete description of buffers

Compressing a Sequence of Frames

To compress sequential data and video streams, use a *compressor*. A compressor is an abstraction that modularizes compression operations.

To compress a sequence of frames, follow these steps:

1. Open a compressor to establish the beginning of a sequence of compression calls.
2. Compress frames one at a time, storing the compressed data after each frame has been compressed.
3. Close the compressor to deallocate the resources associated with that compressor.

Each of these steps is discussed in detail in the following sections.

Opening a Compressor

Call **clOpenCompressor()** to open a compressor for a given algorithm. Its function prototype is

```
int clOpenCompressor(int scheme, CLhandle *handlePtr)
```

where

scheme is the compression scheme to use.

handlePtr is a pointer, which is overwritten by the returned handle of the compressor that is used by subsequent calls to identify the compressor.

More than one compressor can be open at a time. Use the handle that is returned in *handle* to identify a specific compressor.

Compressing Frames

After a compressor has been opened, call **clCompress()** to compress the data. Pass to **clCompress()** the handle returned by **clOpenCompressor()**, the number of frames to be compressed, and pointers to reference the framebuffer containing the data frames, the size of the data, and the location of the buffer that is to store the compressed data.

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

```
int clCompress(CLhandle handle, int numberOfFrames,
               void *frameBuffer, int *compressedDataSize,
               void *compressedBuffer);
```

where

handle is a handle to the compressor

numberOfFrames
is the number of frames to compress: generally 1 for video data, or either CL_CONTINUOUS_BLOCK or CL_CONTINUOUS_NONBLOCK to continue compression until either the framebuffer is marked as done or **clCloseCompressor()** is called. With CL_CONTINUOUS_NONBLOCK, the call to **clCompress()** returns immediately while the compression occurs in a separate thread; CL_CONTINUOUS_BLOCK blocks until compression is completed.

frameBuffer is a pointer to the location of the buffer that contains the data that is to be compressed. Using a NULL argument here invokes the buffered interface that is described in "Using the Buffering Interface" on page 23. An error is reported if no buffer exists. Some compressors allow a value of CL_EXTERNAL_DEVICE, indicating a direct connection to an external video source.

compressedDataSize
is a pointer to the returned size of the compressed data in bytes.

compressedBuffer
is a pointer to the location where the compressed data is to be written. Using a NULL argument here invokes the buffered interface that is described in "Using the Buffering Interface" on page 23.

Call **clCompress()** once to compress *numberOfFrames* sequential frames. **clCompress()** reads the raw data from the location pointed to by *frameBuffer* and writes the compressed data to the location pointed to by *compressedBuffer*. **clCompress()** returns either the number of frames successfully compressed, or in the case of CL_CONTINUOUS_NONBLOCK, returns SUCCESS immediately.

The size of the compressed data is stored in *compressedDataSize*, even if this size exceeds the COMPRESSED_BUFFER_SIZE state parameter. If COMPRESSED_BUFFER_SIZE is less than the actual size returned by **clCompress()**, then the data returned in *compressedBuffer* is not complete.

An application-allocated compressed buffer must be at least COMPRESSED_BUFFER_SIZE bytes. This parameter should be determined by calling **clGetParams()** after the framebuffer dimensions are defined by **clSetParams()**. It is not required to set the COMPRESSED_BUFFER_SIZE, because the default is the largest possible compressed data size, which is computed from the given parameters.

Note: Parameters are explained in detail in Chapter 6, “Using Compression Library Parameters.”

Closing a Compressor

To close a compressor, call **clCloseCompressor()** with the handle of the compressor you wish to close. This frees resources associated with the compressor.

The code fragment in Example 2-4 demonstrates how to compress a series of frames using the CL_MVC1 algorithm. A compressor is opened, and then a compression loop is entered, where frames are accessed one at a time and compressed using the selected algorithm, then written to a data buffer. The compressor is closed when all of the frames have been compressed.

Example 2-4 Compressing a Series of Frames

```
#include <dmedia/cl.h>

int pbuf[][2] = {
    CL_IMAGE_WIDTH,  0,
    CL_IMAGE_HEIGHT, 0,
    CL_COMPRESSED_BUFFER_SIZE, 0
};
 ...
/* Compress a series of frames */
clOpenCompressor(CL_MVC1, &handle);

/* set parameters */
pbuf[0][1] = 320;
pbuf[1][1] = 240;
clSetParams(handle, (int *)pbuf, 4);
/* allocate the required size buffer */
clGetParams(handle, (int *)pbuf, 6);
compressedBuffer = malloc(pbuf[2][1]);

for(i = 0; i < numberOfFrames; i++)
{
    /* Get a frame from somewhere */
    ...
    clCompress(handle, 1, frameBuffer, &compressedBufferSize,
        compressedBuffer);
    /* Write the compressed data to somewhere else. */
    ...
}
clCloseCompressor(handle);
```

Decompressing a Sequence of Frames

Decompressing sequential data and video streams requires the use of a *decompressor*. A decompressor is an abstraction that modularizes decompression operations.

To decompress a sequence of frames, follow these steps:

1. Query the stream header to get the compression scheme used.
2. Open a decompressor to establish the beginning of a sequence of decompression calls.
3. Decompress frames one at a time, storing the decompressed data after each frame has been decompressed.
4. Close the decompressor to deallocate the resources associated with that decompressor.

Each of these steps is discussed in detail in the following sections.

Getting Stream Information

To determine which *scheme* to pass to the decompressor, use **clQueryScheme()** to get the *scheme* from the 16 bytes of the stream header (see Table 2-4 for a list of typical header contents, and Table 2-5 for a list of additional video stream header contents). **clQueryScheme()** returns the scheme, or the (negative) error code when an error occurs.

Once you determine the scheme, you can open the decompressor and read the header using **clReadHeader()**, which returns the actual size of the header, or zero if none is detected. Use **clQueryMaxHeaderSize()**, which returns the maximum size of the header, or zero if none is detected, to determine the size of the header to send to **clReadHeader()**. You should free the space used for the header buffer when you are finished with it.

clReadHeader() is generally called before **clCreateBuf()** to help calculate the compressed buffer size. It uses the data passed to it without affecting the buffering. **clReadHeader()** also sets up any state parameters that can be determined from the header.

The function prototypes are

```
int clQueryScheme(void *header)
int clQueryMaxHeaderSize(int scheme)
int clReadHeader(CLhandle handle, int headerSize,void *header)
```

where

header is a pointer to a buffer containing at least 16 bytes of the header.

scheme is the decompression scheme to use.

handle is a handle to the decompressor.

headerSize is the maximum size of the header in bytes.

header is a pointer to a buffer containing the header.

A typical header begins with a start code and a size, followed by parameter-value pairs such as those listed in Table 2-4.

Table 2-4 Typical Stream Header Contents

Parameter	Information supplied
CL_ALGORITHM_ID	Algorithm scheme
CL_ALGORITHM_VERSION	Version of the algorithm
CL_INTERNAL_FORMAT	Format of images immediately before compression
CL_NUMBER_OF_FRAMES	Number of frames in the sequence
CL_FRAME_RATE	Frame rate

Note: For complete information on algorithms used with the OCTANE Compression option, see Chapter 7, “Using Compression Library Algorithms.” For information on parameters, see Chapter 6, “Using Compression Library Parameters.”

In addition, video algorithms usually supply the width and height parameters listed in the header, as shown in Table 2-5.

Table 2-5 Additional Video Stream Header Contents

Parameter	Information Supplied
CL_IMAGE_WIDTH	Width
CL_IMAGE_HEIGHT	Height

The code fragment in Example 2-5 demonstrates how to query a stream header and read its contents.

Example 2-5 Getting the Decompression Scheme From a Header

```
#include <cl.h>
...
int decompressionScheme;
...
/*
 * Determine the scheme from the first 16 bytes of the
 *  header(from the beginning of video data)
*/
header = malloc(16);
read(inFile, header, 16);
decompressionScheme = clQueryScheme(header);
if(decompressionScheme < 0) {
   fprintf(stderr, “Unknown compression scheme in stream
            header.0);
   exit(0);
}
free(header);

clOpenDecompressor(decompressionScheme, &decompressorHdl);

/* Find out how big the header can be. */
headerSize = clQueryMaxHeaderSize(decompressionScheme);
if(headerSize > 0) {
   /* Read the header from the beginning of video data */
   header = malloc(headerSize);
   lseek(inFile, 0, SEEK_SET);
read(inFile, header, headerSize);
}
```

Opening a Decompressor

Call **clOpenDecompressor()**, with the desired compression scheme and a pointer for returning a handle, to open a decompressor for a given algorithm. Its function prototype is

```
int clOpenDecompressor(int scheme, CLhandle *handlePtr)
```

where

scheme is the decompression scheme to use

handlePtr is a pointer to the returned handle of the decompressor that is used by subsequent calls to identify the decompressor.

More than one decompressor can be open at a time. Use the handle that is returned in *handle* to identify a specific decompressor.

Decompressing Frames

After a decompressor has been opened, call **clDecompress()** to decompress the data. Pass to **clDecompress()** the handle returned by **clOpenDecompressor()**, the number of frames to be decompressed, the size of the data, and pointers to reference the decompressed data and the framebuffer that contains the compressed frames.

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

```
int clDecompress (CLhandle handle, int numberOfFrames,
                  int compressedDataSize, void *compressedData
                  void *frameBuffer);
```

where

handle is a handle to the decompressor.

numberOfFrames
: is the number of frames to decompress: generally 1 for video data, or either CL_CONTINUOUS_BLOCK or CL_CONTINUOUS_NONBLOCK to continue decompression until either the framebuffer is marked as done or **clCloseDecompressor()** is called. With CL_CONTINUOUS_NONBLOCK, the call to **clDecompress()** returns immediately while the compression occurs in a separate thread; CL_CONTINUOUS_BLOCK blocks until compression is completed. Using a NULL argument invokes the buffered interface that is described in "Using the Buffering Interface" on page 23.

compressedDataSize
: is a pointer to the returned size of the decompressed data in bytes.

compressedData
: is a pointer to the location where the decompressed data is to be written.

frameBuffer
: is a pointer to the location of the framebuffer that contains the data that is to be decompressed. Some compressors allow a value of CL_EXTERNAL_DEVICE, indicating a direct connection to an external video source. Using a NULL argument invokes the buffered interface that is described in "Using the Buffering Interface" on page 23. An error is reported if no buffer exists.

Closing a Decompressor

To close a decompressor, call **clCloseDecompressor()** with the handle of the decompressor you wish to close.

The code fragment in Example 2-6 demonstrates how to decompress a series of 320 × 240 (32-bit) RGBX frames by using the CL_MVC1 algorithm. A decompressor is opened, then a decompression loop is entered, where frames are accessed one at a time and decompressed by using the selected algorithm, then written to a location such as the screen. The decompressor is closed when all of the frames have been compressed.

Example 2-6 Decompressing a Series of Frames

```
#include <cl.h>
...
int compressedBufferSize;
int compressedBuffer[320][240], frameBuffer[320][240];
int     width, height, k;
static int    paramBuf[][2] = {
    CL_IMAGE_WIDTH, 0,
    CL_IMAGE_HEIGHT, 0,
    CL_ORIGINAL_FORMAT, 0,
};
width = 320;
height = 240;

clOpenDecompressor(CL_MVC1, &decompressorHdl);
paramBuf[0][1] = width;
paramBuf[1][1] = height;
paramBuf[2][1] = CL_RGBX;
clSetParams(decompressorHdl, (int *)paramBuf,
             sizeof(paramBuf) / sizeof(int));

for (k = 0; k < numberOfFrames; k++)
{ /* Decompress each frame and display it */
  dataSize = GetCompressedVideo(k, frameSize, data);
  clDecompress(decompressorHdl, 1, dataSize, data,
             frameBuffer);
  lrectwrite(0, 0, width-1, height-1,
             (unsigned int *)frameBuffer);
}
/* Close Decompressor */
clCloseDecompressor(decompressorHdl);
```

Using the Buffering Interface

Table 2-6 lists the calls explained in this section.

Table 2-6 Buffering Interface Calls

Creating and Destroying Buffers	Managing Buffers
clCreateBuf()	clQueryFree()
clDestroyBuf()	clUpdateHead()
clQueryBufferHdl()	clUpdateTail()
clQueryHandle()	clQueryValid()
	clDoneUpdatingHead()
	clQuery()
	clUpdate()

The buffered interface is designed for

- VCR-like control over the video stream
- maximum efficiency by buffering compressed data and uncompressed frames
- blocking and nonblocking access
- transparent buffering for hardware acceleration or for multiprocessor operation
- multi-threaded applications

This interface includes the calls of the sequential interface, plus buffer-management routines to access the compressed data and the uncompressed framebuffers.

The buffer management routines allow blocking and nonblocking access and accumulation of compressed data and decompressed frames. The compression or decompression modules can each be placed in separate processes. Separating the processes allows the compression or decompression process to get ahead a few frames, which is advantageous for algorithms such as MPEG, which compress the data using techniques that take advantage of similarities between frames, and it also facilitates hardware acceleration.

Buffers manage compression and decompression for data that is accessed randomly, or when it is necessary to separate the task into several processes or across multiple processors. Buffering allows the accumulation of compressed data to be independent of that of decompressed frames. The buffering interface can be used for multi-threaded applications.

Buffers are implemented as ring buffers in *libcl*. A ring buffer contains a number of blocks of arbitrary size. It maintains a pointer to the buffer location, a size, and pointers to the head of newest and tail of oldest valid data. Separate processes can be producing (adding to the buffer) and consuming (removing from the buffer).

Figure 2-1 is a conceptual drawing of a ring buffer.

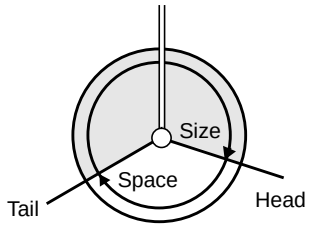

Figure 2-1 Ring Buffer

The circle represents the ring buffer. The shaded part of the circle contains frames or data, depending on the buffer type; the blank part is free space. The size of the data (or the number of frames) available and the size of the space (or the number of frames of space) are shown by the arrows within the circles. Head marks the location where new data or frames, depending on the buffer type, are inserted. Tail marks the location where the oldest data or frames, depending on the buffer type, are removed. The head and tail march around the circle as data or frames, depending on the buffer type, are produced and consumed. The double vertical bar at the top signifies the discontinuity between the end of the buffer and the beginning of the buffer in linear physical memory.

Note: The OCTANE Compression hardware is optimized for use in the asynchronous ring-buffer modes of operation of the CL. Both the compressed and uncompressed channels to main memory are buffered; flushing those buffers slows processing. Applications that require real-time operation must use the ring-buffer modes of the CL.

DMA operations on OCTANE Compression vary depending upon the number of bytes available to transfer. The device driver attempts to transfer data in large blocks, but can step it down to less efficient, smaller block sizes to transfer data out of a ring buffer completely. The minimum transfer size supported by the OCTANE Compression option is eight bytes.

Creating a Buffer

The buffer management routines allow buffer space to be allocated by the library (internal) or by the application (external). A buffer often already exists in memory where the frames exist (on compression) or need to be placed (on decompression). External buffering allows this to happen without having to copy the data to or from an internal buffer. An external buffer is managed entirely within *libcl* as a ring buffer.

Use **clCreateBuf()** to create an internal or external buffer. Use **clDestroyBuf()** to destroy an internal or external buffer. If **clDecompress()** or **clCompress()** is called with NULL for the compressed data or framebuffer parameters, then the buffer specified by **clCreateBuf()** is used. An error is reported if no buffer was created.

The function prototypes are

```
CLbufferHdl * clCreateBuf (CLhandle handle, int bufferType,
                int blocks, int blockSize, void **bufferPtr)
int          clDestroyBuf (CLbufferHdl bufferHdl)
```

where

handle is the handle to the compressor or decompressor.

bufferType specifies the type of the ring buffer, which can be either

- CL_FRAME for a framebuffer
- CL_DATA for a data buffer

blocks specifies the number of blocks in the buffer.

blockSize specifies the size in bytes of the block. This value is either 1 for data buffering or a multiple of the frame size for frame buffering.

bufferPtr is a pointer to a pointer to the ring buffer. If it points to a NULL pointer, it specifies an internally allocated buffer, and the value it points to receives the buffer pointer.

bufferHdl is a handle to the buffer.

The handle returned in *bufferHdl* is used in subsequent buffering calls, with which you can get the buffer handle or the compressor or decompressor handle.

Use **clQueryBufferHdl()** to get the buffer handle from a compressor or decompressor handle. Its function prototype is

```
CLbufferHdl clQueryBufferHdl(CLhandle handle,
                             int bufferType, void **bufferPtr2)
```

Use **clQueryHandle()** to get the compressor or decompressor handle from a buffer handle. Its function prototype is

```
CLhandle clQueryHandle(CLbufferHdl bufferHdl)
```

The code fragment in Example 2-7 demonstrates how to create and use an internal buffer.

Example 2-7 Creating and Using an Internal Buffer

```
#include <cl.h>
CLhandle       handle;
CLbufferHdl    bufferHdl;
void    *buffer;
 ...
clOpenCompressor(CL_MVC1, &handle);

/* Create a buffer of 10 blocks of size 10000 */
buffer = NULL;
bufferHdl = clCreateBuf(handle, CL_DATA, 10, 10000, &buffer);
bufferHdl = clQueryBufferHdl(handle, CL_DATA, &buffer);
handle = clQueryHandle(bufferHdl);
 ...
clDestroyBuf(bufferHdl);
clCloseCompressor(handle);
```

The code fragment in Example 2-8 demonstrates how to create and use an external buffer.

Example 2-8 Creating and Using an External Buffer

```
#include <cl.h>
CLhandle       handle;
CLbufferHdl    bufferHdl;
void    *buffer;
clOpenCompressor(CL_MVC1, &handle);

/* Create a buffer of 10 blocks of size 10000 */
buffer = malloc(10*10000);
bufferHdl = clCreateBuf(handle, CL_DATA, 10, 10000, &buffer);
bufferHdl = clQueryBufferHdl(handle, CL_DATA, &buffer);
handle = clQueryHandle(bufferHdl);
 ...
clDestroyBuf(bufferHdl);
clCloseCompressor(handle);
```

Managing Buffers

The buffer management routines are used for both uncompressed (or decompressed) frames and compressed data. When used for compressed data, they return the number of blocks (of selectable byte size) of valid contiguous data (or free space for data). When used for frames, they return the actual number of valid contiguous frames (or free space for frames).

Use **clQueryFree()** to find out how much free space is available and where it is located.

Use **clUpdateHead()** to notify the library that data has been placed in the ring buffer and to update the head pointer.

Use **clQueryValid()** to find out how many blocks of valid data are available and where they are located.

Use **clUpdateTail()** to notify the library that valid data has been consumed from the ring buffer and that data is no longer needed.

Use **clDoneUpdatingHead()** to notify a decompressor that no more data will be arriving, in which case **clDecompress()** returns when the buffer empties.

The function prototypes are

```
int clQueryFree (CLbufferHdl bufferHdl, int space,
                 void **freeData, int *wrap)

int clUpdateHead (CLbufferHdl bufferHdl, int amountToAdd)

int clQueryValid (CLbufferHdl bufferHdl, int amount,
                  void **ValidData, int *wrap)

int clUpdateTail (CLbufferHdl bufferHdl, int amountToRelease)

int clDoneUpdatingHead (CLbufferHdl bufferHdl)
```

where

bufferHdl is a handle to a compressor buffer.

space is the number of blocks of free space in the framebuffer to wait for. If it is zero, then the current number of blocks of space is returned without waiting.

freeData is a pointer to the returned pointer to the location where data or frames can be placed.

wrap is the number of blocks that have wrapped to the beginning of the ring buffer (see Figure 2-2 and the accompanying discussion). If it is greater than zero, then the end of the ring buffer has been reached and the routine return value will not increase (on subsequent calls) until either **clUpdateHead()** for free space or **clUpdateTail()** for valid data has been called.

amountToAdd is the number of blocks of free space that were written by the caller and are ready to be consumed by the library.

amount is the number of blocks of valid data in the data buffer to wait for. If it is zero, then the number of blocks currently available is returned without waiting.

validData is a pointer to the returned pointer to the location where valid data can be retrieved.

amountToRelease is the number of blocks of valid data that were consumed by the call and can be reused by the library.

Each compressor or decompressor can have a (compressed) data buffer and a (uncompressed) framebuffer.

The block size for the uncompressed framebuffer must be a multiple of the size of one frame. This value, multiplied by the number of blocks specified, determines how many frames ahead a decompressor can get if you allow it to work ahead.

Producing and Consuming Data in Buffers

Figure 2-2 shows snapshots of the buffer state over time as a sequence of produce and consume processes operate on the buffer. Initially, the buffer is empty and both head and tail point to the beginning of the buffer. When head and tail are equal, the buffer is either empty or full—in this case, the buffer is empty. The library keeps track internally of whether the buffer is empty or full.

In the first frame of Figure 2-2, a process begins producing—adding data to the buffer. First, a call is made to **clQueryFree()** to determine how much free space is available. An amount equal to the entire buffer size is returned. Data is written to the buffer, then the location of head is updated to point to the beginning of the next available free space.

In the second frame of Figure 2-2, the next call to **clQueryFree()** returns the free space that exists from head to tail. More data is written and the head is updated once again.

In the third frame of Figure 2-2, a process begins consuming—taking data from the buffer. A call is made to **clQueryValid()** to determine the amount of valid data in existence. The size of the data that was written by the producers so far is returned. Data is read from the beginning of the buffer to the desired location, and tail is updated to point to the next location containing valid data.

The final frame of Figure 2-2 shows what happens when the free space is not contiguous. When the next producer queries for the available free space, two pieces of free space exist—one on each side of the buffer discontinuity. The first piece of free space, which is from head to the end of the buffer, is returned as usual. The second piece of free space, which is from the beginning of the buffer to tail, is returned in the wrap argument. You can't write data across the buffer boundary, so it must be written to the buffer in two steps. First write the data until the end of the buffer is reached, then write the data from the beginning of the buffer until all of the data has been used. Head can then be updated to point to the next available free space.

The process for reading data across the frame discontinuity is analogous.

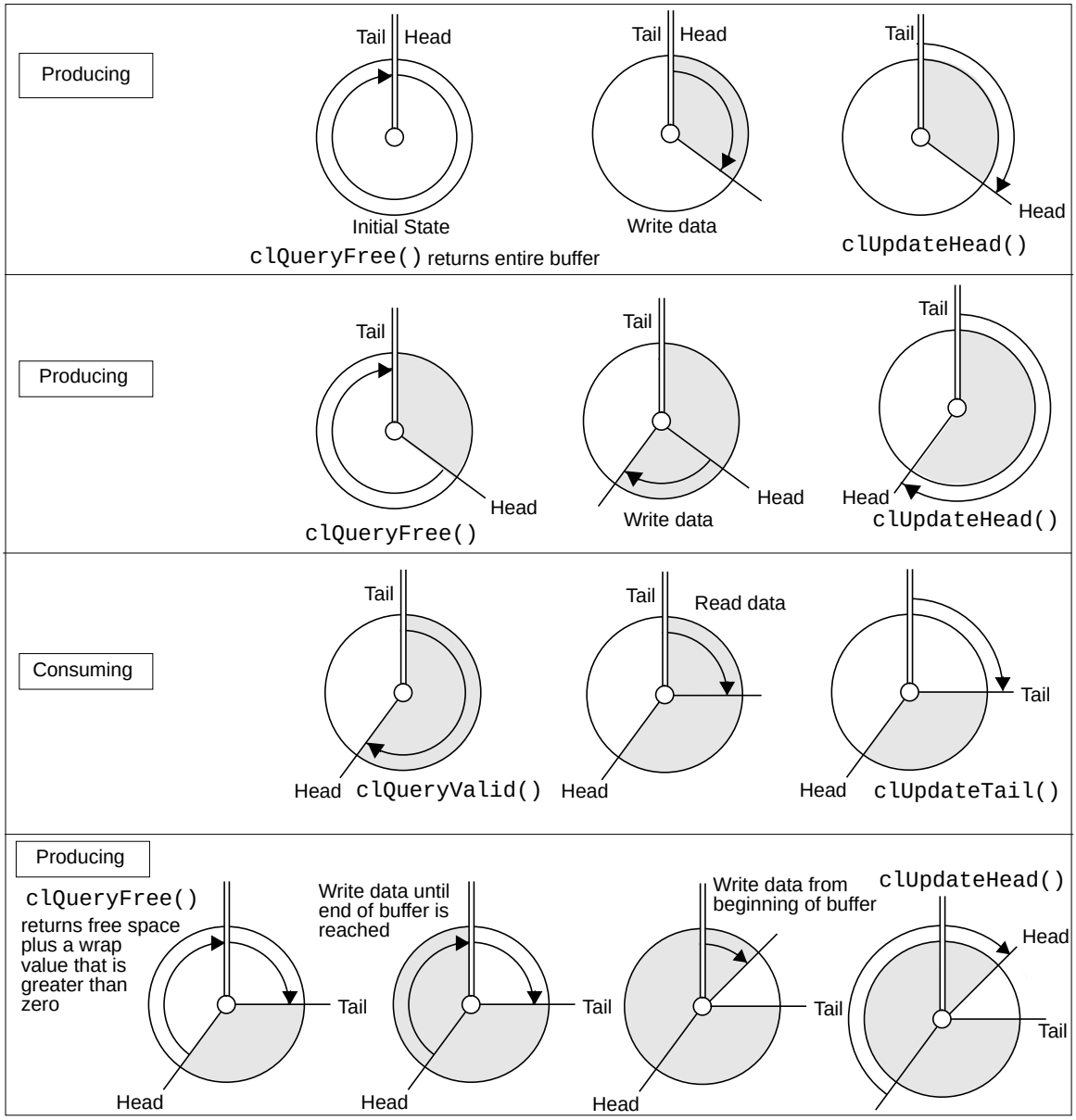

Figure 2-2 Snapshots of Buffer State During Producing and Consuming Processes

Figure 2-3 shows the architecture of the buffer management. Rectangles represent code modules that can be placed in separate synchronized processes. The buffer management routines are shown within the boxes. Arrows show the flow of data from the modules to and from the buffers.

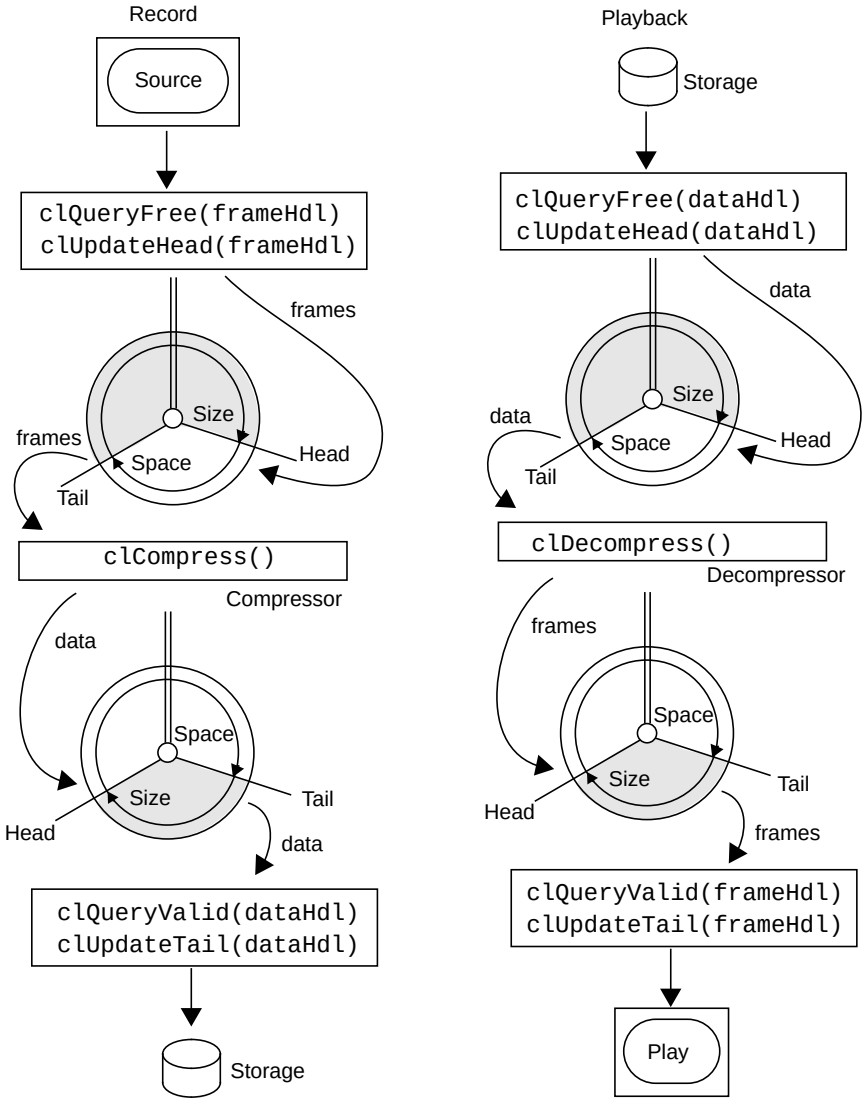

Figure 2-3 Flow of Data in a Buffered Compression and Decompression Scheme

Hardware Buffer Flushing and Latency

When an image is compressed or decompressed in memory-to-memory modes, it can be partially contained in the hardware for some time. The device driver does not notify the application that data is available until all data for a particular field has made its way to memory and a complete processor cache line has been completed.

This situation is of concern only for applications that use the asynchronous ring-buffer mode of operation, and do not wish to close the compressor or decompressor (codec). When the codec is closed, any buffered data is processed and flushed to the application.

A portion of the image is usually trapped in hardware buffering until either the compressor is closed or a subsequent image flushes this image portion out. If the application is decompressing to a CL_EXTERNAL_DEVICE, the application should ensure that the last compressed image sent to the board is completely decompressed. When the application wants to explicitly flush an image out to the video portion of the board, it should send 16 bytes of the value 255 ($0 \times FF$), as shown in Example 2-9.

Example 2-9 Flush Compressed Data to CL_EXTERNAL_DEVICE

```
...
/* get an image from somewhere and put into ring buffer */
/* send an image of data to the decompressor */
clUpdateHead( bufferHandle, size_of_image );
/* flush data though JPEG decompressor */
avail = clQueryFree( bufferHandle, 16, &free, &wrap );
if (prewrap > 16) {
    memset( free, 0xFF, 16 );
    clUpdateHead( bufferHandle, 16 );
}
else {
    /* handle wrapped CL buffers */
}
```

Note: This flush operation is necessary only when the application expects some time between the images that are decompressed. If the application is immediately sending another compressed image, that image flushes the previous image through the decompressor.

Creating a Buffered Record and Play Application

This section provides several examples of how to use buffering. Blocking and nonblocking playback and record examples are provided.

Creating a Basic Buffered Playback Application

The code fragment in Example 2-10 demonstrates how to use buffers for a playback application. The amount of space is queried, the data is read directly into the data buffer, and the decompressor is notified of the change. The data can then be decompressed and retrieved by querying the number of frames, displaying them directly from the framebuffer, then releasing the consumed frames.

Example 2-10 Using Buffers for Playback

```
#include <cl.h>
 ...
actualLen = clQueryFree(decompressorHdl, len, &buf, &wrap);
read(fd, buf, actualLen);
len = clUpdateHead(dataHdl, actualLen);

clDecompress(decompressorHdl, 1, 0, NULL, NULL);

actualNumberOfFrames = clQueryValid(frameHdl, numberOfFrames,
  &frameBuffer, &wrap);
ConsumeFrames(actualNumberOfFrames, frameBuffer);
numberOfFrames = clUpdateTail(bufferHdl, actualNumberOfFrames);
```

clUpdateHead() indicates to the library that the data has been placed in the data buffer, but does not copy the data.

clDecompress() reads compressed data from the data buffer and writes uncompressed frames to the framebuffer. If space for a frame exists in the framebuffer, then the routine begins decompressing directly to the framebuffer. It consumes data from the data buffer until there is no more data, then it sleeps for a while and periodically continues to check for data until there is enough. When it finishes decompressing a frame, it updates the framebuffer pointers and returns. **clDecompress()** does not return until decompression is complete or until an error occurs.

If no more data is added to the buffer, the application can call **clDoneUpdatingHead()** so that the library does not stall.

clQueryValid() returns the pointer into the frame ring buffer. **clUpdateTail()** is required to free the internal framebuffer space, which you don't want to happen until after you consume it. The pointer to the next valid frame is kept internally, and only the actual number of framebuffers that have been decompressed are returned.

The *size* (or *numberOfFrames*) returned by the routines are for the contiguous data (or frames, depending on the buffer type). The *wrap* argument of the **clQuery()** routines returns the *actualLen* (or *numberOfFrames*) that have wrapped to the beginning of the buffer.

The frame accesses does not cross the buffer boundary, and the *wrap* argument does not need to be used if both

- the allocated size of the frame ring buffer is a multiple of the size of a frame times the *numberOfFrames* that will be requested, and
- the same number of frames will always be requested

If the *len* (or *numberOfFrames*) passed to the **clQuery()** routines is greater than zero, the routine blocks until that much data (or that many frames) is available. If it is less than or equal to zero, then the routine returns immediately with whatever data is available. In either case, the buffer pointers are not adjusted until the **clUpdate()** routines are called.

Creating a Nonblocking Buffered Playback Application

The code fragment in Example 2-11 demonstrates how to implement nonblocking playback.

Example 2-11 Using Buffers for Nonblocking Playback

```
actualLen = clQueryFree(decompressorHdl, 0, &buf, &wrap);
if((actualLen > MIN_READ_SIZE) || (wrap > 0)) {
   read(fd, buf, actualLen);
   len = clUpdateHead(decompressorHdl, actualLen);
}
/* Go do something else */
 ...
```

Each call to **clQueryFree()** returns the same *buf* pointer but increasing values of *actualLen* until MIN_READ_SIZE is reached, whereupon **clUpdateHead(dataHdl)** updates the pointers, and the next call to **clQueryFree()** returns a different *buf* pointer and a reset *actualLen*. If *wrap* becomes greater than zero, the end of the buffer has been reached and *actualLen* does not get any larger, so the amount remaining in the buffer must be consumed.

Creating a Buffered Record Application

The code fragment in Example 2-12 demonstrates how to use buffers for recording.

Example 2-12 Using Buffers for Recording

```
actualNumberOfFrames = clQueryFree(bufferHdl, numberOfFrames,
                                   &frameBuffer, &wrap);
ProduceFrames(actualNumberOfFrames, frameBuffer);
numberOfFrames = clUpdateHead(bufferHdl, actualNumberOfFrames);

clCompress(compressorHdl, 1, NULL, 0, NULL);

actualBufSize = clQueryValid(compressorHdl, bufSize, &buf,
                             &wrap);
write(fd, buf, actualBufSize);
bufSize = clUpdateTail(compressorHdl, actualBufSize);
```

The amount of free space is queried, the frames are read directly into the framebuffer, and the compressor is notified of the change. The frames can then be compressed and the data can be retrieved by querying the amount of the data, consuming directly from the data buffer, then releasing the consumed data.

clUpdateHead() indicates that the frames have been placed in the framebuffer, but does not copy the data.

clCompress() reads from the framebuffer and writes to the data buffer. If a frame exists in the framebuffer, then the routine begins compressing directly from the framebuffer. It places compressed data in the data buffer until there is no more room, then it blocks until there is enough room. When it completes compression of a frame, it updates the framebuffer pointers and returns. **clCompress()** does not return until compression is complete (or an error occurs).

clQueryValid() returns the pointer into the data ring buffer. **clUpdateTail()** is required to free the internal data buffer space, which you don't want to happen until after you consume it—in this case, by writing it. The pointer to valid data is kept internally, and **clUpdateTail()** returns only the actual number of bytes released.

The *amount / numberOfFrames* returned by the routines are for contiguous data or frames. The *wrap* parameter of the **clQuery()** routines returns the *amount/numberOfFrames* that have wrapped to the beginning of the buffer.

If the allocated size of the frame ring buffer is a multiple of the size of a frame times the *numberOfFrames* that will be requested, assuming that the same number of frames is always requested, then the frame accesses will not cross the buffer boundary, and the *wrap* parameter does not need to be used.

If the *amount* passed to the **clQuery()** routines is greater than zero, then the routine blocks until that much data is available. If it is less than or equal to zero, then the routine returns immediately with whatever data is available. In either case, the buffer pointers are not adjusted until the **clUpdate()** routine is called.

Creating a Nonblocking Buffered Record Application

The code fragment in Example 2-13 demonstrates how to use buffers for nonblocking recording.

Example 2-13 Using Buffers for Nonblocking Recording

```
actualLen = clQueryValid(dataHdl, 0, &buf, &wrap);
if((actualLen > MIN_READ_SIZE) || (wrap > 0)){
   write(fd, buf, actualLen);
   len = clUpdateTail(dataHdl, actualLen);
}
```

Each call to **clQueryValid()** returns the same *buf* pointer but increasing values of *actualLen* until MIN_READ_SIZE is reached, whereupon **clUpdateTail()** updates the pointers, and the next call to **clQueryValid()** returns a different *buf* pointer and a reset *actualLen*. If *wrap* becomes greater than zero, then the end of the buffer has been reached, and *actualLen* does not get any larger, so the amount remaining in the buffer must be consumed.

Note that the consuming, compressing or decompressing, and producing have been separated into different sets of calls. The most powerful use of the interface is to separate these functional groupings into shared processes using **sproc()**, or to allocate them to separate (shared data) processors. See sproc(2) for more information about using **sproc().**

The buffers are set up by **clCreateBuf()**. To use data input buffering, **clDecompress()** receives NULL for *compressedData*. To use frame output buffering, **clDecompress()** receives NULL for *frameBuffer*.

clCompress() reads from the framebuffer and writes to the data buffer. If a frame exists in the framebuffer, then the routine begins compressing directly from the framebuffer. It places compressed data in the data buffer until there is no more room, then it sleeps for a while and checks again until there is enough room. When it finishes compressing a frame, it updates the framebuffer pointers and returns. **clCompress()** does not return until compression is complete or until an error occurs.

Creating Buffered Multiprocess Record and Play Applications

Consuming, compressing or decompressing, and producing can be separated into different sets of calls. The most powerful use of the buffering interface, however, is to separate these functional groups into shared processes using **sproc()** or to allocate them to separate (shared data) processors.

The code fragment in Example 2-14 demonstrates how to implement multiprocess playback. The functions in boldface can be implemented as separate processes.

Example 2-14 Using Buffers for Multiprocess Playback

```
ProduceDataProcess()
  actualLen = clQueryFree(dataHdl, len, &buf, &wrap);
  read(fd, buf, actualLen);
  len = clUpdateHead(dataHdl, actualLen);

DecompressProcess()
  clDecompress(decompressorHdl, 1, 0, NULL, NULL);

ConsumeFrameProcess()
  actualNumberOfFrames = clQueryValid(frameHdl,
    numberOfFrames, &frameBuffer, &wrap);
  lrectwrite(0, 0, width - 1, height - 1, frameBuffer);
  numberOfFrames = clUpdateTail(frameHdl,actualNumberOfFrames);
```

The code fragment in Example 2-15 demonstrates how to use buffers for multiprocess recording. The functions in boldface can be implemented as separate processes.

Example 2-15 Using Buffers for Multiprocess Recording

```
ProduceFrameProcess()
  actualNumberOfFrames = clQueryFree(frameHdl,
    numberOfFrames, &frameBuffer, &wrap);
  lrectread(0, 0, width - 1, height - 1, frameBuffer);
  numberOfFrames = clUpdateHead(frameHdl,
    actualNumberOfFrames);

CompressProcess()
  clCompress(compressorHdl, 1, NULL, &compressedDataSize,
      NULL);

ConsumeDataProcess()
  actualBufSize = clQueryValid(dataHdl, bufSize,&buf, &wrap);
  write(fd, buf, actualBufSize);
  bufSize = clUpdateTail(dataHdl, actualBufSize);
```

Implementing functions as separate processes allows the application nonblocking access to compression and decompression. The application will almost always use **ProduceDataProcess()** for playback and the **ProduceFrameProcess()** for record, since the single process blocks forever within **clDecompress()**/**clCompress()** if insufficient data or frames, depending on the buffer type, are supplied. The other processes can be made parts of the **main()** process. These processes could also be spread across multiple processors.

Chapter 3

Programming With the Video Library

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

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

- "Video Library Capabilities"
- "The VL Programming Model"
- "Performing Preliminary Steps"
- "Opening a Connection to the Video Daemon"
- "Specifying Nodes on the Data Path"
- "Creating and Setting Up the Data Path"
- "Setting Parameters for Data Transfer to or From Memory or Codec Nodes"
- "Transferring Video Data to and From Devices"
- "Ending Data Transfer"
- "Example Programs"

Video Library Capabilities

The Video Library provides a software interface to the OCTANE Compression board, enabling applications to

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

The Video Library (VL) is a collection of device-independent and device-dependent C language calls for Silicon Graphics workstations equipped with video options. The VL provides generic video tools, including simple tools for importing and exporting digital data to and from Silicon Graphics systems, as well as to and from third-party video devices that adhere to the Silicon Graphics architectural model for video devices. Video tools are described in the *Media Control Panels User's Guide*, which you can view using the IRIS InSight™ viewer; similar applications are supplied in source-code form as examples in the directories */usr/share/src/dmedia/video/vl* and */usr/share/src/dmedia/video/vl/OpenGL*).

The VL works with other Silicon Graphics libraries, such as OpenGL®. The VL does not depend on the X Window System™, but you can use X Window System libraries or toolkits to create a windowing interface.

The VL allows programs to get events 60 times per second on a quiescent system; it also enables programs to share resources or to gain exclusive use of resources. It supports input and output of video data to or from locked-down memory at the nominal frame rate. The VL provides an API that enables applications to capture or play back video from system memory.

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

This section explains

- VL system software architecture
- VL architectural model of video devices
- OCTANE Digital Video formats

VL System Software Architecture

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

- video daemon
- generic video tools
- library and header files

Figure 3-1 diagrams the interaction between the VL, the video daemon, the kernel, the hardware, and the X Window System server.

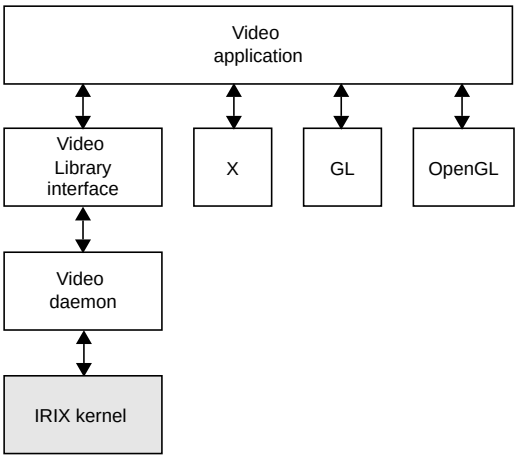

Figure 3-1 VL System Components

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

Besides these components, the VL includes a collection of applications that support device configuration and control setting and retrieval, generic tools that display video on a workstation, and video control panels.

Video Daemon

The video daemon */usr/etc/videod*, which has device-dependent and device-independent portions, handles video device management and status information.

Management that the video daemon performs includes

- multiple client access to multiple devices

  The library supports connections from multiple client applications and manages their access to a limited number of video devices.

- dispatching events

  As events are handled and noted by devices, the daemon notifies applications that have expressed interest in those events.

- handling events

  As events are generated by the various devices, the daemon initiates any action required by an event before it hands the event off to interested applications.

- maintaining exclusive use

  Types of data or control usage for video clients in a Video Library application are *Done Using*, *Read-only*, *Lock*, and *Shared*. These usage levels apply only to write access on controls, not read access. Any application can open and read the control's values at any time.

- client cleanup on exit

  When a client exits or is terminated abnormally, its connection to the daemon is broken; the daemon performs any cleanup required of the system. Any exclusive-use modes that have been set are cleared; interested clients are notified that the device is no longer in exclusive use. Controls set by the client might persist, but are not guaranteed to remain after the client closes the connection.

Status information for which the video daemon is responsible includes

- system status of video devices

  The video devices installed in a system can be queried as to availability and control status.

- video positioning (offset) information

- control setting and retrieval

  Device-independent and device-dependent controls are set and retrieved through the video daemon.

Generic Video Tools

The generic video tools include

videopanel (vcp)	Use this graphical user interface to set controls, such as hue or contrast, on devices. The panel resizes itself dynamically to reflect available video devices.
vlcmd	Use the Video Library command-line interface to enter Video Library shell-level and other commands.
videoin	Use the video input window tool to view input video in a window.
videoout	Use the video output tool to output video from a rectangular area of the screen on hardware that supports the screen-to-video path.
vlinfo	Use the video info tool to display information about video devices available through the VL, such as the name of the X server, number of devices on the server, and the types and ID numbers of nodes, sources, and drains on each device.
vintovout	Use this tool to display video input on the device attached to video output.
memtovid	Use this tool to output frames (images) to video out on hardware that supports the memory-to-video path.
vidtomem	Use this tool to capture a single frame (the current video input) or a specified number of frames, depending on the hardware limits for burst capture, and write the data to disk. Capture size can also be specified. The data, which can be translated or left as raw data, can be used by the *memtovid* tool.

The *vlinfo*, *vidtomem*, and *memtovid* tools are command-line tools. In addition to their reference pages, these tools have explanations in the *Media Control Panels User's Guide*. Similar applications are supplied in source-code form as examples in the directories */usr/share/src/dmedia/video/vl* and */usr/share/src/dmedia/video/vl/OpenGL*).

Library and Header Files

The client library is */usr/lib/libvl.so*. The header files for the VL are in */usr/include/dmedia*. The header file for the VL, *vl.h*, contains the main definition of the VL API and controls. The header files for OCTANE Compression are

- */usr/include/dmedia/dev_mgv.h* (linked to */usr/include/vl/vl_mgv.h*)
- */usr/include/dmedia/dev_impact.h* (linked to */usr/include/vl/vl_impact.h*)
- */usr/include/dmedia/dev_mgc.h* (linked to */usr/include/vl/vl_mgc.h*), which is the header file for OCTANE Compression
- */usr/include/dmedia/vl_impact.h* (linked to */usr/include/vl/dev_impact.h*), which contains definitions common to the OCTANE Digital Video and OCTANE Compression devices

VL Architectural Model of Video Devices

The VL recognizes these classes of objects:

- *devices*, each including sets of nodes

  A video device can be internal, such as the OCTANE Digital Video board, or external, such as a videotape recorder connected to the OCTANE Digital Video board.
- *nodes*: sources, drains, and internal nodes
- *paths*, connecting sources and drains
- *ports*, the entities on nodes that produce or consume video data
- *controls*, or parameters, that modify how data flows through nodes; for example:
  - video device parameters, such as blanking width, gamma value, horizontal phase, sync source
  - video data capture parameters
  - blending parameters
- *buffers*, for sending frame data to and receiving frame data from host memory; the VL buffers contain a number of blocks; each with a pointer, a size, and pointers to the head (oldest) and tail (newest) valid data

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

Node

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

Path

The path is an abstraction for a way of moving data around. A path is a set of nodes with video routes (connections) between the ports on the nodes. A path defines the useful connections between video sources and video drains. Figure 3-2 shows a simple path in which a frame from a videotape is displayed in a workstation window.

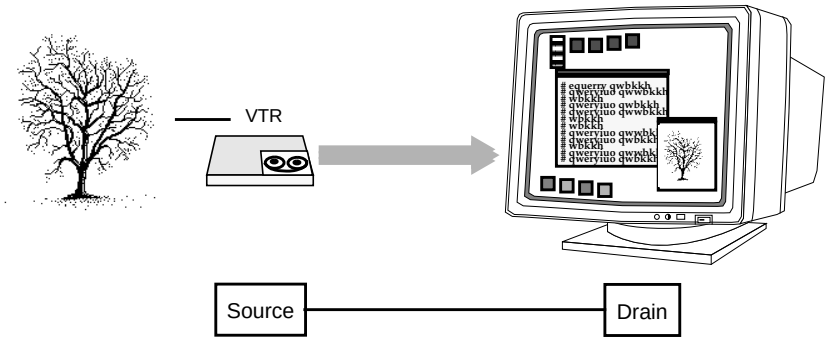

Figure 3-2 Simple VL Path

Figure 3-3 shows a more complex path with two video sources: a frame from a videotape and a computer-generated image are blended and output to a workstation window. This path is set up in stages.

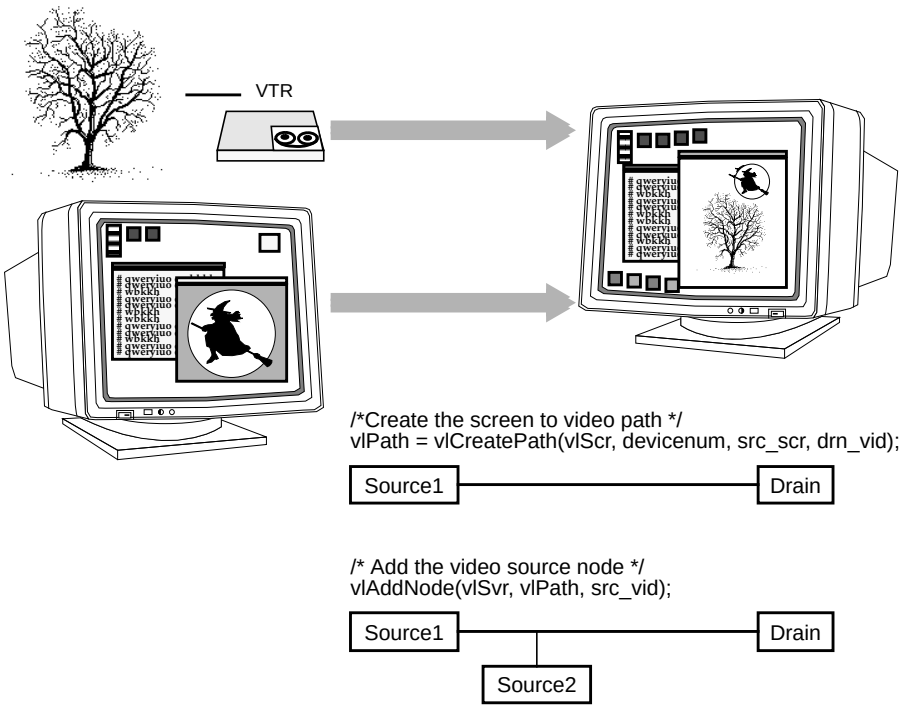

Figure 3-3 Simple VL Blending

Port

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

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

Ports have several attributes:

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

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

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

Ports have generic names; for example:

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

For the symbolic names for ports, see */usr/include/dmedia/dev_impact.h*. Appendix A, "Video Library Controls and Compression Library Parameters for the OCTANE Compression Option," gives the ports associated with each node.

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

Data flows from a source port to a drain port. It is not permissible to connect a source port to another source port, or a drain port to another drain port.

Connections obey stream-usage levels set with **vlSetupPaths()**. Usage is drain-centric: the usage levels of the path(s) using the drain node serve as the usage level of the connection.

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

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

OCTANE Compression Formats

The OCTANE Compression board translates video signals into a form usable by the Indigo^2 workstation. It also does the reverse, translating memory buffers into video signals.

Table 3-1 summarizes the formats that the OCTANE Compression board supports.

Table 3-1 Video Formats for OCTANE Compression

Format	Signal	Nodes
SMPTE YUV (VL_FORMAT_SMPTE_YUV)	Contains YUV components in the range 1-254; superblack and superwhite values can be present.	All memory nodes
RGB (VL_FORMAT_RGB)	Full-range 8-bit per component RGBA. Component range is 0 to 255 (8-bit).	All memory nodes

The VL Programming Model

Syntax elements are as follows:

- VL types and constants begin with uppercase VL; for example, VLServer
- VL functions begin with lowercase vl; for example, **vlOpenVideo()**

Data transfers fall into two categories:

- transfers involving memory (video to memory, memory to video), which require setting up a buffer
- transfers that do not involve memory (video in to video out), which do not require setting up a buffer.

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

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

Table 3-2 lists calls explained in this chapter.

Table 3-2 Video Library Calls for Data Transfer

All Transfers	Transfers Involving Memory	Setting Controls
vlOpenVideo()	vlGetTransferSize()	vlSetControl()
vlGetDevice()	vlCreateBuffer()	vlGetControl()
vlGetDeviceList()	vlRegisterBuffer()	vlControlList()
vlGetNode()	vlGetNextValid()	vlGetControlInfo()
vlCreatePath()	vlGetLatestValid()	
vlSetConnection()	vlPutValid()	
vlGetConnection()	vlGetNextFree()	
vlAddNode()	vlGetActiveRegion()	
vlRemoveNode()	vlPutFree()	
vlSetupPaths()	vlGetDMediaInfo()	
vlSelectEvents()	vlGetImageInfo()	
vlBeginTransfer()	vlDeregisterBuffer()	
vlEndTransfer()	vlDestroyBuffer()	
vlDestroyPath()		
vlCloseVideo()		

Performing Preliminary Steps

To build programs that run under VL, you must

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

The client library is */usr/lib/libvl.so*. The header files for the VL are in */usr/include/dmedia*; see “Library and Header Files” on page 46 for a list.

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

Opening a Connection to the Video Daemon

The first thing a VL application must do is open the device with **vlOpenVideo()**. Its function prototype is

```
VLServer vlOpenVideo(const char *sName)
```

where *sName* is the name of the server to which to connect; set it to a NULL string for the local server. For example:

```
vlSvr = vlOpenVideo("")
```

Specifying Nodes on the Data Path

Use **vlGetNode()** to specify nodes; this call returns the node's handle. Its function prototype is

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

where

VLNode is a handle for the node, used when setting controls or setting up paths

vlSvr names the server (as returned by **vlOpenVideo()**)

type specifies the type of node:

- VL_SRC: source
- VL_DRN: drain
- VL_DEVICE: device for device-global controls

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

kind specifies the kind of node:

- VL_CODEC: compressor/decompressor (codec node)
- VL_MEM: region of workstation memory
- VL_VIDEO: connection to a video device; for example, a video tape deck or camera

Note: Appendix A gives full details of all OCTANE Digital Video nodes.

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

To discover which node the default is, use the control VL_DEFAULT_SOURCE after getting the node handle the normal way. The default video source is maintained by the VL. For example:

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

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

Creating and Setting Up the Data Path

Once nodes are specified, use VL calls to

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

Creating the Path

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

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

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

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

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

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

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

Getting the Device ID

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

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

For example:

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

Adding a Node

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

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

where

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

This example fragment adds a source node and a blend node:

```
vlAddNode(vlSvr, vlPath, src_vid);
vlAddNode(vlSvr, vlPath, blend_node);
```

Setting Up the Data Path

Use **vlSetupPaths()** to set up the data path. Its function prototype is

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

where

vlSvr names the server to which the path is connected

paths specifies a list of paths you are setting up

count specifies the number of paths in the path list

ctrlusage specifies usage for path controls:

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

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

- VL_READ_ONLY: controls cannot be set, only read; for example, this control can be used to monitor controls
- VL_LOCK: prevents other paths from setting controls on this path; controls cannot be used by another path
- VL_DONE_USING: the resources are no longer required; the application releases this set of paths for other applications to acquire

streamusage specifies usage for the data:

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

  Note: When using VL_SHARE, pay attention to events. If another user has taken over the node, a VLStreamPreempted event occurs.

- VL_READ_ONLY: the path cannot perform transfers, but other resources are not locked; set this value to use the path for controls
- VL_LOCK: prevents other paths that share data transfer resources with this path from transferring; existing paths that share resources with this path will be preempted
- VL_DONE_USING: the resources are no longer required; the application releases this set of paths for other applications to acquire

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

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

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

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

Note: Do not combine implicit and explicit routing.

Specifying the Path-Related Events to Be Captured

Use **vlSelectEvents()** to specify the events you want to receive. Its function prototype is

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

where

vlSvr names the server to which the path is connected

path specifies the data path.

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

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

Table 3-3 VL Event Masks

Symbol	Meaning
VLStreamBusyMask	Stream is locked
VLStreamPreemptedMask	Stream was grabbed by another path
vlStreamChangedMask	Video routing on this path has been changed by another path
VLAdvanceMissedMask	Time was already reached
VLSyncLostMask	Irregular or interrupted signal
VLSequenceLostMask	Field or frame dropped
VLControlChangedMask	A control has changed
VLControlRangeChangedMask	A control range has changed
VLControlPreemptedMask	Control of a node has been preempted, typically by another user setting VL_LOCK on a path that was previously set with VL_SHARE
VLControlAvailableMask	Access is now available
VLTransferCompleteMask	Transfer of field or frame complete
VLTransferFailedMask	Error; transfer terminated; perform cleanup at this point, including **vlEndTransfer()**

Table 3-3 (continued) VL Event Masks

Symbol	Meaning
VLEvenVerticalRetraceMask	Vertical retrace event, even field
VLOddVerticalRetraceMask	Vertical retrace event, odd field
VLFrameVerticalRetraceMask	Frame vertical retrace event
VLDeviceEventMask	Device-specific event, such as a trigger
VLDefaultSourceMask	Default source changed

For example:

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

Event masks can be Or'ed; for example:

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

Setting Parameters for Data Transfer to or From Memory or Codec Nodes

Transferring data to or from memory requires creating a VL buffer; its size is determined by the size of the frame data you are transferring.

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

Setting Node Controls for Data Transfer

Important data transfer controls for source and drain nodes are summarized in Table 3-4. They should be set in the order in which they appear in the table.

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

Note: For drain nodes, VL_PACKING must be set first. Note that changes in one parameter may change the values of other parameters set earlier; for example, clipped size may change if VL_PACKING is set after VL_SIZE.

Table 3-4 Data Transfer Controls

Control	Basic Use	Video Nodes	Memory and Codec Nodes
VL_FORMAT	Video format on the physical connector	See "Using VL_FORMAT" in this chapter	N/A
VL_TIMING	Video timing	See Table 3-5 for values	N/A
VL_CAP_TYPE	Setting type of field(s) or frame(s) to capture	N/A	VL_CAPTURE_NONINTERLEAVED VL_CAPTURE_INTERLEAVED VL_CAPTURE_EVEN_FIELDS VL_CAPTURE_ODD_FIELDS VL-CAPTURE_FIELDS
VL_PACKING	Pixel packing (conversion) format	N/A	Changes pixel format of captured data; see Table 3-7 for values
VL_ZOOM	Decimation size	N/A	Memory nodes only: any n/m where n is less than or equal to m Codec nodes: N/A
VL_SIZE	Clipping size	Full size of video; read only	Clipped size
VL_OFFSET	Position within larger area	Position of active region; read only	Offset relative to video offset
VL_RATE	Field or frame transfer speed	N/A	If type is INTERLEAVED, rate is in frames; otherwise, it is in fields

To determine default values, use **vlGetControl()** to query the values on the video source or drain node before setting controls. The initial offset of the video node is the first active line of video.

Similarly, the initial size value on the video source or drain node is the full size of active video being captured by the hardware, beginning at the default offset. Because some hardware can capture more than the size given by the video node, this value should be treated as a default size.

For all these controls, it pays to track return codes. If the value returned is VLValueOutOfRange, the value set is not what you requested.

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

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

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

Using VL_TIMING

Timing type expresses the timing of video presented to a source or drain. Table 3-5 summarizes dimensions for VL_TIMING.

Table 3-5 Dimensions for Timing Choices

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

Using VL_FORMAT

To specify video input and output formats of the video signal on the physical connector, use VL_FORMAT. Table 3-6 summarizes the options.

Table 3-6 VL_FORMAT

Format	Explanation
VL_FORMAT_SMPTE_YUV	8-bit YCrCb
VL_FORMAT_RGB	Full-range 8-bit (0-255) RGBA

Using VL_PACKING

A video *packing* describes how a video signal is stored in memory, in contrast to a video format, which describes the characteristics of the video signal.

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

Packing types for eight bits per component are summarized in Table 3-7.

Table 3-7 Packing Types for Eight Bits per Component

Type	63-56	55-48	47-40	39-32	31-24	23-16	15-8	7-0
VL_PACKING_YVYU_422_8 YUV 4:2:2, single-link	U0	Y0	V0	Y1	U2	Y2	V2	Y3
VL_PACKING_RGB_8 RGB, single-link 24-bit word, values beginning with X are ignored	X0	B0	G0	R0	X1	B1	G1	R1

Using VL_ZOOM

In the VL, VL_ZOOM controls the expansion or decimation of the video image. For OCTANE Compression, VL_ZOOM is used in this way:

- OCTANE Compression memory drain nodes support any ratio where the numerator is less than or equal to the denominator—that is, decimation, but not zoom.
- Other OCTANE Compression nodes support zoom and decimation ratios of 1:1 only, that is, neither zoom nor decimation.

Figure 3-4 illustrates decimation.

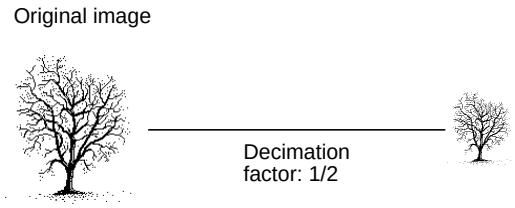

Figure 3-4 Decimation

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

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

Note: For a source, decimation takes place before blending; for a drain, blending takes place before decimation.

This fragment captures half-size decimation video to the screen, with clipping to 320 × 243 (NTSC size minus overscan):

```
val.fractVal.numerator = 1;
val.fractVal.denominator = 2;
if (vlSetControl(server,screen_path, screen_drain_node,
   VL_ZOOM, &val))
{
    vlPerror("Unable to set zoom");
    exit(1);
}
val.xyVal.x = 320;
val.xyVal.y = 243;
if (vlSetControl(server, screen_path, screen_drain_node,
   VL_SIZE, &val))
{
    vlPerror("Unable to set size");
    exit(1);
}
```

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

```
if (vlGetControl(server, screen_path, screen_source, VL_SIZE, &val))
{
   vlPerror("Unable to get size");
   exit(1);
}
if (val.xyVal.x/xsize < val.xyVal.y/ysize)
   zoom_denom = (val.xyVal.x + xsize - 1)/xsize;
else
   zoom_denom = (val.xyVal.y + ysize - 1)/ysize;
val.fractVal.numerator = 1;
val.fractVal.denominator = zoom_denom;
    if (vlSetControl(server, screen_path, screen_drain_node, VL_ZOOM,
    &val))
{
   /* allow this error to fall through */
   vlPerror("Unable to set zoom");
}
val.xyVal.x = xsize;
val.xyVal.y = ysize;
if (vlSetControl(server, screen_path, screen_drain_node,
    VL_SIZE, &val))
{
   vlPerror("Unable to set size");
   exit(1);
}
```

Using VL_SIZE

VL_SIZE controls how much of the image sent to the drain is used, that is, how much clipping takes place. This control operates on the zoomed image; for example, when the image is zoomed to half size, the limits on the size control change by a factor of 2. Figure 3-5 illustrates clipping.

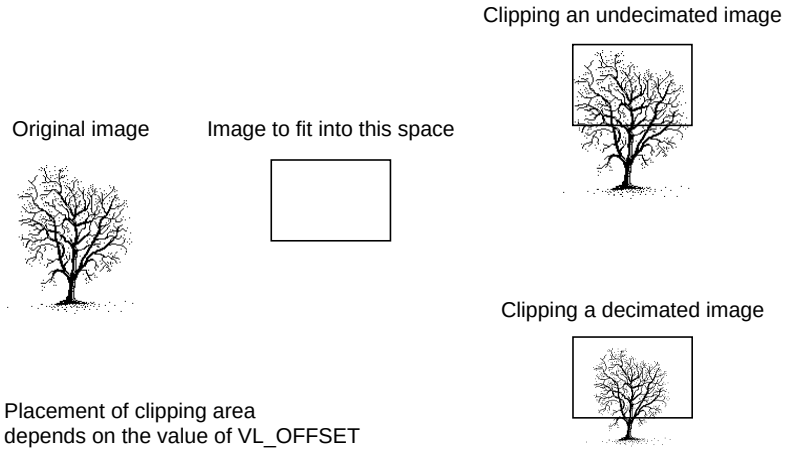

Figure 3-5 Clipping an Image

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

```
VLControlValue value;
value.xyval.x=320;
value.xyval.y=243;
vlSetControl(vlSvr, path, drn, VL_SIZE, &value);
```

Note: Because this control is device-dependent and interacts with other controls, always check the error returns. For example, if offset is set before size and an error is returned, set size before offset.

Using VL_OFFSET

VL_OFFSET puts the upper left corner of the video data at a specific position; it sets the beginning position for the clipping performed by VL_SIZE. The values you enter are relative to the origin.

This example places the data ten pixels down and ten pixels in from the left:

```
VLControlValue value;
value.xyval.x=10;
value.xyval.y=10;
vlSetControl(vlSvr, path, drn, VL_OFFSET, &value);
```

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

Figure 3-6 shows the relationships between the source and drain size, and offset.

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

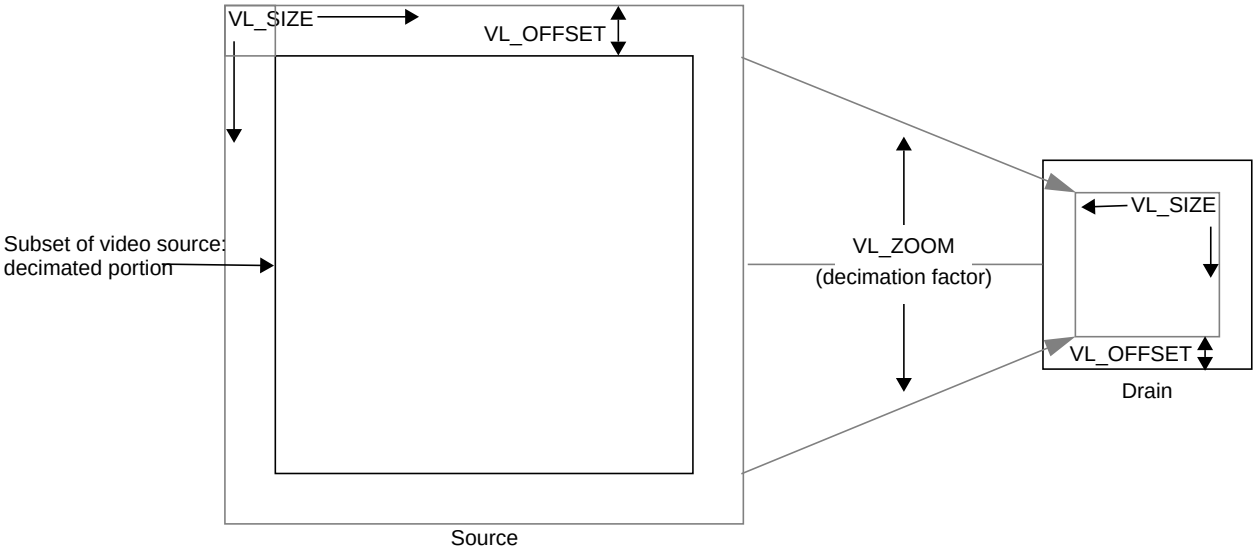

Figure 3-6 Zoom (Decimation), Size, and Offset

Using VL_CAP_TYPE and VL_RATE

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

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

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

```
10101010101010101010...
```

Capture types are as follows:

- VL_CAPTURE_NONINTERLEAVED
- VL_CAPTURE_INTERLEAVED
- VL_CAPTURE_EVEN_FIELDS
- VL_CAPTURE_ODD_FIELDS
- VL_CAPTURE_FIELDS

VL_RATE determines the data transfer rate by field or frame, depending on the capture type as specified by VL_CAP_TYPE, as shown in Table 3-8.

Table 3-8 VL_RATE Values (Items per Second)

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

Note: Not all rates are supported on all memory nodes; see Appendix A, "VL Controls and CL Parameters for the OCTANE Compression Option," for details. The buffer size must be set in accordance with the capture type, as listed in Table 3-10 in this chapter.

VL_CAPTURE_NONINTERLEAVED

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

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

```
1100110011001100...
```

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

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

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

VL_CAPTURE_INTERLEAVED

Interleaved capture interleaves the two fields of a frame and places them in a single buffer; the order of the frames depends on the value set for VL_MGV_DOMINANCE_FIELD (see Table A-3 or Table A-4 in Appendix A for details). OCTANE Digital Video guarantees that the interleaved fields are from the same frame: if one field of a frame is dropped, then both are dropped.

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

```
1100110011001100....
```

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

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

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

VL_CAPTURE_EVEN_FIELDS

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

```
1010101010101010...
```

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

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

```
1000100010001000...
```

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

VL_CAPTURE_ODD_FIELDS

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

VL_CAPTURE_FIELDS

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

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

```
101010101010101010...
```

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

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

Note: The OCTANE Digital Video can make use of the Unadjusted System Time (UST)/Media Stream Count (MSC) feature. See Chapter 2 of the OCTANE Digital Video Programmer's Guide for information.

Setting Field Dominance

Use the control VL_MGC_DOMINANCE_FIELD to set the field dominance mode, which determines the order in which the fields are read from memory. This control applies only to the frame-oriented capture types VL_CAPTURE_INTERLEAVED and VL_CAPTURE_NONINTERLEAVED.

The values for the control VL_MGC_DOMINANCE_FIELD are VL_MGC_DOMINANCE_F1 (the default) and VL_MGC_DOMINANCE_F2. Figure 3-7 diagrams the field dominance values.

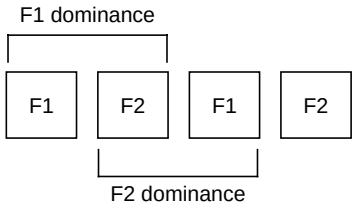

Figure 3-7 Field Dominance

You can set field dominance independently for each DMA channel.

- VL_CAPTURE_INTERLEAVED
  - VL_MGC_DOMINANCE_F1: For video timings VL_TIMING_525_CCIR601 and VL_TIMING_525_SQ_PIX, F1 (odd) dominance dictates that data for the F1 field resides in memory after that for F2. For VL_TIMING_625_CCIR601 and VL_TIMING_625_SQ_PIX, the data for F1 resides in memory before that of F2.
  - VL_MGC_DOMINANCE_F2: For VL_TIMING_525_CCIR601 and VL_TIMING_525_SQ_PIX, F2 (even timings) dominance dictates that data for the F1 field resides in memory before that for F2. For VL_TIMING_625_CCIR601 and VL_TIMING_625_SQ_PIX, the data for F1 resides in memory after that of F2.

  The meaning of *before* and *after* depends on the capture type. For interleaved frames, before indicates that the data comprising the first line of the designated field begins at the first byte of the buffer. In this format, the lines of F1 and F2 are interleaved within the one ring buffer, thus the second line of the buffer belongs to the other field, and so forth.

  For noninterleaved frames, before indicates that the dominant field is in a buffer preceding the buffer(s) containing nondominant fields.
- Values for VL_CAPTURE_NONINTERLEAVED:
  - VL_MGC_DOMINANCE_F1: The F1 field is in the first buffer of the pair, and the F2 field in the second.
  - VL_MGC_DOMINANCE_F2: The F2 field is in the first buffer of the pair, the F1 field in the second.

Padding and Scaling

OCTANE Compression has hardware acceleration for shrinking images that have an original size of up to 1000 × 1000 pixels. Original sizes with height or width larger than 1000 pixels are sized (and optionally converted to the RGB color space) by software on the host CPU.

Table 3-9 lists controls you can use to pad and scale images on capture.

Table 3-9 Padding and Scaling Controls

Control	Values or Range	Type	Use
VL_MGC_HASPECT VL_MGC_VASPECT	0 < value ≤ 1/VL_ZOOM	fractVal	Fraction less than or equal to 1 that shrinks the horizontal or vertical aspect, respectively
VL_MGC_PAD_TOP VL_MGC_PAD_BOTTOM	0	intVal	Number of lines to pad at the top or bottom (respectively of the image on capture
VL_MGC_PAD_LEFT VL_MGC_PAD_RIGHT	0	intVal	Number of pixels to pad at the left or right (respectively) of the image on capture
VL_MGC_PAD_ENABLE	0, 1	boolVal	Boolean value that activates or deactivates padding
VL_MGC_PAD_Y VL_MGC_PAD_U VL_MGC_PAD_V	1 ≤ value ≤ 254	intVal	Value between 16 and 235 that specifies the padding color of the Y, U, or V value, respectively; default is black
VL_MGC_VIDEO_TOP_CLIP	0	intVal	Number of lines to clip from the top on playback to video output

For examples, see */usr/share/src/dmedia/video/vl/OpenGL/contcapt.c*.

Transferring Video Data to and From Devices

The processes for data transfer are as follows:

- creating a buffer for video data (for transfers involving memory)
- registering the VL buffer with the path (for transfers involving memory)
- starting data transfer
- reading data from the buffer (for transfers involving memory)

Each process is explained separately.

Note: You can use either VL buffers or DM buffers. For information on DMbuffers, see Chapter 5 of the *Digital Media Programming Guide* (007-1799-060), or Chapter 2 of the *OCTANE DIgital Video Programmer's Guide* (007-3513-001).

Creating a Buffer for Video Data

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

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

Like other libraries in the IRIX digital media development environment, the VL uses VL buffers. VL buffers provide a way to read and write varying sizes of video data. A frame of data consists of the actual frame data and an information structure describing the underlying data, including device-specific information.

When a VL buffer is created, constraints are specified that control the total size of the data segment and the number of frame or field buffers (sectors) to allocate.

A head and a tail flag are automatically set in a VL buffer so that the latest frame can be accessed. A sector is locked down if it is not called; that is, it remains locked until it is read. When the VL buffer is written to and all sectors are occupied, data transfer stops. The sector last written to remains locked down until it is released.

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

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

For example:

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

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

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

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

where

VLBuffer	is the handle of the buffer to be created
vlSvr	names the server to which the path is connected
path	specifies the data path
node	specifies the memory node containing data to transfer to or from the VL buffer
numFrames	specifies the number of sectors in the buffer (fields or frames, depending on the capture type)

For example:

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

Table 3-10 shows the relationship between capture type and minimum VL buffer size.

Table 3-10 Buffer Size Requirements

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

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

Registering the VL Buffer

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

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

where

vlSvr names the server to which the path is connected

path specifies the data path

memnodeid specifies the memory node ID

buffer specifies the VL buffer handle

For example:

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

Starting Data Transfer

To begin data transfer, use **vlBeginTransfer()**. Its function prototype is

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

where

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

Tailor the data transfer by means of *transfer descriptors*. Multiple transfer descriptors are supplied; they are executed in order. The transfer descriptors are

xferDesc.mode Transfer method:

- VL_TRANSFER_MODE_DISCRETE: a specified number of frames are transferred (burst mode)
- VL_TRANSFER_MODE_CONTINUOUS (default): frames are transferred continuously, beginning immediately or after a trigger event occurs (such as a frame coincidence pulse), and continues until transfer is terminated with **vlEndTransfer()**
- VL_TRANSFER_MODE_AUTOTRIGGER: frame transfer takes place each time a trigger event occurs; this mode is a repeating version of VL_TRANSFER_MODE_DISCRETE

xferDesc.count Number of frames to transfer; if *mode* is VL_TRANSFER_MODE_CONTINUOUS, this value is ignored.

xferDesc.delay Number of frames from the trigger at which data transfer begins.

xferDesc.trigger Set of events to trigger on; an event mask. This transfer descriptor is always required. VLTriggerImmediate specifies that transfer begins immediately, with no pause for a trigger event. VLDeviceEvent specifies an external trigger.

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

This example fragment transfers the entire contents of the buffer immediately.

```
xferDesc.mode = VL_TRANSFER_MODE_DISCRETE;
xferDesc.count = imageCount;
xferDesc.delay = 0;
xferDesc.trigger = VLTriggerImmediate;
```

This fragment shows the default descriptor, which is the same as passing in a null for the descriptor pointer. Transfer begins immediately; *count* is ignored.

```
xferDesc.mode = VL_TRANSFER_MODE_CONTINUOUS;
xferDesc.count = 0;
xferDesc.delay = 0;
xferDesc.trigger = VLTriggerImmediate;
```

Reading Data From the Buffer

If your application uses a buffer, use various VL calls for reading frames, getting pointers to active buffers, freeing buffers, and other operations. Table 3-11 lists the buffer-related calls.

Table 3-11 Buffer-Related Calls

Call	Purpose
vlGetNextValid()	Returns a handle on the next valid frame or field of data
vlGetLatestValid()	Reads only the most current frame or field in the buffer, discarding the rest
vlPutValid()	Puts a frame or field into the valid list (memory to video)
vlPutFree()	Puts a valid frame or field back into the free list (video to memory)
vlGetNextFree()	Gets a free buffer into which to write data (memory to video)
vlBufferDone()	Informs you if the buffer has been vacated
vlBufferReset()	Resets the buffer so that it can be used again

Figure 3-8 illustrates the difference between **vlGetNextValid()** and **vlGetLatestValid()**, and their interaction with **vlPutFree()**.

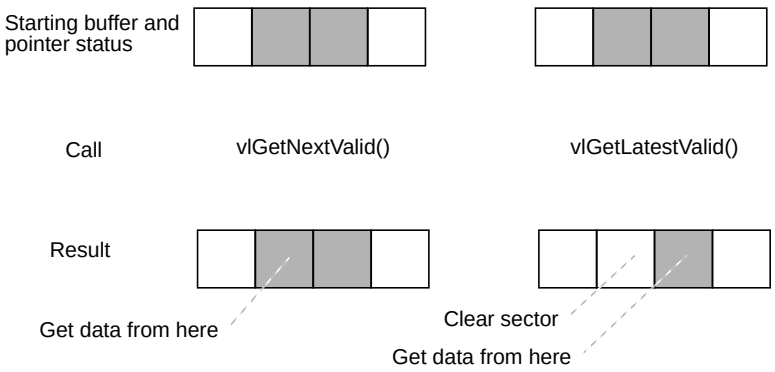

Figure 3-8 vlGetNextValid(), vlGetLatestValid(), and vlPutFree()

Table 3-12 lists the calls that extract information from a buffer.

Table 3-12 Calls for Extracting Data From a Buffer

Call	Purpose
vlGetActiveRegion()	Gets a pointer to the data region of the buffer (video to memory); called after **vlGetNextValid()** and **vlGetLatestValid()**
vlGetDMediaInfo()	Gets a pointer to the DMediaInfo structure associated with a frame; this structure contains timestamp and field count information
vlGetImageInfo()	Gets a pointer to the DMImageInfo structure associated with a frame; this structure contains image size information

Caution: None of these calls has count or block arguments; appropriate calls in the application must deal with a NULL return in cases of no data being returned.

In summary, for video-to-memory transfer, use

```
buffer = vlCreateBuffer(vlSvr, path, memnode1);
vlRegisterBuffer(vlSvr, path, memnode1, buffer);
vlBeginTransfer(vlSvr, path, 0, NULL);
info = vlGetNextValid(vlSvr, buffer);
/* OR vlGetLatestValid(vlSvr, buffer); */
dataptr = vlGetActiveRegion(vlSvr, buffer, info);

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

For memory-to-video transfer, use

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

These calls are explained in separate sections.

Reading the Frames to Memory From the Buffer

Use **vlGetNextValid()** to read all the frames in the buffer or get a valid frame of data. Its function prototype is

`VLInfoPtr vlGetNextValid(VLServer` *vlSvr*`, VLBuffer` *vlBuffer*`)`

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

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

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

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

Sending Frames From Memory to Video

Use **vlGetNextFree()** to get a free buffer to which to write data. Its function prototype is

```
VLInfoPtr vlGetNextFree(VLServer vlSvr, VLBuffer vlBuffer,
      int size)
```

After filling the buffer with the data you want to send to video output, use **vlPutValid()** to put a frame into the valid list for output to video (memory to video). Its function prototype is

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

Caution: These calls do not have count or block arguments; appropriate calls in the application must deal with a NULL return in cases of no data being returned.

Getting DMediaInfo and Image Data From the Buffer

Use **vlGetActiveRegion()** to get a pointer to the active buffer. Its function prototype is

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

Use **vlGetDMediaInfo()** to get a pointer to the DMediaInfo structure associated with a frame. This structure contains timestamp and field count information. The function prototype for this call is

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

Use **vlGetImageInfo()** to get a pointer to the DMImageInfo structure associated with a frame. This structure contains image size information. The function prototype for this call is

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

Ending Data Transfer

To end data transfer, use **vlEndTransfer()**. Its function prototype is

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

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

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

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

- for transfers involving memory: **vlDeregisterBuffer()**, **vlDestroyPath()**, **vlDestroyBuffer()**
- for all transfers: **vlCloseVideo()**

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

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

where

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

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

```
int vlDestroyPath(VLServer vlSvr, VLPath path)
int vlDestroyBuffer(VLServer vlSvr, VLBuffer vlBuffer)
int vlCloseVideo(VLServer vlSvr)
```

This example ends a data transfer that used a buffer:

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

Example Programs

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

- a simple screen application: *simplev2s.c*

  This program shows how to send live video to the screen.

- a video-to-memory frame grab: *simplegrab.c*

  This program demonstrates video frame grabbing.

- memory-to-video frame output *simplem2v.c*

  This program sends a frame to the video output.

- continuous frame capture: *simpleccapt.c*

  This program demonstrates continuous frame capture.

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

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

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

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

Chapter 4

Using the Compression Library With OCTANE Compression

This chapter gives specific information for using the CL with OCTANE Compression. Besides the interfaces presented in Chapter 2, the CL includes JPEG-specific and board-specific CL parameters.

In this chapter:

- "Adding OCTANE Compression Support to an Application" explains how to add OCTANE Compression support to your application.
- "Using OCTANE Compression Image Formats"describes the CL image parameters that OCTANE Compression supports.
- "Getting Compressed Image Information" explains how to get such information as the size, timestamp, and a relative image index value for images as they are compressed or decompressed.
- "Specifying Memory-to-Memory Compression and Decompression" explains how to use memory-to-memory compression and decompression to compress images from a movie file to a buffer or scale down the images as you decompress them.
- "Compressing and Decompressing Video Through External Connections" explains how to use OCTANE Compression to compress images from an external video connection into memory and decompress JPEG images from memory to a video device.

Note: For information on tuning the JPEG algorithm, trading quality for compression ratio, and vice versa, see Chapter 7, "Using Compression Library Algorithms."

Adding OCTANE Compression Support to an Application

To add OCTANE Compression support to your application, follow these steps:

1. Include the *dmedia/cl_impactcomp.h* header in order to get definitions for OCTANE Compression:

   ```
   #include <dmedia/cl_impactcomp.h>
   ```

2. Set OCTANE Compression-specific compression parameters:
   - Set image formats as described in "Using OCTANE Compression Image Formats" on page 88.
   - Enable CL_ENABLE_IMAGEINFO as described in "Getting Compressed Image Information" on page 90.
3. Query the CL to determine the appropriate scheme argument for **clOpenCompressor()** when opening a compressor or **clOpenDecompressor()** when opening a decompressor, as described in "Opening a Compression Session" in Chapter 2.

   Only two OCTANE Compression codecs are available concurrently. An error is returned if no OCTANE Compression codec is available.
4. Compress or decompress frames.

Determining the JPEG Codec

OCTANE Compression has two independent JPEG codecs. When an application opens a compressor or decompressor, one of these codecs is allocated to the application.

The control CL_IMPACT_VIDEO_INPUT_CONTROL is used by the application to determine which codec was allocated; when CL_EXTERNAL_DEVICE is used, it specifies the CL_CODEC node to be used by the VL.

Values for this control are CL_IMPACT_VIDEO_CHANNEL0 and CL_IMPACT_VIDEO_CHANNEL1; they depend upon which codec was allocated. There is no default value for this parameter.

An application can query this parameter at any time, but can set it only before a call to a data-processing routine, such as **clCompress()** or **clDecompress()**. Since the codec channels are identical, it is not usually necessary to select a specific channel.

Although you can try to set the value of this parameter to the other possible value, success is not guaranteed; another application might have the other codec allocated. Example 4-1 shows use of this control.

Example 4-1 Capture Using CL_IMPACT_VIDEO_INPUT_CONTROL

```
channel = clGetParam(clHandle,CL_IMPACT_VIDEO_INPUT_CONTROL);
...
vlGetNode(videoServer,VL_DRN,VL_CODEC,channel);
```

Controlling Compression and Decompression Operation

An application can control compression or decompression with the CL parameter CL_IMPACT_CODEC_CONTROL. The default value of this parameter is CL_IMPACT_START.

If this value of CL_IMPACT_CODEC_CONTROL is CL_IMPACT_START, the operation begins immediately when **clCompress()** or **clDecompress()** is called. If the value is CL_IMPACT_STOP, the CL configures and initializes the hardware necessary for the operation, but does not begin the operation until the value is set to CL_IMPACT_START. This feature allows more precise control over the time that the operation begins.

After a codec has begun operation, setting the parameter CL_IMPACT_CODEC_CONTROL to the value CL_IMPACT_STOP halts the compression or decompression operation. If **clCompress()** or **clDecompress()** was called with CL_CONTINUOUS_BLOCK, the function returns. If **clCompress()** or **clDecompress()** was called with CL_CONTINUOUS_NONBLOCK, the associated thread terminates.

Setting the value to CL_IMPACT_START on a codec that is already processing data has no effect, nor does setting the value to CL_IMPACT_STOP on a codec that is not processing data.

Using OCTANE Compression Image Formats

The Compression Library works with data that is contained in frames. A frame is defined as a sample in time so that:

```
width * height * components * bitsPerComponent/8 = n bytes
```

For video compression or decompression, images must be supplied as fields. Because the JPEG compression algorithm processes images in blocks of 16×8 pixels, OCTANE Compression requires that input images have a height that is a multiple of 8 pixels and a width that is a multiple of 16 pixels. The CL associates two sets of image dimensions with an instance of a video compressor or decompressor:

- CL_IMAGE_WIDTH and CL_IMAGE_HEIGHT
- CL_INTERNAL_IMAGE_WIDTH and CL_INTERNAL_IMAGE_HEIGHT

For compression operations, CL_IMAGE_WIDTH and CL_IMAGE_HEIGHT equal the original, uncompressed image size, and CL_INTERNAL_IMAGE_WIDTH and CL_INTERNAL_IMAGE_HEIGHT equal the final compressed image size.

For decompression operations, CL_IMAGE_WIDTH and CL_IMAGE_HEIGHT equal the final, uncompressed image size, and CL_INTERNAL_IMAGE_WIDTH and CL_INTERNAL_IMAGE_HEIGHT equal the original, compressed image size.

Table 4-1 summarizes the image format parameters.

Table 4-1 OCTANE Compression Image Format Parameters

Image Attribute	Description	Parameter	Values
Pixel format	The option supports 32-bit RGB or YCrCb 4:2:2 for memory-to-memory transfers and YCrCb 4:2:2 only for video-to-memory transfers.	CL_ORIGINAL_FORMAT	CL_RGBX CL_YUV
Interlacing	The option operates on interlaced NTSC or PAL video data for video-to-memory compression and memory-to-video decompression. Even and odd fields are compressed as separate images.	DM_IMAGE_INTERLACING	NTSC or CCIR(525): DM_IMAGE_INTERLACED_EVEN PAL or CCIR(625): DM_IMAGE_INTERLACED_ODD

Table 4-1 (continued) OCTANE Compression Image Format Parameters

Image Attribute	Description	Parameter	Values
Orientation	The option compresses/decompresses images that have top-to-bottom or bottom-to-top orientation.	CL_ORIENTATION	CL_TOP_DOWN CL_BOTTOM_UP DM_TOP_TO_BOTTOM (for Silicon Graphics movies)
Dimensions in pixels	Compression operations: original, uncompressed image height. Decompression operations: final, uncompressed image height	CL_IMAGE_HEIGHT	Range: 16–4088, in multiples of 8 (NTSC must use either 240 or 248) Memory-to-memory decompression can be any size less than or equal to CL_INTERNAL_IMAGE_HEIGHT Default: 248
	Compression operations: original, uncompressed image height. Decompression operations: final, uncompressed image width	CL_IMAGE_WIDTH	Range: 16-4080 in multiples of 16 Memory-to-memory decompression can be any size less than or equal to CL_INTERNAL_IMAGE_WIDTH Default: 640
Dimensions in pixels	Compression operations: final, compressed image height). Decompression operations: original, compressed image height.	CL_INTERNAL_IMAGE_HEIGHT	Range: 16–4088, in multiples of 8 Default:
	Compression operations: final, compressed image width. Decompression operations: original, compressed image width.	CL_INTERNAL_IMAGE_WIDTH	Range: 16-4080, in multiples of 16 Default: 320

Getting Compressed Image Information

The CL provides a function used exclusively by hardware-assisted JPEG operations that lets you get information such as the size, timestamp, and a relative image index value for images (fields or frames) as they are compressed or decompressed through OCTANE Compression. For compressing from external video, the timestamp returned represents the time at which the first line of the uncompressed field arrived at the OCTANE Compression board.

Note: If an application attempts to decompress data that is not valid JPEG data, the decompressor can hang.

To get compressed image information, follow these steps:

1. Call **clSetParam()** to set the CL_ENABLE_IMAGEINFO parameter to TRUE before compressing or decompressing any frames.
2. Call **clGetNextImageInfo()** to get a structure containing information about the compressed image:

   `int clGetNextImageInfo(CL_Handle` *handle*`, CLimageInfo *`*info*`,`
   `int` *sizeofimageinfo*`)`

   handle specifies an open handle that is actively compressing or decompressing

   info is a pointer where a CLimageInfo structure is to be placed

   sizeofimageinfo specifies the size of the CLimageInfo structure in bytes

   The CLimageInfo structure is defined in *dmedia/cl.h* and has the following fields:

```
typedef struct {
    unsigned size; /* size of compressed image in bytes */
    long long ustime;            /* time in nanoseconds */
    unsigned imagecount;       /*  media stream counter */
    unsigned status;   /* additional status information */
} CLimageInfo;
```

   The *ustime* field returns a meaningful value only when compressing from or decompressing to an external device. The status field is reserved for future use.

Note: To get valid JPEG data, an application using the compressor must enable **clGetNextImageInfo()** by setting CL_ENABLE_IMAGEINFO, and then read a CLimageInfo structure corresponding to each compressed image, before calling **clQueryValid** to read the compressed image data.

For the decompressor, you do not need to read CLimageInfo structures. When **clGetNextImageInfo()** is called, the CL queries the hardware for information pertaining to the field that was most recently displayed on the video hardware. **clGetNextImageInfo()** blocks only when it is waiting for the first valid decompressed field to exit the decompressor.

Specifying Memory-to-Memory Compression and Decompression

You can use OCTANE Compression to compress images from a memory archive to a buffer. For example, you can use OCTANE Compression to compress images from a movie file to a buffer, and then insert the JPEG-compressed images into a movie file to create a compressed movie. Taking this idea a step further, you can then use OCTANE Compression to scale down the images as it decompresses them, in order to display thumbnail images similar to the ones in Movie Player.

Memory-to-Memory Compression

To compress frames into memory using OCTANE Compression:

1. Open an OCTANE Compression compressor.
2. Set the CL image parameters to characterize the input image data.
3. Compress images into memory.

When compressing images from memory into a buffer, OCTANE Compression supports image widths of 16–4080 (in multiples of 16 pixels) and image heights of 16–4088 (in multiples of 8 pixels). Images may be scaled down to one half horizontally and/or one half vertically. Images may also have black padding regions added to the image prior to the scaling operation.

The CL parameters CL_IMAGE_WIDTH and CL_IMAGE_HEIGHT specify the original uncompressed image size, and the parameters CL_INTERNAL_IMAGE_WIDTH and CL_INTERNAL_IMAGE_HEIGHT specify the final compressed image size.

The uncompressed data format must be 32-bit RGB (CL_RGBX) or YUV 4:2:2 (CL_YUV), and the uncompressed image size cannot be larger than 4080 × 4088 pixels.

NTSC video frames have a height of 243 lines, but OCTANE Compression supports only input image heights that are multiples of 8. For NTSC, you must specify an image height

of either 240 (causing the image to be cropped 3 lines from the bottom) or 248 (causing the image to be padded with 5 extra lines of black).

Example 4-2 demonstrates memory-to-memory compression of NTSC video.

Example 4-2 Memory-to-Memory Compression

```
#include <dmedia/cl.h>
...
    int pbuf[][2] = {
        CL_IMAGE_WIDTH,  0,
        CL_IMAGE_HEIGHT, 0,
        CL_COMPRESSED_BUFFER_SIZE, 0
    };
     ...
    scheme = clQuerySchemeFromName (CL_ALG_VIDEO, "impact");
    if (scheme < 0) {
        fprintf(stderr, "compression scheme ;'impact' is"
                        " not configured\n");
        return;
    }
    clOpenCompressor (scheme, &handle);

    /* set parameters */
    pbuf[0][1] = 640;
    pbuf[1][1] = 240;
    clSetParams(handle, (int *)pbuf, 3);

    /* allocate the required size buffer */
    clGetParams(handle, (int *)pbuf, 6);
    compressedBuffer = malloc(pbuf[2][1]);

    for(i = 0; i < numberOfFrames; i++)
    {
        /* Get a frame from somewhere */
        ...
        clCompress(handle, 1, frameBuffer,
                   &compressedBufferSize, compressedBuffer);
        /* Write the compressed data to somewhere else. */
        ...
    }
    clCloseCompressor(handle);
```

After compressing the images, you can use **mvInsertCompressedImage()** to insert the compressed images into a movie file, as described in Chapter 5 of the *Digital Media Programming Guide* (007-1799-060 or later).

Memory-to-Memory Decompression

To decompress JPEG images from memory using OCTANE Compression, follow these steps:

1. Open an OCTANE Compression decompressor.
2. Set the CL image parameters to characterize the output image data.
3. Decompress images into a buffer.

You can shrink the images as they are decompressed, which is useful for displaying thumbnail images. When decompressing images from memory into a buffer, OCTANE Compression supports image widths of 16 to 768 and image heights of 16 to 336.

Scaling can be arbitrary, that is, you can scale the image dimensions down by any amount, and the output image dimensions do not have to be multiples of 8. To shrink images as they are decompressed, make the uncompressed image dimensions (CL_IMAGE_WIDTH and CL_IMAGE_HEIGHT) less than the corresponding compressed image dimensions (CL_INTERNAL_IMAGE_WIDTH and CL_INTERNAL_IMAGE_HEIGHT).

For information on padding and scaling images on capture, see “Padding and Scaling” in Chapter 3.

Interleaving

OCTANE Compression supports interleaving fields as they are being decompressed to memory, and deinterleaving as the fields are compressed from memory. This functionality is useful, for example, for taking field-captured media such as that captured from a video source and converting it to a frame medium, such as for display on the graphics monitor.

The interleaving and deinterleaving capability is available only in the memory-to-memory modes of operation. In other modes, use VL controls to select interleaving; see "Using VL_CAP_TYPE and VL_RATE" in Chapter 3.

The CL parameters that control interleaving are CL_IMPACT_FRAME_INTERLEAVE and CL_IMPACT_INTERLEAVE_MODE:

- The CL_IMPACT_FRAME_INTERLEAVE parameter's two possible values, TRUE and FALSE (the default), turn interleaving on and off.
- The way that the fields are actually interleaved into memory is controlled by the CL_IMPACT_INTERLEAVE_MODE parameter.

When CL_IMPACT_FRAME_INTERLEAVE is TRUE, the CL_IMPACT_INTERLEAVE_MODE parameter specifies which of the two fields occupies the top line of the uncompressed region of memory:

- CL_IMPACT_INTERLEAVE_EVEN specifies that the first field decompressed or compressed occupies the first (top) line of the uncompressed memory buffer. This value is appropriate for PAL and CCIR(625) captured media.
- CL_IMPACT_INTERLEAVE_ODD (the default) specifies that the first field decompressed or compressed occupies the second line of the uncompressed memory buffer. This value is appropriate for NTSC and CCIR(525) captured media.

Note: The width and height of each field to be interleaved must be the same. During compression, the width has the same value as CL_IMAGE_WIDTH; during decompression, the width has the same value as CL_INTERNAL_IMAGE_WIDTH.

Compressing and Decompressing Video Through External Connections

You can use OCTANE Compression as a real-time JPEG codec between your application and the analog video ports on OCTANE Compression or the OCTANE Digital Video option.

Video-to-Memory Compression

To capture video from an external video device using OCTANE Compression, follow these steps:

1. Connect the video device to the appropriate port. For example, use either analog port 1 or digital port 1. Video port connections are managed from the *videopanel* control panel.
2. Open a compressor as described in Example 4-1.
3. Query the CL to retrieve the appropriate VL_CODEC drain node identifier.
4. Open a connection to the video server by calling **vlOpenVideo("")**.
5. Create the video transfer paths.
   - Get the source (VL_SRC) node for the video signal connection by calling **vlGetNode()**.
   - Specify the drain node using the drain node identifier from step 3.
   - Create the path from source to drain by calling **vlCreatePath()**.
   - Set up the path to share (VL_SHARE) data by calling **vlSetupPaths()**.
6. Set the CL parameters for image dimensions, quality factor, and compressed image information (CL_ENABLE_IMAGEINFO).
7. Start the video transfer.
8. Use the CL buffered interface to compress frames by calling **clCompress()** with CL_CONTINUOUS_NONBLOCK as the *framecount* parameter and CL_EXTERNAL_DEVICE as the *frameBuffer* parameter.
9. Call **clGetNextImageInfo()** to get a structure containing information about the compressed image.

Note: Instead of using CL_CONTINUOUS_NONBLOCK, you can call **clCompress()** from a separate thread with the value CL_CONTINUOUS_NONBLOCK. In this case, **clCompress()** does not return until the transfer is complete.

See *capture.c* in */usr/share/src/dmedia/dmrecord/dmrecord.cosmo/* for an example of capturing external video through OCTANE Compression.

Video fields entering OCTANE Compression from the direct video connection are captured into an array of field buffers. The field buffers support field widths from 640 to 768 and field heights from 16 to 336. Field dimensions depend on the video timing, as shown in Table 4-2.

Table 4-2 OCTANE Compression Video Field Dimensions

Video Format	WIdth (Pixels)	Height (Pixels)
NTSC	640	243
PAL	768	288
CCIR(525)	720	243
CCIR(625)	720	288

When the compressed image's height is less than the height of the incoming video fields, the video fields are clipped from the bottom before they are sent to the compressor. When the compressed image's height is greater than the height of the incoming video fields, additional lines of black data are appended to the valid video data before the data is sent to the compressor.

Note: NTSC fields have a height (243 pixels) that is not a multiple of 8. For NTSC capture, you can choose to have your application either throw away 3 lines from the bottom of each field (giving a 240 pixel height) or append 5 extra blank lines to the bottom of each field (giving a 248 pixel height) before compression.

You can scale the captured image to half-size before compressing it. This allows for an additional increase in data compression by factor of 4.

Specify vertical decimation by setting the compressed image height (CL_INTERNAL_IMAGE_HEIGHT) to half the size of the uncompressed image height (CL_IMAGE_HEIGHT). Compressed image heights can range from 16 to 168, and uncompressed image heights can range from 32 to 336.

Specify horizontal decimation by setting the compressed image width (CL_INTERNAL_IMAGE_WIDTH) to half the size of the uncompressed image width (CL_IMAGE_WIDTH) as indicated in Table 4-3.

Note: When CCIR(525) or CCIR(625) images are decimated to one half, they are 360 pixels wide, which is not a multiple of 16 pixels. It is for this reason that one-half horizontal decimation is not available for these image sizes.

Table 4-3 OCTANE Compression Field Widths for Compression With Decimation

Video Format	CL_IMAGE_WIDTH (Pixels)	CL_INTERNAL_IMAGE_WIDTH (Pixels)
NTSC	640	320
PAL	768	384
CCIR(525)	720	360
CCIR(625)	720	360

During video compression from an external device, *CLimageInfo.imagecount* is initialized to 1 when the first field is received by the compressor after calling **clCompress()**. The count advances when a new field arrives. If the compression data buffer fills up, then a field will be dropped, but the *imagecount* continues to increase. An application can thus detect a dropped field by noticing a jump in the *imagecount* field of more than one. The *ustime* indicates the time the uncompressed field entered the compressor.

To select the fields to capture, the application can modify the video parameters associated with the VL_CODEC node. It is here that the application specifies the capture type (any fields, paired fields, odd or even fields only) and the rate at which they should be captured (30/60 fields per second, 10/30 frames per second). See "Setting Parameters for Data Transfer to or From Memory or Codec Nodes" in Chapter 3.

Memory-to-Video Decompression

The connections for decompressing from memory to an external video are set up similarly to those for capturing video, except that a decompressor is opened. See *clInit.c* in */usr/share/src/examples/dmedia/dmplay/dmplay.cosmo/* for example code that initializes the CL for JPEG decompression (optionally through OCTANE Compression) from memory to external video.

Video and audio playback of the decompressed frames require media synchronization. See *dmplay.c* and *streamDecompress.c* in */usr/share/src/examples/dmedia/dmplay* for more information.

Uncompressed fields leaving the JPEG decompressor can optionally be scaled up by a factor of 2 in the horizontal and/or vertical dimensions. NTSC, PAL or CCIR(525)/CCIR(625) fields are then scanned out of the array of field buffers. Horizontal scaling is performed by pixel replication; vertical scaling is performed by line doubling.

If the uncompressed fields leaving the decompressor have fewer lines than the field height required by the NTSC/PAL or CCIR(525)/CCIR(625) connection (after optional pistoling), additional lines of black data are added at the bottom of the uncompressed images. If the uncompressed fields leaving the decompressor have more lines than the NTSC/PAL/CCIR(525)/CCIR(625) field height (after optional pistoling), lines are clipped from the bottom of the uncompressed images.

If the uncompressed image is too narrow (less than 640 pixels wide for square-pixel NTSC), the OCTANE Compression board adds extra (black) pixels to make the image the correct width. For example, if the image is 400 pixels wide, the OCTANE Compression board adds 240 black pixels.

Specify horizontal scaling by setting the uncompressed image width (CL_IMAGE_WIDTH) that is twice the compressed image width (CL_INTERNAL_IMAGE_WIDTH) as indicated in Table 4-4.

Table 4-4 OCTANE Compression Field Widths for Decompression

Video Format	CL_IMAGE_WIDTH (Pixels)	CL_INTERNAL_IMAGE_WIDTH (Pixels)
NTSC	640	320
PAL	768	384
CCIR(525)	720	Not available
CCIR(625)	720	Not Available

Specify vertical scaling by setting the uncompressed image height (CL_IMAGE_HEIGHT) to twice the size of the compressed image height (CL_INTERNAL_IMAGE_HEIGHT). Compressed image heights can range from 16 to 168, and uncompressed image heights can range from 32 to 336.

During video decompression to an external device, *CLimageInfo.imagecount* reflects the count of fields sent by the application to the decompressor. The *ustime* indicates the time that field left the decompressor. In certain situations, fields are repeated on output, in which case the *imagecount* remains the same, but the *ustime* increases.

Use the CL parameter CL_IMPACT_CODEC_CONTROL to control compression or decompression. When a codec is opened, this parameter is initialized with the value CL_IMPACT_START. If this value is CL_IMPACT_START when **clCompress()** or **clDecompress()** is called, the operation begins immediately. If the value is CL_IMPACT_STOP, the operation configures and initializes the hardware necessary for the operation, but does not begin the operation until the value is set to CL_IMPACT_START.

After a codec has begun operation, setting CL_IMPACT_CODEC_CONTROL to the value CL_IMPACT_STOP halts the compression or decompression operation. If **clCompress()** or **clDecompress()** was called with CL_CONTINUOUS_BLOCK, the function returns. If **clCompress()** or **clDecompress()** was called with CL_CONTINUOUS_NONBLOCK, the associated thread terminates.

Setting Interlacing for NTSC and PAL

For video-to-memory compression and memory-to-video decompression, use the control DM_IMAGE_INTERLACING to set interlacing for NTSC or PAL video data:

- for NTSC or CCIR(525), use DM_IMAGE_INTERLACED_EVEN
- for PAL or CCIR(625), use DM_IMAGE_INTERLACED_ODD

Even and odd fields are compressed as separate images.

Chapter 5

Using Video Library Controls

Video Library (VL) controls enable you to

- specify data transfer parameters, such as the frame rate or count
- specify the capture region and decimation, or output window
- specify video format and timing
- adjust signal parameters, such as hue, brightness, vertical sync, and horizontal sync
- specify sync source

This chapter explains

- VL control type and values
- VL control fraction ranges
- VL control classes
- VL control groupings

Device-independent controls are documented in */usr/include/dmedia/vl.h*.
Device-dependent controls for the OCTANE Digital Video option are documented in the header files

- */usr/include/dmedia/dev_mgv.h* (linked to */usr/include/vl/vl_mgv.h*)
- */usr/include/dmedia/dev_impact.h* (linked to */usr/include/vl/vl_impact.h*)
- */usr/include/dmedia/dev_mgc.h* (linked to */usr/include/vl/vl_mgc.h*)

Note: For information on the controls used for specific nodes, see Appendix A.

Table 5-1 is an alphabetical list of device-independent VL controls that apply to the Compression option, along with their values or ranges. For a complete listing of VL controls for OCTANE Compression, see Appendix A

Table 5-1 Device-Independent VL Controls

Control	Purpose	Comments
VL_CAP_TYPE	Type of frame(s) or field(s) to capture	
VL_DEFAULT_SOURCE	Default source for the video path	
VL_DEFAULT_DRAIN	Default drain for the video path	
VL_FORMAT	Video format	
VL_FREEZE	Data transfer freeze; suspends transfer at the drain node, used only for analog video out	0 = off 1 = on
VL_OFFSET	On VL_VIDEO nodes, the offset to the active region of the video; on all other nodes, the offset within the video Because the default is 0,0, use negative values to get blanking data	
VL_PACKING	Packing of video data at source or drain	
VL_RATE	Transfer rate in fields or frames	
VL_SIZE	On VL_VIDEO nodes, the size of the video; on all other nodes, the clipped size of the video	
VL_SYNC	Sync mode	VL_SYNC_INTERNAL VL_SYNC_GENLOCK VL_MGC_SYNC_SLAVE
VL_SYNC_SOURCE	Sets sync source for analog breakout box	0 = composite 1 = S-Video 2 = genlock
VL_TIMING	Video timing	
VL_ZOOM	Decimation	Memory nodes only: n/m where n ≤ m

Note: For detailed information on using VL_CAP_TYPE, VL_FORMAT, VL_OFFSET, VL_PACKING, VL_RATE, VL_SIZE, and VL_TIMING, see "Setting Parameters for Data Transfer to or From Memory or Codec Nodes" in Chapter 3.

VL Control Type and Values

The type of VL controls is

```
typedef long VLControlType;
```

Common types used by the VL to express the values returned by the controls are

```
typedef struct __vlControlInfo {
    char name[VL_NAME_SIZE]; /* name of control */
    VLControlType type;      /* e.g. WINDOW, HUE, BRIGHTNESS */
    VLControlClass ctlClass; /* SLIDER, DETENT, KNOB, BUTTON */
    VLControlGroup group;    /* BLEND, VISUAL QUALITY, SIGNAL, SYNC */
    VLNode node;             /* associated node */
    VLControlValueType valueType;       /* what kind of data do we have */
    int valueCount;             /* how many data items do we have */
    int numFractRanges;     /* number of ranges to describe control */
    VLFractionRange *ranges; /* range of values of control */

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

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

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

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

The control info structure is returned by a **vlGetControlInfo()** call, and it contains many of the items discussed above.

VLControlInfo.number is the number of the *VLControlInfo.node* that the information pertains to. There may be several controls of the same type on a particular node, but usually there is just one.

VLControlInfo.numFractRanges is the number of fraction ranges for a particular control. The names correspond 1-to-1 with the *rangeNames*, up to the number of range names, *numRangeNames*. That is, there may be fewer names than ranges, but never more.

VL Control Fraction Ranges

The VL uses fraction ranges to represent the values possible for a control. A VLFractionRange generated by the VL is guaranteed never to generate a fraction with a zero denominator, or a fractional numerator or denominator.

For a range type of VL_LINEAR, *numerator.increment* and *denominator.increment* are guaranteed to be greater than zero, and the limit is always guaranteed to be *{numerator,denominator}.base*, plus some integral multiple of *{numerator,denominator}.increment*.

The type definition for fraction types in the header file is

```
typedef struct {
    VLRange numerator;
    VLRange denominator;
} VLFractionRange;
```

VL Control Classes

The VL defines control classes for user-interface developers. The classes are hints only; they are the VL developer's idea of how the control is commonly represented in the real world.

```
#define VL_CLASS_NO_UI          0
#define VL_CLASS_SLIDER         1
#define VL_CLASS_KNOB           2
#define VL_CLASS_BUTTON         3
#define VL_CLASS_TOGGLE         4
#define VL_CLASS_DETENT_KNOB    5
#define VL_CLASS_LIST           6
```

In the list above, VL_CLASS_NO_UI is often used for controls that have no user-interface metaphor and are not displayed in the video control panel or saved in the defaults file.

The VL controls can be read-only, write-only, or both. The VL includes these macros:

```
#define VL_CLASS_RDONLY         0x8000  /* control is read-only */
#define VL_CLASS_WRONLY         0x4000  /* control is write-only */
#define VL_CLASS_NO_DEFAULT     0x2000  /* don't save in default files */

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

The macros test these conditions:

```
#define VL_CLASS_MASK         0xfff

typedef unsigned long VLControlClass; /* from list above */
```

VL Control Groupings

Like control class, control grouping is an aid for the user-interface developer. The groupings are the VL developer's idea of how the controls would be grouped in the real world. These groupings are implemented in the video control panel *vcp*.

The type definition for groupings is

```
typedef char NameString[80];
#define VL_CTL_GROUP_PATH     9    /* Path Controls */
```

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

Table 5-2 summarizes the VL control groupings.

Table 5-2 VL Control Groupings

Grouping	Includes controls for...
VL_CTL_GROUP_BLENDING	Blending; for example, VL_BLEND_B_FCN
VL_CTL_GROUP_VISUALQUALITY	Visual quality of sources or drains; for example, VL_H_PHASE or VL_V_PHASE
VL_CTL_GROUP_SIGNAL	Signal of sources or drains; for example, VL_HUE
VL_CTL_GROUP_CODING	Encoding or decoding sources or drains; for example, VL_TIMING or VL_FORMAT
VL_CTL_GROUP_SYNC	Synchronizing video sources or drains; for example, VL_SYNC
VL_CTL_GROUP_ORIENTATION	Orientation or placement of video signals; for example, VL_ORIGIN
VL_CTL_GROUP_SIZING	Setting the size of the video signal; for example, VL_SIZE
VL_CTL_GROUP_RATES	Setting the rate of the video signal; for example, VL_RATE
VL_CTL_GROUP_WS	Specifying the windowing system of the workstation; for example, VL_WINDOW
VL_CTL_GROUP_PATH	Specifying the data path through the system; these controls, often marked with the VL_CLASS_NO_UI, are often internal to the VL, with no direct access for the user
VL_CTL_GROUP_SIGNAL_ALL	Specifying properties of all signals
VL_CTL_GROUP_SIGNAL_COMPOSITE	Specifying properties of composite signals
VL_CTL_GROUP_SIGNAL_CLUT_COMPOSITE	Specifying properties of composite color lookup table (CLUT) controls
VL_CTL_GROUP_KEYING	Specifying properties of chroma or luma keying controls, such as VL_KEYER_FG_OPACITY
VL_CTL_GROUP_PRO	Specifying values not commonly found on the front panel of a real-world video device; for example, a wipe control
VL_CTL_GROUP_MASK	Masking optional bits to extract only the control group

Chapter 6

Using Compression Library Parameters

The CL has a group of routines for working with a set of state variables called "parameters" that are unique for each instantiation. These routines—**clQueryParams()**, **clGetParams()**, **clSetParams()**, **clGetDefault()**, **clSetDefault()**—are similar to a set of routines in the Audio Library. You can get and set parameters, either individually or as a group; however, all of the parameters have reasonable defaults that are algorithm-dependent and need not be set.

This chapter describes how to use the Compression Library parameters:

- "Compression Library Parameter Definitions" describes parameters by category.
- "Setting and Querying Compression Library Parameters" explains how to use the parameters in programs.

Compression Library Parameter Definitions

Parameters provide state information about or set frame characteristics, data formats, and algorithms for each compressor/decompressor. This section discusses parameters by category.

Image Frame Dimension Parameters

The CL_IMAGE_WIDTH and CL_IMAGE_HEIGHT parameters provide information about image frame dimensions. For more information on these parameters, see "Using OCTANE Compression Image Formats" in Chapter 4.

Data Format Parameters

These parameters describe data formats:

CL_ORIGINAL_FORMAT
: On compression, this is the format of the original video. On decompression, this is the format that you want after decompression. The value, a symbolic constant, is CL_RGB, CL_RGBX (default), CL_RGBA, CL_RGB332, CL_GRAYSCALE, CL_YUV, CL_YUV422, or CL_YUV422DC.

CL_INTERNAL_FORMAT
: Some video algorithms have several “natural” formats that can be compressed without color-space conversion. This parameter allows the selection of one of these formats. The video default is algorithm-specific.

CL_COMPONENTS
: A read-only value, as determined by CL_ORIGINAL_FORMAT, that indicates the number of components in the data. For example, video is generally 1 for gray-scale, and 3 or 4 for color. The default is 4.

CL_BITS_PER_COMPONENT
: The number of bits per component. For OCTANE Digital Video, this value is always 8.

CL_ORIENTATION
: Specifies the orientation of compressed data, which can be one of the following:

  - CL_TOP_DOWN: for pixels arranged top-to-bottom (default)
  - CL_BOTTOM_UP: for pixels arranged bottom-to-top
  - DM_TOP_TO_BOTTOM for Silicon Graphics movies

  The orientation of compressed data is always top down. When compression or decompression is specified, the original format (or final format) of the data can be bottom up. Specify this inversion by setting the CL_ORIENTATION parameter to CL_BOTTOM_UP instead of the default.

Buffer Parameters

These parameters describe buffer sizes and characteristics:

CL_FRAME_BUFFER_SIZE
: The maximum size, in bytes, of the frame buffer. If **clDecompress()** is called with *numberOfFrames* larger than 1, this value should be the frame size × *numberOfFrames*.

CL_COMPRESSED_BUFFER_SIZE
: The maximum size of the compressed data buffer. The default is calculated as the maximum possible size, taking into account all the factors such as algorithm, encoding method, data type, and so on. If you want to use a smaller buffer, you can set this value explicitly. If **clCompress()** is called with *numberOfFrames* larger than 1, this value should be the maximum compressed size of one frame × *numberOfFrames*.

CL_BLOCK_SIZE
: The natural block size of the algorithm in samples. It is most efficient to specify *numberOfFrames* to be a multiple of the block size when calling **clCompress()** or **clDecompress()**.

CL_PREROLL
: The number of blocks of frames that must be supplied to **clDecompress()** before decompressed frames are returned.

CL_FRAME_RATE
: The requested number of frames per second.

CL_FRAME_TYPE
: The decompressor fills in the frame type when it decompresses a frame. Frame type is one of:

CL_KEYFRAME	*frame* is a keyframe
CL_INTRA	equivalent to CL_KEYFRAME
CL_PREDICTED	*frame* contains information about its succeeding frames
CL_BIDIRECTIONAL	*frame* contains information about frames that precede and succeed it

Compression Ratio and Quality Parameters

These parameters control the compression ratio and quality:

CL_ALGORITHM_ID
: A parameter that can be queried to find out the scheme identifier of the algorithm of an open compressor or decompressor.

CL_EXACT_COMPRESSION_RATIO
: A flag determines whether the compression ratio is a target or must be exact. Some algorithm implementations, such as for JPEG, can be only approximated and can never be exact. For algorithms that do support it, it is generally kept within a small range that over time is guaranteed to average out to the specified compression ratio.

JPEG and MPEG Parameters

JPEG has the following additional parameters:

CL_JPEG_COMPONENT_TABLES
: Specifies the IDs of the AC Huffman table, DC Huffman table, and quantization table to be used for each component. This parameter cannot be changed directly; rather, it is set up automatically for processing the selected CL_INTERNAL_FORMAT.

  YUV formats use AC Huffman table 0, DC Huffman table 0, and quantization table 0 for component 0; AC Huffman table 1, DC Huffman table 1, and quantization table 1 for components 1 and 2. RGB formats use tables AC table 0, DC table 0, and quantization table 0 for all components.

CL_JPEG_QUANTIZATION_TABLES
: Sets or gets the quantization tables to be used. For more information, see "Defining and Using Custom JPEG Quantization Tables" in Chapter 7.

CL_JPEG_QUALITY_FACTOR
: A JPEG quantization table scale factor that represents a rough percentage of the image detail preservation. For more information, see "Defining and Using Custom JPEG Quantization Tables" in Chapter 7.

MPEG_VIDEO has the following additional parameter:

CL_END_OF_SEQUENCE
: An end-of-sequence flag. When the decompressor arrives at the end of the sequence, it sets this flag. The default is FALSE (0).

For a summary of parameters and their types, ranges, and defaults, see Table A-1 in Appendix A, "Video Library Controls and Compression Library Parameters for the OCTANE Compression Option.""

Setting and Querying Compression Library Parameters

After a compressor or decompressor is opened, thus specifying the compression scheme to use, various parameters can be modified using **clSetParams()**. All of these parameters have reasonable defaults that are algorithm-dependent and need not be set. Some parameters, such as CL_IMAGE_WIDTH and CL_IMAGE_HEIGHT for video compression, should be set, but setting them is not required.

Getting a List of Parameters and Parameter Types

Use **clQueryParams()** to get a list of valid parameters and their types for a specified a compressor or decompressor. The compressor being queried is identified by its handle. Its function prototype is:

```
int clQueryParams(CLhandle handle,int *paramValuebuffer, int maxLength)
```

where

handle is the handle to a compressor or decompressor.

paramValuebuffer
: is a pointer to an array of *ints* into which **clQueryParams()** can write parameter identifier/parameter type pairs for each parameter associated with the compressor or decompressor. The even (0,2,4,...) entries receive a string that is the parameter identifier. The odd entries (1,3,5,...) receive the parameter type. Parameter type is one of four values:

- CL_RANGE_VALUE, meaning that a parameter can assume a range of values in which the relative magnitude of the value is meaningful—that is, increasing values indicate increasing quantities of whatever this parameter controls, and vice versa.
- CL_ENUM_VALUE, meaning that a parameter assumes values from an enumerated type. The values have a limited range, but there is no inherent relationship between the range values.
- CL_FLOATING_RANGE_VALUE, meaning that a parameter can assume a range of floating point values, in which the relative magnitude of the value is meaningful—that is, increasing values indicate increasing quantities of whatever this parameter controls, and vice versa.
- CL_FLOATING_ENUM_VALUE, meaning that a parameter assumes values from an enumerated type. The values have a limited floating point range, but there is no inherent relationship between the range values.

maxLength
: is the length of the buffer, in *ints*, pointed to by *paramValuebuffer*. If *maxLength* is zero, then *paramValuebuffer* is ignored and only the return value is valid.

clQueryParams() returns the size of the buffer, in *ints*, needed to hold all the parameter identifier/parameter type pairs for the compressor or decompressor identified by *handle*. The parameters are returned in the even locations of *paramValuebuffer*, and their types are returned in the odd locations.

If the size of the *paramValuebuffer* is smaller than the returned value, a partial list of the parameter identifier/parameter type pairs is returned, making it necessary to enlarge the *paramValuebuffer* in order to receive a complete list. To avoid this situation, you can obtain the correct size of the buffer by calling **clQueryParams()** with a NULL buffer pointer and a *maxLength* of 0 to return the actual buffer length without writing any data.

clQueryParams() also reports whether the parameter is one of a set of enumerated types, any integer number within a specific range, or any floating point number within a specific range. In each case, the values are numbers within the range returned by **clGetMinMax()** and have the defaults returned by **clGetDefault()**.

Example 6-1 demonstrates how to get a list of parameters for a specified compressor/decompressor.

Example 6-1 Getting a List of Parameters for a Compressor/Decompressor

```
#include <dmedia/cl.h>
#include <malloc.h>

/*
* Get a buffer containing all the parameters for a specified
* compressor or decompressor.
*/

int *buf, bufferLength;
bufferLength = clQueryParams(handle, 0, 0);
buf = (int *)malloc(bufferLength * sizeof(int));
clQueryParams(handle, buf, bufferLength);
```

Getting the Parameter ID that Corresponds to a Parameter Name

If you know the name of a parameter, but not its identifier, you can use **clGetParamID()** to get the identifier of a parameter from its name.

Its function prototype is:

```
int clGetParamID(CLhandle handle, char *name)
```

Getting and Setting Parameter Values

You can get or set parameter values as a group or individually.

Use **clGetParams()** to return the current values for the parameters referenced in the *paramValuebuffer* array. The values are written into the odd locations of *paramValuebuffer* immediately after the corresponding parameters.

Use **clSetParams()** to set the current state of the parameters referenced in the *paramValuebuffer* array.

To change a state parameter:

1. Call **clQueryParams()** to find out which parameters are available.
2. Call **clGetParams()** to find out the current state.
3. Fill in the even entries of the *paramValuebuffer* array corresponding to the parameters to be changed and then call **clSetParams()**.

The function prototypes are:

```
void clGetParams ( CLhandle handle, int *paramValuebuffer,
                   int bufferLength )
void clSetParams ( CLhandle handle, int *paramValuebuffer,
                   int bufferLength )
```

where

handle is a handle that identifies a compressor or decompressor.

paramValuebuffer
is a pointer to an array of pairs of *ints*. The even elements of this array select the parameters to be read or changed. The subsequent odd elements are the current or new values of these parameters.

bufferLength is the number of *ints* in the buffer pointed to by *paramValuebuffer*.

Alternatively, parameters can be changed individually with **clSetParam()** and **clGetParam()**. **clGetParam()** returns the current value of the parameter. **clSetParam()** returns the previous value of the parameter.

The function prototypes are:

```
int clGetParam(CLhandle handle, int paramID)
int clSetParam(CLhandle handle, int paramID, int value)
```

where

handle is a handle that identifies a compressor or decompressor.

paramID is the identifier of the parameter to get or set.

value is the new value of the parameter.

Example 6-2 demonstrates how to extract the current value of specific parameters from a list of parameters returned as a group. In this case, the current block size and preroll values are obtained from the list of parameters that are returned in *paramValuebuffer* from **clGetParams()**.

Example 6-2 Getting the Current Values of Selected Parameters

```
#include <dmedia/cl.h>
...
/* Get the block size and preroll */
int paramValueBuffer[][2] = {
CL_BLOCK_SIZE, 0,
CL_PREROLL, 0
};
clGetParams(handle, (int *)paramValueBuffer,
sizeof(paramValueBuffer) / sizeof(int));
/* paramValueBuffer[0][1] is the block size */
/* paramValueBuffer[1][1] is the preroll */
```

Getting or Setting the Value of a Floating Point Parameter

Some parameters, such as CL_FRAME_RATE, are floating point values. You don't have to cast expressions involving floating point values, because macros are provided within *libcl* that handle the conversions for you; even though a value is a *float* you can cast to an *int*. To set a floating point value, use the macro **CL_TypeIsInt()**; to retrieve a floating point value, use the macro **CL_TypeIsFloat()**.

The argument must be a variable, because the type definitions in */usr/include/dmedia/cl.h* are

float `*(float *) &value`

int `*(int *) &value`

Example 6-3 demonstrates how to use the *libcl* macros to get/set a floating point parameter value.

Example 6-3 Using Macros to Get or Set the Value of a Floating Point Parameter

```
float number;
number = 3.0;
...
clSetParam(handle, CL_COMPRESSION_RATIO, CL_TypeIsInt(number));
number = CL_TypeIsFloat(clGetParam(handle,CL_COMPRESSION_RATIO));
```

Getting or Setting Individual Parameter Attributes

You can query parameters individually to get the name, defaults, and range of valid values, given the parameter identifier and a handle.

Use **clGetName()** to return a pointer to a null-terminated string that supplies the English name of a parameter. Its function prototype is

```
char* clGetName(CLhandle handle, int param)
```

where

handle is a handle that identifies a compressor or decompressor.

param is a parameter identifier.

Use **clGetDefault()** to return the default value of the parameter specified by *param*. Use **clSetDefault()** to set the default value. Setting the default value is particularly useful when an algorithm has been added and new defaults need to be set.

The function prototypes are

```
int clGetDefault(CLhandle handle, int param)
int clSetDefault(int scheme, int paramID, int value)
```

where

handle is a handle that identifies a compressor or decompressor.

paramID is a parameter identifier.

scheme is the identifier of the scheme for which to set the defaults.

value is the new default value associated with *param*.

Example 6-4 demonstrates how to get and set defaults for a parameter. In this case, the default for the CL_ORIGINAL_FORMAT parameter is set to CL_RGBX for the specified decompressor.

Example 6-4 Getting and Setting Parameter Defaults

```
#include <dmedia/cl.h>
int default;
...
clOpenDecompressor(scheme, &handle);
...
default = clGetDefault(handle, CL_ORIGINAL_FORMAT);
clSetDefault(scheme, CL_ORIGINAL_FORMAT, CL_RGBX);
...
```

Use **clGetMinMax()** to get the maximum and minimum values for a parameter. Use **clSetMin()** and **clSetMax()** to set new minimum and maximum parameter values, or to establish the minimum and maximum values when adding a new algorithm.

The function prototypes are

```
int clGetMinMax ( CLhandle handle, int param, int *minParam,
                  int *maxParam)
int clSetMin(int scheme, int paramID, int min)
int clSetMax(int scheme, int paramID, int max)
```

where

handle is a handle that identifies a compressor or decompressor.

paramID is a parameter identifier.

minParam is a pointer to the parameter into which **clGetMinMax()** can write the minimum value associated with *paramID*.

maxParam is a pointer to the parameter into which **clGetMinMax()** can write the maximum value associated with *paramID*.

scheme is the identifier of the scheme that is to have its minimum or maximum value changed.

min is the new minimum value associated with *paramID*.

max is the new maximum value associated with *paramID*.

Example 6-5 demonstrates how to get and set the minimum and maximum values of a particular parameter for the specified compressor or decompressor.

Example 6-5 Getting and Setting Minimum and Maximum Parameter Values

```
#include <dmedia/cl.h>
int oldMin, oldMax;
...
clOpenDecompressor(scheme, &handle);
6
...
clGetMinMax(handle, CL_ORIGINAL_FORMAT, &oldMin, &oldMax);
clSetMin(scheme, CL_ORIGINAL_FORMAT, CL_RGB);
clSetMax(scheme, CL_ORIGINAL_FORMAT, CL_RGB332);
...
```

Using Frame Type Parameters

Some compression algorithms do not allow direct compression or decompression of an arbitrary frame. These algorithms—MPEG, CCITT H.261, and so on—have blocks of frames, where each frame can be decompressed only if all previous frames in the block have been decompressed. The frame at the beginning of the block is called a *keyframe.*

A frame can be queried for its status as a keyframe by using the CL_FRAME_TYPE state parameter. Legal values are CL_KEYFRAME (or CL_INTRA), CL_PREDICTED, and CL_BIDIRECTIONAL. Predicted frames use information from a previous keyframe, bidirectional frames use information from both previous and future reference frames, where a reference frame is either of the other two types—CL_KEYFRAME or CL_PREDICTED. The Compression Library interface allows keyframe control from the application.

Some algorithms contain only keyframes, such as JPEG, MVC1, RTR, RLE, G.711, and so on. MPEG Video is the only algorithm currently supported that has all three types of frames.

Chapter 7

Using Compression Library Algorithms

This chapter describes how to use the algorithms that are supplied with *libcl*. To use one of these algorithms, you need to select an appropriate algorithm for your application and specify it in the compress or decompress routines.

In this chapter:

- "Choosing a Compression Library Algorithm" gives factors for selecting an algorithm for specific types of applications.
- "Querying Compression Library Algorithms" tells how to get a list of available algorithms, the name and type of the algorithm, and licensing information for it.
- "Controlling JPEG Compressed Image Quality" explains controls for optimizing the JPEG compression algorithm.

Choosing a Compression Library Algorithm

Perhaps the most important aspect of developing an application that uses *libcl* is selecting the appropriate algorithm to use for the application. The algorithm affects the data size and quality and the rate of compression and decompression, so it is important to consider how an algorithm might affect the end result and whether a particular algorithm achieves the desired effect. A certain amount of experimentation may be necessary.

If you are interested in a particular quality level, you need to set the compression ratio to achieve that quality; if you are primarily interested in a particular data size or data rate, you need to set the compression ratio to achieve the desired data size or rate.

Here are some suggestions for typical application categories:

Note: The performance quoted is for Indigo® workstations with 33 MHz MIPS® R3000® processors only. Corresponding capabilities for the OCTANE workstation have not yet been measured, but are expected to surpass these statistics generally.

- multimedia information delivery applications

  The key factors to consider when choosing a video compression algorithm for multimedia applications are playback speed, data size or rate, and quality.

  MPEG gives the best video quality for a given data size or rate, but playback speed is limited by the CPU. MVC1 is usually the best choice if MPEG is not fast enough. If an expensive frame-by-frame VCR is not available, recording in real time to disk is important, which can be done with RTR1.

- telecommunications applications

  The key factors to consider when choosing a video compression algorithm for video/voice mail, video teleconferencing, and other telecommunications applications are the combined compression-decompression speed, data size/rate, and to a lesser extent, quality.

  MVC1 gives the best result for video of about 10 frames per second for a 160 by 120 frame size at the cost of a very high data rate. More performance can be achieved by using gray-scale.

- previewing animations

  The key factors when choosing a video compression algorithm for previewing 2D and 3D animations are playback speed, quality, and, to a lesser extent, data size/rate. MVC1 gives the appropriate speed and quality.

- editing movies

  The key factors to consider when choosing a video compression algorithm for movie editing applications are decompression speed, image quality, data size/rate, and compression speed.

  For motion video applications, MVC1 is the best choice, especially when the playback is provided by the MoviePlayer tool. MVC1 provides rapid decompression. Playback speed can be traded off with image quality. When recording from video hardware to disk, recording in real time to disk is important if a frame-by-frame VCR is not available—leading to the use of RTR1.

Table 7-1 summarizes the compression and performance relationships of the image and motion video algorithms. Compression, decompression, and codec performance measurements are in frames per second (FPS), as measured for 320 by 240 frames on Indigo workstations with 33 Mhz MIPS R3000 processors only.

Table 7-1 Capabilities of Image and Video Algorithms (Indigo Workstations With 33 MHz MIPS R3000)

Algorithm	Typical Compression Ratio From 24-bit RGB	Average Bits per Pixel	Megabits per Second at 15 Frames per Second	Kilobytes per Frame compression	Compress (Frames per Second)	Decompress[a] (Frames per Second)	Codec (Frames per Second)
Uncompressed	1:1	24	27.65	230.4			
RLE 8-bit	4.8:1	5	5.76	48	6	11.5	3.9
MVC1	5.33:1	4.5	5.2	43.2	3	25	2.8
MVC1 Gray-scale	8:1	3	3.456	28.8	7	28	5.6
RTR1	6:1	4	4.608	38.4	NYM[b]	2.5	2.0
RTR1 Gray-scale	9:1	2.67	3.072	25.6	NYM	8	NYM
JPEG	16:1	1.5	1.728	14.4	1.1	1.8	0.7
MPEG	48:1	0.5	0.576	4.8	<< 1	4.75	<<1

a. Decompressed frame per second is the measured performance, including reading the data from disk, decompressing it, and writing it to the screen.

b. NYM: not yet measured.

Note: The corresponding capabilities for OCTANE workstation have not yet been measured, but are expected to surpass these statistics generally.

Querying Compression Library Algorithms

This section explains how you can get a list of available algorithms for a video compressor or decompressor, along with the name and type of algorithm, or find the identifier for an algorithm given its name. Other features of the algorithms can also be queried by the application at run time. Querying algorithms, rather than having hard-coded setups, makes it possible to have an algorithm-independent interface, which lets you take advantage of future algorithms as they are implemented without redesigning your code.

Getting a List of Algorithms

Use **clQueryAlgorithms()** to get a list of algorithms for the compressor or decompressor identified by *handle*. **clQueryAlgorithms()** returns the size of the buffer needed to contain the list of algorithms and their types.

If the size of the *algorithmTypeBuffer* is smaller than the returned value, a partial list of the algorithms and their types is returned, and you must enlarge the *algorithmTypeBuffer* in order to receive a complete list.

The function prototype for **clQueryAlgorithms()** is:

```
int clQueryAlgorithms ( int algorithmMediaType,
                int *algorithmTypebuffer, int bufferLength )
```

where

algorithmMediaType
: is the media type of the algorithm. For OCTANE Digital Video, always set this to CL_ALG_VIDEO.

algorithmTypeBuffer
: is a pointer to an array of *ints* into which **clQueryAlgorithms()** can write algorithm name/type pairs for each parameter associated with *handle*. The even (0,2,4,...) entries receive the algorithm name. The odd entries (1,3,5,...) receive the types.

  The returned types take on one of three values:

CL_COMPRESSOR	for compression
CL_DECOMPRESSOR	for decompression
CL_CODEC	for both compression and decompression

bufferLength
: is the length of the buffer, in *ints*, pointed to by *paramValueBuffer*. If *bufferLength* is zero, then *paramValueBuffer* is ignored and only the return value is valid.

Getting an Algorithm Scheme or Name

Use **clQuerySchemeFromHandle()** or **clQuerySchemeFromName()** to return the algorithm scheme identifier used by the other compression functions. Use **clGetAlgorithmName()** to return the algorithm name. Their function prototypes are:

```
int clQuerySchemeFromHandle(CLhandle handle)
int clQuerySchemeFromName(int algorithmMediaType, char *name)
char *clGetAlgorithmName(int scheme)
```

where

handle
: is a handle to a compressor or a decompressor

algorithmMediaType
: is the media type of the algorithm. For OCTANE Digital Video, always set this to CL_ALG_VIDEO.

name
: is the algorithm name

scheme
: is the algorithm scheme

Example 7-1 demonstrates how to query the CL for a list of algorithms—in this case, video algorithms. The necessary buffer size is returned in the first call to **clQueryAlgorithms()**, and then **malloc()** is used to allocate enough buffer space to store the returned list of video algorithms.

Example 7-1 Getting a List of Compression Library Algorithms

```
#include <dmedia/cl.h>
#include <malloc.h>

int *buffer, bufferLength;
char *name;
/*
* Get a buffer containing all the video algorithms and types
*/
bufferLength = clQueryAlgorithms(CL_VIDEO, NULL, 0);
buffer = (int *)malloc(bufferLength * sizeof(int));
clQueryAlgorithms(CL_VIDEO, buffer, bufferLength);

scheme = clQuerySchemeFromName(handle);
name = clGetAlgorithmName(scheme);
```

Getting License Information

Use **clQueryLicense()** to obtain license information about an algorithm. The returned message is text intended for inclusion in a message box that is displayed for a user, explaining how to license an algorithm. Failure returns the license error code.

The function prototype is:

```
int clQueryLicense ( int scheme, int functionality,
                     char **message)
```

where

scheme is the algorithm scheme.

functionality is the type of algorithm, which can be one of:

- CL_COMPRESSOR for compression
- CL_DECOMPRESSOR for decompression
- CL_CODEC for both compression and decompression

message is a pointer to a returned pointer to a character string containing a message.

Controlling JPEG Compressed Image Quality

JPEG is a tunable algorithm—you can trade quality for compression ratio and vice versa. You can specify a hint (CL_COMPRESSION_RATIO) for an approximate compression ratio, or you can set more explicit quality factors or target bit rates, as described in this section.

The source image is compressed in three basic steps.

1. Data is transformed from spatial to frequency form in eight-by-eight blocks using a discrete cosine transform (DCT).
2. The frequency coefficients are filtered down by a linear quantization.
3. The coefficients are Huffman-encoded into a bit stream.

The process is reversed for decompression.

The quantization step controls the trade-off between image quality and size. The JPEG quantization table is used to scale each of the 64 DCT coefficients. The luminance (Y) and the chrominance (Cr and Cb) components each use a separate table.

The CL provides three methods for controlling image quality from these quantization tables. You can

- specify an overall JPEG quality factor (CL_JPEG_QUALITY_FACTOR) for scaling the default JPEG quantization tables
- manually set the quantization tables using CL_JPEG_QUANTIZATION_TABLES
- specify a target bit rate that you would like the compressed data to approximate

The JPEG algorithm does not allow you to specify exact compression ratios, but the hardware implementation of JPEG used in OCTANE Compression supports the concept of a target bit rate. Specifying CL_BITRATE causes the hardware to create a new quantization table as each field is compressed. If the current field were compressed again, this quantization table would yield the exact target bit rate. Since this bit rate would reduce the maximum capture rate, the CL applies the new quantization table to the next field, since adjacent fields usually have similar compressibility.

Specifying a JPEG Quality Factor

You can use the CL_JPEG_QUALITY_FACTOR parameter to specify a JPEG quantization table scale factor that represents a rough percentage of the image detail preservation. This is one method to control the image loss and therefore the compression ratio for the OCTANE Compression JPEG algorithm.

Each time the quality factor is set, the reference quantization tables are scaled and downloaded into the codec. The formula used to obtain the scale factor is:

```
scalefactor = 50/quality          (quality < 50)
scalefactor = 2 - 2*quality/100;  (otherwise)
```

The default quality is CL_JPEG_QUALITY_DEFAULT, which represents a good-quality compressed image. A quality factor of 1 results in coarse quantization, a high compression ratio, and very poor image quality.
A quality factor of 100 results in the finest possible quantization, a low compression ratio (perhaps even image expansion), and near-perfect image quality. The most useful quality factor is typically in the range of 25–95.

To bypass scaling, specify CL_JPEG_QUALITY_NO_SCALE.

When CL_QUALITY_FACTOR is set, the approximate value of CL_COMPRESSION_RATIO is calculated; when CL_COMPRESSION_RATIO is set, the approximate value of CL_QUALITY_FACTOR is calculated. When decompressing JPEG, **clDecompress()** fills in this value. The actual compression ratio is determined by the quality factor and the image content and therefore may not be exactly what you expect.

Defining and Using Custom JPEG Quantization Tables

You can customize the JPEG quantization tables by using the CL_JPEG_QUANTIZATION_TABLES parameter. To set the tables, specify an unsigned short **qtables[4]* argument. For each *j*, *qtables[j]* must either be NULL or point to a unsigned short[64] area of memory that represents a JPEG-baseline quantization table in natural scanning order. These custom tables are stored as reference tables; then scaled versions of them based on the current CL_JPEG_QUALITY_FACTOR are downloaded into the codec, becoming the tables associated with the ID *j*.

When getting the value of CL_JPEG_QUANTIZATION_TABLES, the CL allocates the required memory and returns the currently used tables, as indicated by CL_JPEG_COMPONENT_TABLES, scaled by the value of CL_JPEG_QUALITY_FACTOR. Your application is responsible for freeing the memory allocated to return these tables.

You can specify the quantization tables on a per-component basis, by using the CL_JPEG_COMPONENT_TABLES parameter. It specifies the IDs of the AC Huffman table, DC Huffman table, and quantization table to be used for each component. You cannot change this parameter for OCTANE Compression; it is set up for YUV422 processing. This setting uses AC Huffman table 0, DC Huffman table 0, and quantization table 0 for component 0; AC Huffman table 1, DC Huffman table 1, and quantization table 1 for components 1 and 2.

Specifying a Bit Rate Target

You can specify a target bit rate for the compressed data stream. The bit rate is the number of bits per second.

```
bitrate = (image_height * image_width * components_per_pixel
             * fields_per_second * 8) / compression_ratio;
```

Useful values for bit rate for NTSC video range from 15,000,000 (2:1 compression) to 3,000,000 (100:1).

Chapter 8

Differences Between OCTANE Compression and Earlier Silicon Graphics Compression Options

OCTANE Compression is functionally the same as the Indigo^2 IMPACT Compression board. The Indigo^2 IMPACT Compression board has, with some exceptions, a superset of the functionality provided by Cosmo Compress option board.

Before an application for the Indigo^2 IMPACT Compression or Cosmo Compress option board can run on the OCTANE Compression option, it must be ported.

This chapter is designed for those porting software from Cosmo Compress to OCTANE Compression. It explains hardware and software differences between the options.

Note: Compression programs that currently execute on the Indigo^2 IMPACT Compression board should be recompiled on the OCTANE workstation because of issues related to the change of operating system.

Hardware Differences

This section describes hardware changes between the options, including additional functionality and restricted functionality.

Data created with OCTANE Compression is compatible with Indigo^2 IMPACT Compression and Cosmo Compress, with these caveats:

- Movies created with low compression ratios (greater than approximately 7:1) do not play in real time on Cosmo Compress hardware.
- Images larger than 768 x 300 pixels cannot be decompressed with Cosmo Compress.

The following points summarize hardware differences between the two compression options:

- Two independent JPEG codecs

  OCTANE Compression and Indigo^2 IMPACT Compression add a second identical and independent JPEG codec circuit. This circuit allows two applications to process JPEG compressed data independently, or allows one application to achieve both JPEG compression and decompression concurrently.

  Instead of depending upon a separate video option board for video input and output, OCTANE Compression and Indigo^2 IMPACT Compression add built-in analog video support. The options can also be installed with OCTANE Digital Video for I/O of digital component video formats; the two boards can operate together.

  OCTANE Compression and Indigo^2 IMPACT Compression include one analog input, which is accessible by both codecs simultaneously, and one analog output, which is accessible by only one codec at a time. Analog genlock capability is included.

  Analog formats supported are standard NTSC and PAL, and non-square pixel format NTSC (CCIR 525) and PAL (CCIR 625).

- Maximum image size

  The Cosmo Compress option requires all images to transit the field buffer memories, even in memory-to-memory modes. OCTANE Compression and Indigo^2 IMPACT Compression remove this limitation, and support image sizes up to 4080 pixels wide by 4088 lines high in memory-to-memory modes.

  OCTANE Compression and Indigo^2 IMPACT Compression do not process images larger than video size (768 x 576 at 50 fields/second) in real time.

OCTANE Compression has the following capabilities not found in Cosmo Compress:

- Compression ratios as low as one output byte for each two input bytes (2:1) in real time (for video-sized images)
- Hardware-implemented approximate target bit-rate control
- Enhanced management of access and transactions on the GIO bus (the Silicon Graphics proprietary option bus used in the Indigo^2 and Indy® workstations):
  - real-time memory-to-memory transfers (of images up to 768 x 576 at 50 fields/second)
  - uncompressed data transfers in top-down, bottom-up, or interleaved (both odd or even) patterns
- Scaler and color-space conversion on each codec subsystem for video-sized images with a maximum size of 1000 x 1000 pixels during memory-to-memory decompression operations; larger images scaled and converted on the host CPU by a thread that the CL creates
  - shrinking of video-sized images
  - YCrCb 4:2:2-to-RGBX color-space conversion

Software Differences

OCTANE Compression CL software uses the same programming paradigms as Indigo^2 IMPACT Compression and Cosmo Compress, with differences necessary to enable the added capabilities of the hardware. The most pervasive change from Cosmo Compress is in the way that the Compression Library and the Video Library interact. OCTANE Compression is treated as a combination VL and CL device, with synchronization between the two libraries being handled at the device driver level.

OCTANE Compression does not have a predefined value for its compression scheme. Instead, as for Indigo^2 IMPACT Compression, applications use the **clQuerySchemeFromName()** routine to query the CL for the current scheme value for the name *impact*. See the example “Memory-to-Memory Compression” on page 90.

Since OCTANE Compression has two JPEG codecs, an application that processes data to a CL_EXTERNAL_DEVICE needs a way of telling the VL which VL_CODEC node to open. (There is a one-to-one correspondence between the two VL_CODEC nodes and the two JPEG codecs.) See "Compressing and Decompressing Video Through External Connections" in Chapter 4 for a discussion and example.

For Cosmo Compress, the application sets CL parameters to control the video capture rate. OCTANE Compression controls the rate with the control VL_RATE on the VL_CODEC node that is the source or drain of the VL path.

OCTANE Compression software includes Internal Video Sync, a synchronization signal that ensures that simultaneous audio and video signals are precisely synchronized with other devices that have this feature. For a complete explanation of how to use Internal Video Sync, see Chapter 5 of the *OCTANE Digital Video Programmer's Guide*.

Appendix A

Video Library Controls and Compression Library Parameters for the OCTANE Compression Option

This appendix summarizes Video Library controls and Compression Library parameters for the OCTANE Compression option:

- "Device Node Controls"
- "Codec Node Parameters"
- "Memory Node Controls"
- "Memory Node DMA Controls"
- "Analog Input and Output Device Controls"

Device Node Controls

Table A-1 summarizes device node controls.

Table A-1 OCTANE Compression Device Node Controls

Control	Default	Type	Use
VL_MGC_DEFAULT_ANALOG _PLAY_SYNC_SOURCE	VL_MGC_SYNC_SOURCE _ANALOG_GENLOCK	intVal	Sets default value of sync source for playback to analog destination. Values are VL_MGC_SYNC_SOURCE_DEFAULT, VL_MGC_SYNC_SOURCE_ANALOG_IN, or VL_MGC_SYNC_SOURCE_ANALOG _GENLOCK
VL_MGC_DEFAULT_DIGITAL _PLAY_SYNC_SOURCE	VL_MGC_SYNC_SOURCE _DIGITAL_GENLOCK	intVal	Sets default value of sync source for playback to digital destination. Values are VL_MGC_SYNC_SOURCE_DEFAULT or VL_MGC_SYNC_SOURCE_DIGITAL _GENLOCK

Codec Node Parameters

Codec parameters fall into several categories:

- image frame dimensions
- data formats
- buffer characteristics
- compression ratio and quality control
- compression algorithms

For more information on these categories, see "Compression Library Parameter Definitions" in Chapter 6.

Table A-2 summarizes codec parameters.

Table A-2 OCTANE Compression Image Format Parameters

Parameter	Values or Range	Use
CL_ALGORITHM_ID	Current ID	Returns ID of current algorithm.
CL_ALGORITHM_VERSION	Current version	Returns version of current algorithm.
CL_BITRATE	10,000 to 100,000,000 bits per second of compressed data (default 0: no bitrate control)	Specifies a target bit rate to which to approximate the compressed data.
CL_BITS_PER_COMPONENT	Always 8 for OCTANE Compression	Number of bits per component.
CL_BLOCK_SIZE	0–2 billion Default: 1; depends on algorithm	Natural block size of algorithm in samples. It is most efficient to specify *numberOfFrames* to be a multiple of the block size when calling **clCompress()** or **clDecompress()**.
CL_COMPONENTS	Always 3 for OCTANE Compression	Read-only value indicating number of components in the data.

Table A-2 (continued) OCTANE Compression Image Format Parameters

Parameter	Values or Range	Use
CL_COMPRESSED_BUFFER_SIZE	0–2 billion Default: maximum possible size, taking into account all the factors such as algorithm, encoding method, data type, and so on.	Maximum number of bytes in compressed data buffer. For a smaller buffer than the default, set this value explicitly. If **clCompress()** is called with *numberOfFrames* larger than 1, set this value to the maximum compressed size of one frame × *numberOfFrames*.
CL_COMPRESSION_RATIO	JPEG: 15.0:1 Default depends on original format and algorithm	Determines whether compression ratio is a target or is exact. Some algorithms (MVC1, JPEG, and MPEG) are tunable, that is, they allow quality to be traded for compression ratio.
CL_ENABLE_IMAGEINFO	0 (FALSE (default)), 1 (TRUE)	Set to TRUE before getting compressed image information (hardware-assisted JPEG operations).
CL_END_OF_SEQUENCE (MPEG_VIDEO only)	0 (FALSE (default)), 1 (TRUE)	Set by decompressor when it arrives at end of sequence.
CL_EXACT_COMPRESSION_RATIO	Always 0 for OCTANE Compression	Determines whether compression ratio is a target or must be exact.
CL_FRAME_BUFFER_SIZE	0–2 billion Default: size of one frame	Maximum amount of compressed data needed for one frame. If **clDecompress()** is called with *numberOfFrames* larger than 1, this value should be the frame size × *numberOfFrames*.
CL_FRAME_RATE	0–1 million; default: 30.0	Requested number of frames per second.
CL_FRAME_TYPE	0–2 CL_KEYFRAME, CL_INTRA: *frame* is a keyframe CL_PREDICTED: *frame* contains information about its succeeding frames CL_BIDIRECTIONAL: *frame* contains information about frames that precede and succeed it	Supplied by decompressor.

Table A-2 (continued) OCTANE Compression Image Format Parameters

Parameter	Values or Range	Use
CL_IMAGE_HEIGHT	Range: 16–4088, in multiples of 8 (NTSC must use either 240 or 248; default is 248) Memory-to-memory decompression can be any size less than or equal to CL_INTERNAL_IMAGE_HEIGHT	Compression: height in pixels of original uncompressed image. Decompression: height in pixels of final uncompressed image.
CL_IMAGE_WIDTH	Range: 16-4080 in multiples of 16 (default: 640) Memory-to-memory decompression can be any size less than or equal to CL_INTERNAL_IMAGE_WIDTH	Compression: width in pixels of original uncompressed image. Decompression: width in pixels of final uncompressed image.
DM_IMAGE_INTERLACING	NTSC or CCIR(525): DM_IMAGE_INTERLACED_EVEN PAL or CCIR(625): DM_IMAGE_INTERLACED_ODD	Interlacing: the option operates on interlaced NTSC or PAL video data for video-to-memory compression and memory-to-video decompression. Even and odd fields are compressed as separate images.
CL_IMPACT_CODEC_CONTROL	CL_IMPACT_START (default) CL_IMPACT_STOP	Initializes and configures hardware for compression or decompression. For more information, see "Determining the JPEG Codec" in Chapter 4.
CL_IMPACT_FRAME_INTERLEAVE	0 (FALSE (default)), 1 (TRUE)	Determines whether to interleave fields as they are being decompressed to memory.
CL_IMPACT_INTERLEAVE_MODE	CL_IMPACT_INTERLEAVE_EVEN (use for PAL and CCIR(625) CL_IMPACT_INTERLEAVE_ODD (default; use for NTSC and CCIR(525))	Sets type of frame interleaving (whether odd or even field occupies top line of uncompressed region of memory), when CL_IMPACT_FRAME_INTERLEAVE is TRUE.
CL_IMPACT_VIDEO_INPUT_CONTROL	CL_IMPACT_VIDEO_CHANNEL0 CL_IMPACT_VIDEO_CHANNEL1	Determines which codec was allocated. When CL_EXTERNAL_DEVICE is used, it specifies the CL_CODEC node to be used by the VL.
CL_INTERNAL_FORMAT	Always CL_FORMAT_YCbCr422 for OCTANE Compression	Selects "natural" format for the video algorithm in use, which can be compressed without color-space conversion.

Table A-2 (continued) OCTANE Compression Image Format Parameters

Parameter	Values or Range	Use
CL_INTERNAL_IMAGE _HEIGHT	Range: 16–4088 Default: 248	Compression: height in pixels of final uncompressed image height. Decompression: height in pixels of original compressed image.
CL_INTERNAL_IMAGE _WIDTH	Range: 16-4080 Default: 640	Compression: width in pixels of final uncompressed image. Decompression: width in pixels of original compressed image.
CL_JPEG_COMPONENT _TABLES (JPEG only)	0, 1; set by CL_INTERNAL_FORMAT	Specifies IDs of AC or C Huffman table for each component.
CL_JPEG_QUALITY_FACTOR (JPEG only)	0-100 (default 75)	Specifies an overall JPEG quality factor for scaling the default JPEG quantization tables: CL_JPEG_QUALITY_DEFAULT to set default compression quality; range 1-100, with 25-95 being the most useful quality factor range. CL_JPEG_QUALITY_NO_SCALE to bypass quantization table scaling.
CL_JPEG_QUANTIZATION _TABLES (JPEG only)	0-100 See “Compression Library Parameter Definitions” in Chapter 6	Sets the quantization tables manually to custom-designed tables stored as reference tables, which this control downloads to codec.
CL_ORIENTATION	CL_TOP_DOWN (default) CL_BOTTOM_UP DM_TOP_TO_BOTTOM (for Silicon Graphics movies)	Image orientation: compress or decompress images that have top-to-bottom or bottom-to-top orientation. Compressed data is always top down unless specified otherwise.
CL_ORIGINAL_FORMAT	CL_RGB, CL_RGBX (default), CL_FORMAT_YCbCr422, CL_FORMAT_XBGR	Symbolic constant from the following, depending on its data type. Compression: sets format of original video. Decompression: sets format desired after decompression for video.
CL_PREROLL	0–2 billion Default: 0, depends on algorithm	Number of blocks of frames to supply to **clDecompress()** before decompressed frames are returned.

Memory Node Controls

Table A-3 summarizes memory node controls.

Note: For more detail on VL controls, see Chapter 3.

Table A-3 OCTANE Compression Memory Node Controls

Control	Values or Range	Type	Use
VL_CAP_TYPE	VL_CAPTURE_NONINTERLEAVED VL_CAPTURE_INTERLEAVED VL_CAPTURE_EVEN_FIELDS VL_CAPTURE_ODD_FIELDS VL-CAPTURE_FIELDS	intVal	Type of field(s) or frame(s) to capture
VL_FORMAT	VL_FORMAT_SMPTE_YUV: 8-bit YCrCb VL_FORMAT_RGB: full-range 8-bit (0-255) RGBA	intVal	Video format on the physical connector
VL_FREEZE	0,1	boolVal	Data transfer freeze; suspends transfer at the drain node, used only for analog video out
VL_MGC_HASPECT VL_MGC_VASPECT	0 < value ≤ 1/VL_ZOOM	fractVal	Fraction less than or equal to 1 that shrinks the horizontal or vertical aspect, respectively
VL_MGC_PAD_TOP VL_MGC_PAD_BOTTOM	0	intVal	Number of lines to pad at the top or bottom (respectively) of the image on capture
VL_MGC_PAD_LEFT VL_MGC_PAD_RIGHT	0	intVal	Number of pixels to pad at the left or right (respectively) of the image on capture
VL_MGC_ENABLE	0, 1	boolVal	Boolean value that activates or deactivates padding
VL_MGC_PAD_Y VL_MGC_PAD_U VL_MGC_PAD_V	1 ≤ value ≤ 254	intVal	Value between 16 and 235 that specifies the padding color of the Y, U, or V value, respectively; default is black

Table A-3 (continued) OCTANE Compression Memory Node Controls

Control	Values or Range	Type	Use
VL_MGC_VIDEO_TOP_CLIP	0	intVal	Number of lines to clip from the top on playback to video output
VL_MGC_F1_EXTRA_OFFSET VL_MGC_F2_EXTRA_OFFSET	Range depends on other controls	intVal	Number of lines to offset on capture and playback of frame 1 or 2, respectively
VL_MGC_VOUT _STARVATION	VL_MGV_DMA_VO_STARV_RPT (default) VL_MGV_DMA_VO_STARV_FLD	intVal	See Table A-4
VL_OFFSET	*(0,0)*	xyVal	Position within larger area
VL_PACKING	See Table 3-7 for values	intVal	Pixel packing (conversion) format
VL_RATE	Depends on capture type as specified by VL_CAP_TYPE	fractVal	Field or frame transfer speed
VL_SIZE	Depends on timing and capture type	xyVal	Clipping size
VL_TIMING	See Table 3-5 for values	intVal	Video timing
VL_ZOOM	Memory nodes only: n/m where n ≤ m	fractVal	Decimation ratio

Memory Node DMA Controls

Table A-4 summarizes memory node DMA controls.

Table A-4 OCTANE Compression Memory Node DMA Controls

Control	Values	Type	Use
VL_MGC_DMA _VIN_ROUND	VL_MGC_DMA_RND_OFF (default) VL_MGC_DMA_RND_ON	intVal	For capture and compression only, when the source is 10-bit digital video from the OCTANE Video board, this control sets GIO DMA memory drain or codec drain to round from 10-bit to 8-bit as follows: VL_MGC_DMA_RND_OFF: disables rounding, truncates instead. VL_MGC_DMA_RND_ON: enables rounding. Only active area data is rounded.
VL_MGC_DMA _ROUND_TYPE	VL_MGC_DMA_RND_SMPLE (default) VL_MGC_DMA_RND_RAND	intVal	For GIO DMA memory drain and codec drain nodes only, sets the rounding type: VL_MGC_DMA_RND_SMPLE (simple rounding): rounds up if bit 1 is one, or rounds down if bit 1 is zero. VL_MGC_DMA_RND_RAND: (randomized rounding): makes the decision whether or not to round up based on comparing the two least significant bits to a random sequence.
VL_MGC_DMA _RAND_ROUND _MODE	VL_MGC_DMA_RND_RAND _RPT (default) VL_MGC_DMA_RND_RAND _FREE	intVal	For GIO DMA memory drain or codec drain, determines whether or not the random sequence used for randomized rounding is repeated. VL_MGC_DMA_RND_RAND_RPT: repeats the random sequence; in this case a shift register is seeded to a fixed value at the start of each odd field. VL_MGC_DMA_RND_RAND_FREE: causes the random sequence to free-wheel.
VL_MGC _DOMINANCE_FIELD	VL_MGC_DOMINANCE_F1 (default) VL_MGC_DOMINANCE_F2	intVal	Sets the field dominance mode, which determines the order in which the fields are read from memory. This control applies only to the frame-oriented capture types (VL_CAPTURE_INTERLEAVED and VL_CAPTURE_NONINTERLEAVED). For more information, see “Setting Field Dominance” in Chapter 3.

Table A-4 (continued) OCTANE Compression Memory Node DMA Controls

Control	Values	Type	Use
VL_MGC_BUFFER _QUANTUM	Default: 1	intVal	The granularity, or quantum, of data transfer required by the application. The video data is padded at the end so that the size of a field/frame is a multiple of VL_MGC_BUFFER_QUANTUM. This control is intended for applications that do I/O directly from the ring buffer, and may consequently require the frame or field size to be a multiple of the device block size. Direct I/O, for example, usually requires that 512 bytes of data be transferred at a time.
VL_MGC_VOUT _STARVATION	VL_MGV_DMA_VO_STARV_RPT (default) VL_MGV_DMA_VO_STARV_FLD The default value for this control is VL_MGV_DMA_VO_STARV_RPT. Therefore, the ring buffer used in the transfer must contain a minimum of two buffer entries (four for VL_CAPTURE _NONINTERLEAVED), so that one buffer can be repeated by the system while the application is filling the second. If only one buffer is used, then the first buffer output is repeated indefinitely and **vlGetNextFree()** never returns a free buffer.	intVal	For memory and codec source nodes only, sets the video output policy to use in data transfer using a GIO DMA channel when the memory node underflows the ring buffer (that is, the application has not filled the ring buffer at the rate that the memory node consumes it, or is repeating data because of rate control). An application can choose between two starvation policies: VL_MGV_DMA_VO_STARV_RPT: Repeats the last unit transferred (field or frame), until the next transfer unit becomes available. For this repetition, the unit is DMAed continuously. VL_MGV_DMA_VO_STARV_FLD: For frames, repeat only the last field until the next transfer unit is available. Once starvation is detected, the nondominant field is output as both the F1 and F2 fields. This policy halves the vertical resolution but eliminates interfield motion blur. In order to repeat, the field is DMAed continuously. If the capture type is a field, this control value causes identical behavior identical to VL_MGV_DMA_VO_STARV_RPT. In each case, video output from system memory resumes when the application places the next field/frame in the ring buffer via **vlPutValid()**.

Analog Input and Output Device Controls

Table A-5 summarizes analog input device (that is, video) controls.

Table A-5 OCTANE Compression Analog Input Device Controls

Control	Default	Type	Use
VL_MGC_APERTURE	2 = 0.5	intVal	Sets aperture factors for luminance for composite and Y/C inputs
VL_MGC_AUFD	0 = off 1 = on (default)	boolVal	Sets automatic field detect
VL_MGC_BANDPASS	1 = one	intVal	Selects bandpass filters for luminance for composite and Y/C inputs
VL_MGC_CHROMA_AGC	0 = slow	intVal	Sets automatic gain control speed for chrominance for composite or Y/C
VL_MGC_CHROMA_GAIN	44/255	fractVal	Adjusts burst and chrominance output level of composite and Y/C simultaneously
VL_MGC_COLOR_KILL_THRES	-938/42	fractVal	Controls level at which burst amplitude decides if composite or Y/C input is color or monochrome when color mode is automatically set
VL_MGC_CORING	1	intVal	Selects coring levels for luminance for composite and Y/C inputs
VL_MGC_FORCE_COLOR	1 = FALSE	boolVal	Forces color input
VL_MGC_LUMA_DELAY	Depends on format	intVal	Changes composite or Y/C luminance delay without affecting chrominance delay
VL_MGC_PAL_SENS	Fraction range: 0,255,1 Default 144	intVal	In PAL timing, the chroma modulation phase inverts every line. Dropouts off the tape can disrupt this pattern. Use this control to set the recovery time constant (maximum for poor quality tape).
VL_MGC_PREFILTER	0 - off	boolVal	Boosts luminance frequency response for composite and Y/C formats
VL_MGC_VNOISE_REDUCER	normal	intVal	Selects mode of vertical noise reduction
VL_MGC_VTR_LOCK	1 = on	boolVal	Locks videotape recorder

Table A-6 summarizes analog output device controls.

Table A-6 OCTANE Compression Analog Output Device Controls

Control	Default	Type	Use
VL_MGC_ANTI_DITHER	Off	boolVal	Removes interference between frequency components generated by dithered graphics images (Y/C and composite out only) and chrominance frequency present in video signals by using a notch filter in luminance
VL_MGC_CHROMA_BAND	0 = standard	boolVal	Selects standard chrominance bandwidth of about 1.3 MHz or enhanced bandwidth (nonstandard) of about 2.5 MHz for composite and Y/C outputs
VL_MGC_COLOR_OUT_KILL	Off	boolVal	Makes composite or Y/C output into monochrome by turning off color burst and chrominance
VL_MGC_DELAY_SYNC	0	fractVal	Like VL_MGC_H_OFFSET or VL_MGC_V_OFFSET, delays timing of entire video signal (sync and picture) relative to timing reference such as genlock; no effect in slave mode for output timing, but with a narrow range: resolution in pixel clock steps
VL_MGC_C_GAIN VL_MGC_YC_GAIN	1	fractVal	Adjusts burst and chrominance output level of composite and C or Y/C (respectively) simultaneously
VL_MGC_H_OFFSET VL_MGC_V_OFFSET	0	fractVal	Delays timing of entire video signal (sync and picture) relative to timing reference such as genlock; no effect in slave mode for output timing
VL_MGC_SCH_PHASE	0	fractVal	Adjusts SC-H phase +/- 180 degrees
VL_MGC_SUB_FREQ	0	fractVal	Provides fine adjustment of composite and Y/C output color subcarrier frequency

Index

D

E

F

G

H

R

S

T

V

W

Z

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3514-001.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389