# SGI Video Library Buffering Mechanisms

**Buffering mechanisms and interfaces available through the SGI Video Library in IRIX$^{TM}$ 6.3, 6.4, and beyond.**

Michael Minakami
minakami@sgi.com

Silicon Graphics, Inc
May 19, 1997

# Contents

# Introduction

The 6.3 and 6.4 digital media releases introduced a new buffering mechanism for use with the Video Library, Digital Media Buffers (or DMbuffers), and two Video Library APIs for accessing it. The plurality of available buffering mechanisms and APIs has led to some confusion as to which VL devices support DMbuffers, which OS releases are required, and which interface should be used. This paper provides a description of the two buffering mechanisms currently available, their associated APIs available in digital media 6.3 and 6.4, and the Video Library's intended direction. Also discussed here are recommendations for long-term development efforts with an eye to cross-platform portability, and recommendations for short-term development efforts with an eye to minimizing porting efforts.

# Video Library Buffering Mechanisms

With Digital Media 6.3 and 6.4, the SGI Video Library supports two buffering mechanisms that may be used to capture or play back video:

- VL Buffers - The original buffering mechanism supported by the SGI Video Library. This buffering mechanism is specific to the Video Library.
- Digital Media Buffers (DMbuffers) - This is a more flexible buffering mechanism allowing video data to be exchanged between video, compression, and graphics devices. It is supported by the Video, Image Converter (dmIC), and Movie libraries, and is available in the 6.3 and 6.4 digital media releases.

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

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

  DMbuffers, in contrast, are contained in a DMbufferpool. The pool itself is unordered - buffers may be obtained from and returned to the pool in any order, and in general buffers have no implied order except when they are exchanged. DMbuffers are exchanged by enqueing a buffer on the receiver's first-in-first-out queue. The application or library is free to impose any processing order once buffers are dequeued or before they are enqueued.
- The Video Library is responsible for ensuring that VL Buffers are of the appropriate size and alignment for the video device, and for allocating the buffers in the **vlCreateBuffer** call. Except in rare cases, applications cannot modify these attributes to suit the needs of another library or device.

  Because DMbuffers may be used with non-video libraries and devices, the application queries each library for it's buffering requirements. The exact DMbuffer pool requirements are the union of all requested constraints and are enforced when the pool is created. For example, if one library requests alignment on 4k boundaries while another requests alignment on 16k boundaries, the 16k alignment will be used. The application may additionally specify its own pool requirements via the **dmBufferSetPoolDefaults** function.

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

Each buffering mechanism has a set of API functions for creating, registering and manipulating buffers. Once a DMbuffer-specific call has been made, the server connection associated with the call is marked as using DMbuffer semantics. Similarly, once a VL Buffer call is made the server connection is assumed to use VL Buffer semantics. All paths created on the server connection must use only one buffering mechanism. Applying a DMbuffer call to a server connection which previously had a VL Buffer call made on it will result in an error return with vlErrno set to VLAPIConflict. A separate server connection may be established to use a second buffering mechanism. [1]

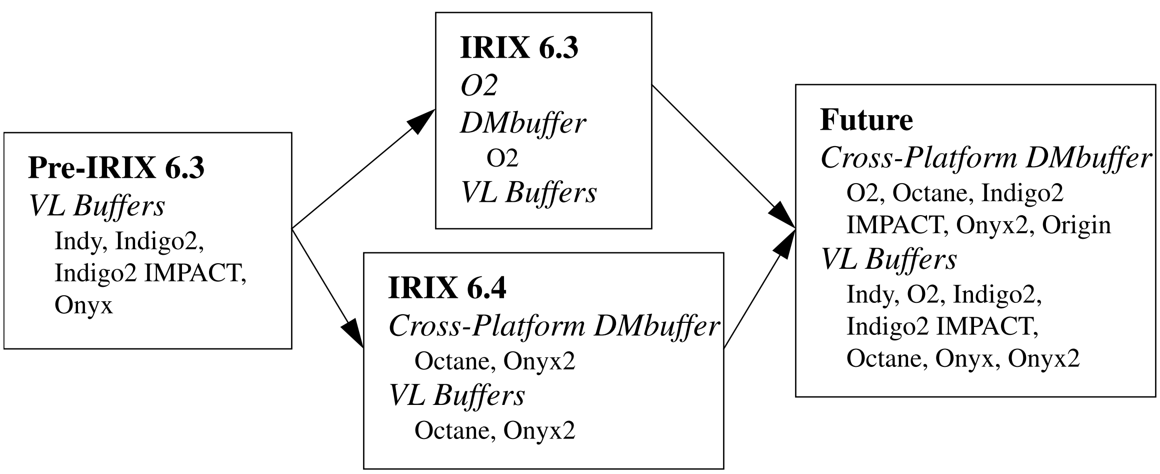


Figure 1: VL DMbuffer API convergence plan

1. This restriction primarily applies to O2. On Octane and Onyx2 it is possible to mix the two buffering mechanisms on a server, as long as each path only uses one buffering mechanism.

**Table 1: Buffering mechanisms and APIs by OS Release**

| Device | IRIX 5.3 / 6.2 | | | IRIX 6.3 | | | IRIX 6.4 | | | Future * | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | VL Buffer | O2 DMbuffer | Cross-Platform DMbuffer | VL Buffer | O2 DMbuffer | Cross-Platform DMbuffer | VL Buffer | O2 DMbuffer | Cross-Platform DMbuffer | VL Buffer | O2 DMbuffer | Cross-Platform DMbuffer |
| VINO Video | Yes | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | Yes | No | No |
| Indy Video | Yes | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | Yes | No | No |
| Galileo Video | Yes | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | Yes | No | No |
| Indigo2 Video, Indigo2 Video for IMPACT | Yes | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | Yes | No | No |
| IMPACT Video | Yes | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | Yes | No | Yes |
| IMPACT Compression | Yes | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | Yes | No | Yes |
| Sirius Video | Yes | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | Yes | No | No |
| O2 Video | N/A | N/A | N/A | Yes | Yes | N/A | N/A | N/A | N/A | Yes | Yes | Yes |
| Octane Digital Video | N/A | N/A | N/A | N/A | N/A | N/A | Yes | No | Yes | Yes | No | Yes |
| Octane Compression | N/A | N/A | N/A | N/A | N/A | N/A | Yes | No | Yes | Yes | No | Yes |
| DIVO Video | N/A | N/A | N/A | N/A | N/A | N/A | Yes | No | Yes | Yes | No | Yes |

*N/A = Device or API not supported in OS release*

* *Relects infromation available at the time of writing. Subject to change.*

### Recommendation for choosing a buffering mechanism

As implied above, Digital Media Buffers represent the buffering mechanism of choice for use by IRIX digital media libraries. The Video Library and Digital Media Image Converter library both use DMbuffers, and it is intended that future SGI digital media libraries will also use DMbuffers as a transport mechanism. Developers are recommended to use DMbuffers when writing new applications or enhancing existing ones. VL Buffers will, however, be retained for the foreseeable future for compatibility with existing applications.

# Flavors of the VL DMbuffer API

The SGI Video Library presently has two independent APIs for using DMbuffers, one available on the O2 for use with O2 Video, and one available on Octane, Onyx2, and Origin for use with the Octane Digital Video, Octane Compression, and DIVO options. Convergence of the two APIs, as depicted in Figure 1 ,  will occur with the next major all-platforms release, with the Octane/Onyx/Origin API also being supported on O2.

Table 1 indicates, for each video device, the APIs supported in each OS release. Note that columns marked “N/A” indicate that either the video device / platform is not supported in that release, or that the API had not been introduced yet.

### Differences between the O2 and Cross-Platform VL DMbuffer APIs

The O2 DMbuffer API implements a new event-based mechanism for receiving buffers. This API consists of the **vlEventRecv**, **vlEventToDMbuffer**, and **vlPathGetFD** functions. Note that on O2, this event mechanism is not interchangeable with the server-based event functions **vlNextEvent**, **vlCheckEvent**, etc. with the exception of **vlGetFD** and **vlPathGetFD**, as explained below. Attempting to mix the two event APIs will result in an error return with vlErrno set to VLAPIConflict.

Using the O2 DMbuffer API, all events are path-based. This contrasts with the server-based event functions, in which all events for a connection arrive on the same file descriptor. The **vlPathGetFD** function returns a file descriptor that can be selected on to determine when events are available on the associated path. The **vlEventRecv** function can then be used to receive the event from the path.

With DMbuffers on O2**,** either **vlPathGetFD** or **vlGetFD** can be used to acquire a file descriptor that becomes ready for reading when events are available. While **vlPathGetFD** returns an fd that becomes ready for reading only when events are available on a specific path, **vlGetFD** returns a file descriptor that becomes ready for reading when the client has an event pending on any path associated with the server connection. The event should then be received using **vlEventRecv**. If **vlGetFD** is used, **vlEventRecv** should be called on all paths associated with the connection - when multiple paths are used it will not be clear which paths have events pending.

DMbuffers containing image data are sent to the client as a VLTransferComplete event. When the client receives a VLTransferComplete event on a video-to-memory path, it should call **vlEventToDMbuffer** on the event to obtain a handle to the associated DMbuffer. (Note: If the event is discarded without returning the associated DMbuffer to the DMbufferpool, the video device will eventually run out of buffers to capture to.)

In the O2 VL DMbuffer API, buffers are sent to the video device using the **vlDMBufferSend** function.

**Table 2: O2 and Cross-Platform API Relationship**

| O2 DMbuffer API | Cross-Platform DMbuffer API |
|---|---|
| vlDMBufferSend | vlDMBufferPutValid |
| vlEventRecv, vlEventToDMbuffer | vlDMBufferGetValid, vlCheckEvent, vlNextEvent |
| vlDMPoolRegister | vlDMPoolRegister |
| vlDMPoolDeregister | vlDMPoolDeregister |
| vlDMPoolGetParams | vlDMGetParams |
| vlPathGetFD | (no equivalent) |
| vlBufferGetFD | vlNodeGetFd |
| (no equivalent) | vlDMBufferGetVideoInfo |
| (no equivalent) | vlDMBufferSetVideoInfo |
| (no equivalent) | vlDMBufferResetNode |
| vlGetFilledByNode | vlDMBufferGetFilledByNode |

The cross-platform VL DMbuffer API attempts to retain a closer programming model to the pre-existing VL Buffer API, with the exception that DMbuffers are used rather than VL Buffers. Buffers are received from the video device via **vlDMBufferGetValid**, while buffers may be sent via **vlDMBufferPutValid**. These are semantically similar to the vlGetNextValid and vlPutValid calls for VL Buffers. Event and buffer delivery remain decoupled, as they were with VL Buffers; applications may capture data without registering for a VLTransferComplete event, although as with VL Buffers, one VLTransferComplete event will be delivered to registered clients when a buffer is captured.

Because the cross-platform API retains the VL Buffer flavor, the concept of a path file descriptor (**vlPathGetFD**) or path events is not available. Events remain connection-based, with the client selecting on a file descriptor return by **vlGetFD** to determine when an event is ready. The cross-platform API does introduce **vlNodeGetFd**, which returns a file descriptor associated with a memory drain node. This file descriptor can be used with select and will be marked ready for reading when a video device sends video data to a client. For developers familiar with VL Buffers , this is conceptually similar to **vlBufferGetFd** and being able to select on the buffer for reading/writing.

Lastly, the cross-platform API provides a data structure for passing video-specific information such as VITC between the video device and the client. **vlDMBufferGetVideoInfo** returns this structure and **vlDMBufferSetVideoInfo** may be used to fill in the structure. This functionality is over and above what was possible with VL Buffers. Note that actual usage of the structure varies on a device basis. Not all devices are capable of supplying VITC or inserting VITC, for example.

A comparison of corresponding O2 dmBuffer and cross-platform VL DMbuffer functions are listed in Table 2 . For developers already familiar with the VL Buffer API, Table 3 lists the corresponding VL Buffer and DMbuffer API functions.

### Recommendations for choosing a DMbuffer API

To maximize portability, it is recommended that the "cross-platform VL DMbuffer" interfaces be used. As this interface is not available on O2 running IRIX 6.3, developers are encouraged to instead use the O2 VL DMbuffer API. This will minimize any porting effort to the cross-platform VL DMbuffer API and allow use of DMbuffer's greater flexibility. Developers wanting to develop cross-platform applications before the all platforms release can safely use both the O2-specific and cross-platform interfaces in their applications. One method of handling the differences is illustrated later in this paper.

**Table 3: VL Buffer and Cross-Platform DMbuffer API differences**

| VL Buffer API | Cross-Platform DMbuffer API |
|---|---|
| vlCreateBuffer | vlDMGetParams,<br>dmBufferCreatePool |
| vlDestroyBuffer | dmBufferDestroyPool |
| vlPutValid | vlDMBufferPutValid |
| vlPutFree | dmBufferFree |
| vlGetNextFree | dmBufferAllocate |
| vlGetNextValid | vlDMBufferGetValid |
| vlGetLatestValid | (no equivalent) |
| vlBufferGetFd | dmBufferGetPoolFD,<br>dmBufferSetPoolSelectSize,<br>vlNodeGetFd |
| vlBufferAdvise | dmBufferSetPoolDefaults |
| vlBufferReset | vlDMBufferResetNode |
| vlBufferDone | (does not apply) |
| vlGetFilled | vlDMBufferGetFilledByNode |
| vlRegisterBuffer | vlDMPoolRegister |
| vlDeregisterBuffer | vlDMPoolDeregister |

**Table 3: VL Buffer and Cross-Platform DMbuffer API differences**

| VL Buffer API | Cross-Platform DMbuffer API |
|---|---|
| vlGetDMediaInfo | dmBufferGetUSTMSCPair |
| vlGetImageInfo | (no equivalent) |

# Using cross-platform DMbuffers to capture or output video

This section describes the programming steps required to use dmBuffers with the Video Library. A similar description of using VL Buffers to do video capture our output can be found in the IRIS Digital Media Programmer's Guide.

### Setting Controls

Set controls on the video, memory and other nodes used by the path. These include VL_SIZE, VL_PACKING, VL_ZOOM and other controls which describe the video as it is stored in a buffer. Based on the control settings, the Video Library will construct a set of requirements for the DMbufferpool, such as the minimum buffer size and alignment.

### Obtaining DMbufferpool Requirements

Before a DMbufferpool is created, the pool requirements of any library that will interact with the pool must be obtained. Pool requirements are maintained in a DMparams list, created using **dmParamsCreate** and initialized by calling **dmBufferSetPoolDefaults**.

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

where

| | |
|---|---|
| *poolParams* | specifies the DMparams list to use for gathering pool requirements |
| *bufferCount* | specifies the number of buffers the pool should contain |
| *bufferSize* | specifies the size of each buffer in the pool |
| *cacheable* | specifies whether buffers allocated from the pool can be cached (DM_TRUE) or not (DM_FALSE). If the CPU will never access the data, marking the buffers non-cacheable may improve performance. If a CPU will be used to manipulate the buffer contents, then the buffer should be marked cacheable. |
| *mapped* | specifies whether the memory allocated for the pool should be mapped as soon as the pool is created (TRUE), or only when **dmBufferMapData** is called (FALSE). If an application requires a pointer to buffer contents, for example to process or store the contents to disk, then the pool should be created mapped. This will improve the performance of the **dmBufferMapData** call. |

The Video Library pool requirements are obtained by calling **vlDMGetParams**:

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

where

| | |
|---|---|
| *svr* | names the server on which video I/O will be performed |
| *path* | specifies the data path performing the transfer |
| *node* | specifies the memory node which the DMbufferpool will be used with |
| *params* | specifies the pool requirements list |

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

[Note: The number of buffers may be increased to accomodate a library's minimum buffer requirements. This is the minimum number of buffers required for the specific library to operate, and may not leave sufficient buffers for other libraries or the application. If multiple libraries are to be used, the application needs to aggregate individual library requirements manually. One method of doing this is to save the value of the DM_BUFFER_COUNT dmParam before calling a "get params" function, set the DM_BUFFER_COUNT in the dmParams list should be set to 0, call the "get parms function", then add the new value of DM_BUFFER_COUNT to the old value. After querying all libraries, set DM_BUFFER_COUNT to the aggregate value before createing the DMbufferpool.]

### Creating a DMbufferpool

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

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

where

| | |
|---|---|
| *poolParams* | specifies the requirements for the pool, as established by **dmBufferSetPoolDefaults** and each library's "get pool params" function |
| *returnPool* | points to a location where the DMbufferpool handle will be stored |

### Registering a DMbufferPool with the Video Library

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

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

where

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

When the video device is ready to capture a new frame or field, it will allocate a DMbuffer from the specified pool, place the field or frame in it, then send the buffer to the application. Note that for video output, it is not necessary to register a pool with the Video Library.

### Receiving Buffers from the Video Library

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

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

where

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

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

### Sending DMbuffers to the Video Library

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

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

where

| | |
|---|---|
| *svr* | specifies the server to which the path is attached |
| *path* | specifies the path on which video is sent |
| *node* | specifies the memory source receiving the buffer |
| *DMbuffer* | specifies the buffer to send |

The DMbuffer may have been obtained from another library, such as dmIC, or generated by the application itself. The **dmBufferAllocate** function can be used to allocate a buffer from a DMbufferPool.

### Freeing a DMbuffer

Once the application is done with a buffer, it should call **dmBufferFree** to indicate that it no longer intends to use the buffer. Note that this does not free the memory from the buffer pool - it only indicates to the buffering mechanism that the application is done using the buffer. Only after all users of a buffer have called **dmBufferFree** can the buffer be reallocated. The Video Library never implicitly releases the application's access to a buffer. Consequently, an application can send the same buffer to a memory node multiple times, or hold a captured image for an indefinite period.

### Cleaning up

**vlDMPoolDeregister** disassociates a DMbufferpool from a memory node. It should be called to cleanup the memory node or allow a new DMbufferpool to be used after a transfer has been stopped.

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

## Using O2 DMbuffers to capture or output video

The following section described capturing and outputting video using the O2 VL DMbuffer API. Note that in terms of initialization and termination, there is a strong resemblance to the cross-platform API. Major differences lie in the receipt of path events and their translation into DMbuffers.

### Setting Controls

As with the cross-platform API, set controls on the video, memory and other nodes used by the path. These include VL_SIZE, VL_PACKING, VL_ZOOM and other controls which describe the video as it is stored in a buffer. Based on the control settings, the Video Library will construct a set of requirements for the DMbufferpool, such as the minimum buffer size and alignment.

### Obtaining DMbufferpool Requirements

Before a DMbufferpool is created, the pool requirements of any library that will interact with the pool must be obtained. Pool requirements are maintained in a DMparams list, created using **dmParamsCreate** and initialized by calling **dmBufferSetPoolDefaults**.

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

where

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

*cacheable* specifies whether buffers allocated from the pool can be cached (DM_TRUE) or not (DM_FALSE). If the CPU will never access the data, marking the buffers non-cacheable may improve performance. If a CPU will be used to manipulate the buffer contents, then the buffer should be marked cacheable.

*mapped* specifies whether the memory allocated for the pool should be mapped as soon as the pool is created (TRUE), or only when **dmBufferMapData** is called (FALSE). If an application requires a pointer to buffer contents, for example to process or store the contents to disk, then the pool should be created mapped. This will improve the performance of the **dmBufferMapData** call.

The Video Library pool requirements are obtained by calling **vlDMPoolGetParams**:

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

where

*svr* names the server on which video I/O will be performed

*path* specifies the data path performing the transfer

*node* specifies the memory node which the DMbufferpool will be used with

*params* specifies the pool requirements list

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

The caveat regarding DM_BUFFER_COUNT mentioned in "Obtaining DMbufferpool Requirements" on page 7 apply to both the O2 VL DMbuffer API and the cross-platform API.

### Creating a DMbufferpool

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

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

where

*poolParams* specifies the requirements for the pool, as established by **dmBufferSetPoolDefaults** and each library's "get pool params" function

*returnPool* points to a location where the DMbufferpool handle will be stored

### Registering a DMbufferPool with the Video Library

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

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

where

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

When the video device is ready to capture a new frame or field, it will allocate a DMbuffer from the specified pool, place the field or frame in it, then send the buffer to the application. Note that for video output, it is not necessary to register a pool with the Video Library.

### Receiving Buffers from the Video Library

After the transfer has been started, captured video may be retrieved using the **vlEventRecv** and **vlEventToDMBuffer** calls. Recall that in the O2 buffering API, buffers are sent to the application as events. **vlEventRecv** gets the next path event:

```
int vlEventRecv(VLServer svr, VLPath path, VLEvent *event)
```

where

| | |
|---|---|
| *svr* | specifies the server the client is using |
| *path* | specifies the path on which data is received from |
| *event* | points to a location where the event information is stored. |

If the received event is a VLTransferComplete event and the path captures video data (e.g. a video-to-memory path), then **vlEventToDMbuffer** will retrieve the DMbuffer from the event:

```
int vlEventToDMBuffer(VLEvent *event, DMbuffer *buffer)
```

where

| | |
|---|---|
| *event* | points to a VLTransferComplete event |
| *buffer* | points to a location where the buffer handle will be stored. |

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

### Sending DMbuffers to the Video Library

Applications use **vlDMBufferSend** to send buffers to a video device:

```
int vlDMBufferSend(VLServer svr, VLPath path, DMbuffer buffer)
```

where

| | |
|---|---|
| *svr* | specifies the server to which the path is attached |
| *path* | specifies the path on which video is sent |
| *DMbuffer* | specifies the buffer to send |

The DMbuffer may have been obtained from another library, such as dmIC, or generated by the application itself. The **dmBufferAllocate** function can be used to allocate a buffer from a DMbufferPool.

### Freeing a DMbuffer

Once the application is done with a buffer, it should call **dmBufferFree** to indicate that it no longer intends to use the buffer. Note that this does not free the memory from the buffer pool - it only indicates to the buffering mechanism that the application is done using the buffer. Only after all users of a buffer have called **dmBufferFree** can the buffer be reallocated. The Video Library never implicitly releases the application's access to a buffer. Consequently, an application can send the same buffer to a memory node multiple times, or hold a captured image for an indefinite period.

### Cleaning up

**vlDMPoolDeregister** disassociates a DMbufferpool from a memory node. It should be called to cleanup the memory node or allow a new DMbufferpool to be used after a transfer has been stopped.

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

## Developing Cross-Platform Applications

With the all-platforms release following IRIX 6.4, the cross-platform VL DMbuffer API will be available on O2, Octane, Onyx2, and Indigo2 IMPACT video devices[1]. In the interim, executables that run on O2, Octane, and Onyx2 must use both interfaces. The section discusses issues in developing such applications.

In general, the steps required to capture or output video are the same for the O2 VL DMbuffer API and the cross-platform API. By careful coding and evaluation of return codes, it is possible to write code which will execute on all SGI platforms supporting DMbuffer. Under IRIX 6.3, the cross-platform API functions will return an error when run on an O2. Similarly, O2-specific API functions will cause an error to returned on non-O2 systems  In both cases, vlErrno will be set to VLAPIConflict. If this error return is encountered, the function for the other API should be called, as illustrated in the following code fragments. Note that these fragments demonstrate one possible method of writing cross-platform code, and may not be the most applicable to a particular application.

1. With the exception of Indigo2 Video for Indigo2 IMPACT. Indigo2 IMPACT Video for Indigo2 IMPACT and Indigo2 IMPACT Compression for Indigo2 IMPACT will support the cross-platform API.

## Required Software

In order to develop an application that will use both the O2 and cross-platform VL DMbuffer APIs, the following software environment are required.

On O2:

- Digital Media Development Environment, 6.3
- Digital Media Execution Environment, 6.3
- Patch 1882 (O2 Video for 6.3 w/R10k release), or later

On Octane or Onyx2:

- Digital Media Development Environment, 6.4
- Digital Media Execution Environment, 6.4
- Patch 1765 (dmedia_eoe and dmedia_dev patch for 6.4), or later
- Video option product development and execution software

Target systems should have installed the Digital Media Execution Environments and patches listed above.

## Obtaining the pool requirements

```
if (vlDMGetParams(svr, path, node, poolParams) < 0) {
    if (vlErrno==VLAPIConflict) {
        if (vlDMPoolGetParams(svr, path, node, poolParams) < 0) {
            /* Error */
        }
    }
    else {
        /* Error */
    }
}
```

## Buffer Handler for captured video

This section processes a buffer once it has been received. Remember to free the buffer when done with it.

```
/*
 * Call dmBufferMapData if a pointer to the buffer contents are required,
 * or simply send the buffer to another library such as dmIC
 */
dmBufferFree(DMbuffer)
```

## Sending video data to a device

This section sends a DMbuffer to a video device. Try the cross-platform function first, then the O2-specific equivalent.

```
if (vlDMBufferPutValid(svr, path, node, DMbuffer) < 0) {
    if (vlErrno == VLAPIConflict) {
        if (vlDMBufferSend(svr, path, DMbuffer) < 0) {
            /* Error */
        }
    }
    else {
```

```
            /* Error */
        }
    }
```

### Receiving a DMbuffer (cross-platform routine)

This section receives buffers if the cross-platform API is used. In the O2 case, buffer receipt is handled when the path events are received (see below).

```
if (vlDMBufferGetValid(svr, path, node, &DMbuffer) < 0) {
    if (vlErrno == VLAgain) {
        /* Ok, no data available now */
    }
    else {
        /* Error */
    }
}

/* Call buffer handler for captured video */
```

### Get Events

This section gets a server based event and dispatches it. For the O2 API, if we receive a VLTransferComplete event then we also extract the DMbuffer and send it to the buffer handler.

```
if (vlCheckEvent(svr, VLAllEventsMask, &event) < 0) {
    if (vlErrno == VLAPIConflict) {
        if (vlEventRecv(svr, path, &event) < 0) {
            if (vlErrno == VLAgain) {
                /* No event ready, just return */
                return;
            }
            else {
                /* Error */
            }
        }
        if (event.reason == VLTransferComplete) {
            if (vlEventToDMBuffer(&event, &DMbuffer) < 0) {
                /* Error */
            }
        }
        /* Call buffer handler */
    }
    else {
        /* No event ready, just return */
        return;
    }
}

/* Handle event */
```

### Main loop

This section illustrates a main loop which waits for server events and buffer availability (video-to-memory application). Note that for the cross-platform API we wait on the file descriptor returned by **vlNodeGetFd**, and in the O2 API wait on the connection fd for a VLTransferComplete event. This results in minimal overhead for the respective APIs.

For a video output main loop, the application would wait for server events and a file descriptor associated with the DMbufferpool that becomes ready for reading when a buffer can be allocated.

```
/*
 * Get fds for the server and buffer.
 */
eventFD = vlGetFD(svr);
if (eventFD < 0) {
    /* Error */
}
bufferFD = vlNodeGetFd(svr, path, node);
if (bufferFD < 0 && vlErrno != VLAPIConflict) {
    /* Error */
}

maxFD = max(bufferFD, eventFD);

/*
 * We must register for TransferComplete events with the O2 API, and
 * optionally for the cross-platform API.
 */
vlSelectEvents(svr, path, VLTransferCompleteMask |
    VLTransferFailedMask | ...);

/*
 * Standard main loop for waiting for buffers and events
 */
FD_ZERO(readFDs);
FD_ZERO(writeFDs);
FD_ZERO(exceptFDs);
while (1) {
    if (bufferFD > 0) FD_SET(readFDs, bufferFD);
    FD_SET(readFDs, eventFD);
    select(maxFD + 1, readFDs, writeFDs, exceptFDs);

    if (bufferFD > 0 && FD_ISSET(bufferFD, readFDs)) {
        /* Receive a DMbuffer, cross-platform routine*/
    }

    if (FD_ISSET(eventFD, readFDs)) {
        /* Receive an event */
    }
}
```

# Sample Code

Sample code accompanying this paper demonstrates video capture and playback using VL Buffers, O2 VL DMbuffers, and cross-platform VL DMbuffers. An additional pair of examples demonstrates combining O2 and cross-platform API flavors in a single executable.

`memtovid_dmbuffer_xplatform.c`, `vidtomem_dmbuffer_xplatform.c`

> Video capture or playback using the cross-platform VL DMbuffer API. This code will execute on Octane or Onyx2 video devices, and on IMPACT and O2 with the next all-platforms release.

`memtovid_dmbuffer_O2.c`, `vidtomem_dmbuffer_O2.c`

> Video capture or playback using the O2 VL DMbuffer API. This code will execute on an O2 only.

`memtovid_vlbuffer.c`, `vidtomem_vlbuffer.c`

> Video capture or playback using VL Buffers. This code will execute on all SGI video devices.

`memtovid_dmbuffer_xplatform_and_O2.c`, `vidtomem_dmbuffer_xplatform_and_O2.c`

> Video capture or playback using both the O2 and cross-platform VL DMbuffer APIs. This code will execute on O2, Octane, and Onyx2 video devices, and additionally on IMPACT Video and IMPACT Compression with the next all-platforms release.

# Summary

In the short term, until the next major all platforms release, the cross-platform VL DMbuffer API is available on Octane Digital Video, Octane Compression, and DIVO for Onyx2 and Origin 2000 only. Furthermore, the O2 DMbuffer API remains available only with O2 Video. Application developers wishing to develop code that runs on multiple platforms may choose to use VL Buffers, which are available on all VL devices, or to write code using both VL DMbuffer APIs.

For longer term development, the cross-platform VL DMbuffer API will be available on all O2, Octane, Onyx2, and Origin 2000 video devices in the next all-platforms release. It is recommended that developers use this API in favor of VL Buffers. DMbuffers provide an exchange mechanism with other digital media APIs, such as dmIC, and are generally more flexible than VL Buffers. While VL Buffers will continue to be supported within the VL, DMbuffers represents the future of SGI digital media buffering.