Digital Media Software Development Kit Programmer's Guide

007–4280–001

CONTRIBUTORS

Written by Bob Bernard and Tammy Domeier

Illustrated by Chris Wengelski

Edited by Rick Thompson

Production by Diane Ciardelli

Engineering contributions by Frank Bernard, Beryl Chen, Jeff Hane, Jaya Kanajan, Derek Millar, Michael Pruett, Mike Travis, and Ke Wu

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

Record of Revision

Version	Description
001	May 2001 Supports the 2.0 release of the Digital Media Software Development Kit (dmSDK).

Contents

Figures

Tables

About This Guide

This document provides an introduction to the SGI Digital Media Software Development Kit (dmSDK). The dmSDK provides a cross-platform library for controlling digital media hardware. It supports audio and video I/O devices and transcoders.

This document is a general user's guide, for a more detailed treatment of a particular function, see the online reference pages for dmSDK.

Obtaining Publications

To obtain SGI documentation, go to the SGI Technical Publications Library at:

`http://techpubs.sgi.com`

Conventions

The following conventions are used throughout this document:

`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

manpage(x)	Man page section identifiers appear in parentheses after man page names.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact us in any of the following ways:

- Send e-mail to the following address:

  `techpubs@sgi.com`

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

  `http://techpubs.sgi.com`

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

- Send mail to the following address:

  Technical Publications
  Silicon Graphics, Inc.
  1600 Amphitheatre Pkwy.
  Mountain View, California 94043–1351

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

  We value your comments and will respond to them promptly.

Introduction

This chapter is a quick introduction to the Digital Media Software Development Kit (henceforth, the *dmSDK*). It includes a table of terms, followed by an example audio output program.

To get started with the dmSDK, you should read this chapter, then browse the online example programs. For an in-depth treatment, consult later chapters as you experiment with your own programs.

Note: The material in this chapter assumes that the dmSDK is installed on your workstation, and that you have access to the dmSDK example programs.

Terms

These terms are used throughout this document, and some are used in the dmSDK code. Read these first to avoid any confusion.

Term	Definition
graphics / video	In dmSDK, *graphics* and *video* are not synonymous: "graphics" indicates the graphical display used for the user-interface on a computer; "video" indicates the type of signal sent to a video cassette recorder, or received from a camcorder.
capability tree	A capability tree is the hierarchy of all DM devices in the system, and contains information about each DM device. An application may search a capability tree to find suitable media devices for operations you wish to perform.
system	The highest level in the capability tree hierarchy. It is the machine on which your application is running. This machine is given the name `DM_SYSTEM_LOCALHOST`. Each system contains one or more devices.
physical device	A device that corresponds to device-dependent modules in the dmSDK. Typically, each

	device-dependent module supports a set of software transcoders, or a single piece of hardware. Examples of devices are audio cards on a PCI bus, DV camcorders on the 1394 bus, or software DV modules. Each device-dependent module may expose a number of *logical devices*: jacks, paths, or transcoders.
jack	A logical device that is an interface in/out of the system. Examples of jacks are composite video connectors and microphones. Jacks often, but not necessarily, correspond to a physical connector — in fact, it is possible for a single dmSDK jack to refer to several such connectors. It is also possible for a single physical connector to appear as several logical jacks.
path	A logical device that provides logical connections between memory and jacks. For example, a video output path transports data from buffers to a video output jack. Paths are logical entities. Depending on the device, it is possible for more than one instance of a path to be open and in use at the same time.
transcoder	A transcoder is a logical device that takes data from buffers via an input pipe or pipes, performs an operation on the data, and returns the data to another buffer via an output pipe. The connections from memory to the transcoder, and from the transcoder to memory, are called *pipes*. Example transcoders are DV compression, or JPEG decompression.
UST	Unadjusted System Time. UST is a special system clock which runs continuously without adjustment. This clock is used to synchronize media streams.

MSC — Media Stream Count. MSC is a measure of the number of media samples which have passed though a jack. This is useful to synchronize media streams.

Getting Started with the dmSDK

Before you begin, you should examine your system with the `dmquery`(1dm) tool. This tool prints a list of all supported dmSDK devices on the system. Here is an example `dmquery` on the system *Linux i386 Workstation*:

Example 1-1 `dmquery` Printout

```
% dmquery
SYSTEM: Linux i386 Workstation
Devices:
   Software DV_MMX Codec [0]
   OSS audio device [0]
```

This printout indicates that there are two installed devices: a software DV transcoder, and an audio I/O device (which in this case, is built using the Linux OSS driver). Other options to `dmquery` allow you to gather more information about the installed devices; but for now, just knowing their names will suffice.

See the `dmquery`(1dm) man page for more information.

Simple Audio Output Program

This example program outputs a short beep. To keep it simple, a few details, primarily error-checking, are skipped. This program only includes the operations required to produce the beep.

Note: Consult the online example code for more advanced programs.

Step 1: Include the `dmsdk.h` and `dmutil.h` Files

To begin, you will need the `dmsdk.h` and `dmutil.h` files. The dmSDK library provides the core functionality, and the dmUtil library provides some convenient

utility functions built on that core. As an application developer, you may choose to use only the core, or you may find it convenient to utilize the simpler utility functions.

Include the files as follows:

```
#include <dmsdk.h>
#include <dmutil.h>
```

Step 2: Locate a Device

You must query the capabilities of the system to find a suitable digital media device with which to perform your audio output task. To do that, you must search the dmSDK capability tree, which contains information on every dmSDK device on the system.

In your search, you should start at the top of the tree as follows:

1. Query the local system to find the first physical device that matches your desired device name.
2. Look in that device to find its first output jack.
3. Find an output path that goes through that jack.

In this case, assuming that the device name is being passed in as a command-line argument, use some of the utility functions to find a suitable output path:

```
DMint64 devId=0;
DMint64 jackId=0;
DMint64 pathId=0;

dmuFindDeviceByName( DM_SYSTEM_LOCALHOST, argv[1], &devId );
dmuFindFirstOutputJack( devId, &jackId );
dmuFindPathToJack( jackId, &pathId );
```

Step 3: Open the Device Output Path

An open device output path provides your application with a dedicated connection to the hardware, and it allocates system resources for use in subsequent operations. The device path is opened with an `open` call as follows:

```
dmOpen( pathId, NULL, &openPath );
```

If the `open` call is successful, you will get an open path identifier. All operations using that path must use its identifier.

Note: Sometimes an `open` call can fail due to insufficient resources (typically because too many applications may already be using the same physical device).

Step 4: Set Up the Audio Device Path

Now you set up the path you just opened for your operation. In this case you will use signed 16-bit audio samples, with:

- A single (mono) audio channel
- A gain of -12dB
- A sample rate of 44.1kHz

To make those settings, you must construct a controls message to describe them. The controls message is a list of `param/value` (DMpv) pairs, where the last entry in the list is `DM_END`.

```
dmpv controls[5];
DMreal64 gain = -12; // decibels

controls[0].param = DM_AUDIO_FORMAT_INT32;
controls[0].value.int32 = DM_FORMAT_S16;
controls[1].param = DM_AUDIO_CHANNELS_INT32;
controls[1].value.int32 = 1;
controls[2].param = DM_AUDIO_GAINS_REAL64_ARRAY;
controls[2].value.pReal64 = &gain
controls[2].length = 1;
controls[3].param = DM_AUDIO_SAMPLE_RATE_REAL64;
controls[3].value.real64 = 44100.0;
controls[4].param = DM_END;
```

Notice that this message contains both scalar parameters (for example, the number of audio channels) and an array parameter (the array of audio gains).

Step 5: Set Controls on Audio Device Path

After the controls message has been constructed, you must set the controls on the open audio path as follows:

```
dmSetControls(openPath, controls);
```

This call makes all the desired control settings and does not return until those settings have been sent to the hardware. If it returns successfully, it indicates that all of the control changes have been committed to the device (and you are free to delete or alter the controls message).

Note: All control changes within a single controls message are processed atomically. So, either the call succeeds, and they are all applied, or the call fails, and none are applied.

Assuming that the call succeeded, the path is now set up and ready to receive audio data.

Step 6: Send Buffer to Device for Processing

This example assumes that you have already allocated a buffer in memory and filled it with audio samples. To send that buffer to the device for processing, you must first construct a buffers message that describes it. That message includes both a pointer to the buffer and the length of the buffer (in bytes):

```
DMpv msg[2];
 msg[0].param = DM_AUDIO_BUFFER_POINTER;
 msg[0].value.pByte = ourAudioBuffer;
 msg[0].length = sizeof(ourAudioBuffer);
 msg[1].param = DM_END;
```

Then, send the buffers message to the opened path:

```
dmSendBuffers(openPath, msg);
```

When the message is sent, it is placed on a queue of messages going to the device. The `send` call does very little work: it gives the message a cursory look before sending it to the device for later processing.

Note: Unlike the `set` call, the `send` call does not wait for the device to process the message, it simply enqueues it and then returns.

Step 7: Begin Message Processing

You must tell the device to start processing enqueued messages. This is done with the begin transfer call as follows:

```
dmBeginTransfer(openPath);
```

You can sleep while the device is busy working on the message as follows:

```
sleep(5)
```

This is not the best way to approach this, but it is the simplest. (It will be changed in the next example.)

Step 8: Receive the Reply Message

As the device processes each message, it generates a reply message which is sent back to our application. By examining that reply we can confirm that the buffer was transferred successfully, as follows:

```
DMint32 messageType;
DMpv* message;

dmReceiveMessage(openPath, &messageType, &Message );

if( messageType == DM_BUFFERS_COMPLETE )
  printf("Buffer transferred!\n");
```

Step 9: Close the Path

Once you have verified that the buffer transferred successfully, you can close the path as follows:

```
dmClose(openPath);
```

Closing the path ends active transfer and frees any resources allocated when the path was opened.

Realistic Audio Output Program

The preceding procedure was for a single audio buffer. In this example, you will process millions of audio samples.

Step 1: Open the Device Output Path

Open the device output path just as in the previous example:

```
dmOpen( pathId, NULL, &openPath );
```

Opening the path also allocates memory for the message queues used to communicate with the device. One of those queues will hold messages sent from our application to the device, and one will hold replies sent from the device back to our application.

Step 2: Allocate Buffers

If you were only processing a short sound, you could preallocate space for the entire sound and perform the operation straight from memory. However, for a more general and efficient solution, you need to allocate space for a small number of buffers, and reuse each buffer many times to complete the whole transfer.

Here, assume that memory has been allocated for twelve audio buffers, and that those buffers have been filled with the first few seconds of audio data to be output.

Step 3: Send Buffers to the Open Path

Now send each of the twelve buffers to the open path. Here the queue of messages between application and device becomes more interesting. The dmSDK enables you to enqueue all the buffers without the device having even looked at the first one as follows:

```
int i;
for(i=0; i<12; i++)
  {
   DMpv msg[3];
```

```
 msg[0].param = DM_IMAGE_BUFFER_POINTER;
 msg[0].value.pByte = (DMbyte*)buffers[i];
 msg[0].maxLength = imageSize;
 msg[1].param = DM_AUDIO_UST_INT64;
 msg[1].param = DM_END;
 dmSendBuffers(openPath, msg);
}
```

Notice that each audio buffer is sent in its own message, this is because each message is processed atomically, and refers to a single instant in time. In addition to the audio buffer, this message also contains space for an audio Unadjusted System Time (UST) time stamp. That time stamp will be filled in as the device processes each message. It will indicate the time at which the first audio sample in each buffer passed out of the machine.

Step 4: Begin the Transfer

Now you can tell the device to begin the transfer. It reads messages from its input queue, interprets the buffer parameters within them, and processes those buffers with the following:

```
dmBeginTransfer(openPath);
```

At this point, you can sleep as the device processes the buffers. However, a more efficient approach is to select the file descriptor for the queue of messages sent from the device back to your application. In dmSDK terminology, that file descriptor is called a *wait handle* on the receive queue:

```
DMwaitable pathWaitHandle;
 dmGetReceiveWaitHandle(openPath, &pathWaitHandle);
```

Having obtained the wait handle, you can wait for it to fire by using `select` on IRIX/Linux, or `WaitForSingleObject` on Windows as follows:

On IRIX/Linux:

```
fd_set fdset;
 FD_ZERO( &fdset);
 FD_SET( pathWaitHandle, &fdset);

 select( pathWaitHandle+1, &fdset, NULL, NULL, NULL );
```

On Windows:

```
WaitForSingleObject( pathWaitHandle, INFINITE );
```

Step 5: Receive Replies from the Device

Once the `select` call fires, a reply will be waiting. Retrieve the reply from the receive queue as follows:

```
DMint32 messageType;
 DMpv* replyMessage;

 dmReceiveMessage(openPath, &messageType, &replyMessage );

 if( messageType == DM_BUFFERS_COMPLETE )
   printf("Buffer received!\n");
```

This reply has the same format and content as the buffers message that was originally enqueued, plus any blanks in the original message will have been filled in. In this case, the reply message includes the location of the audio buffer that was transferred, as well as a UST time stamp indicating when its contents started to flow out of the machine:

```
DMbyte* audioBuffer = replyMessage[0].value.pByte;
 DMint64 audioUST    = replyMessage[1].value.int64;
```

Note: The UST time stamp is useful to synchronize several different media streams (for example, to make sure the sounds and pictures of a movie match up).

Step 6: Refill the Buffer for Further Processing

At this point you can refill the buffer with more audio data, and send it back to the device to be processed again with the following:

```
dmSendBuffers(openPath, replyMessage);
```

In this case you are making a small optimization, so rather than construct a whole new buffers message, simply reuse the reply to your original message.

At this point you have processed the reply to one buffer. If you wish, you can now go back to the `select` call and wait for another reply from the device. This can be repeated indefinitely.

Step 7: End the Transfer

Once enough buffers have been transferred, you can end the transfer as follows:

```
dmEndTransfer(openPath);
```

In addition to ending the transfer, this call performs the following:

- Flushes the queue to the device.
- Aborts any remaining unprocessed messages.
- Returns any replies on the receive queue to the application.

The `endTransfer` call is a blocking call. When it returns, the queue to the device will be empty, the device will be idle, and the queue from the device to your application will contain any remaining replies.

If you wish, at this point, you can send more buffers to the path (see "Step 3: Send Buffers to the Open Path", page 8).

Step 8: Close the Path

Use the following to close the path:

```
dmClose(openPath);
```

Note: This chapter has provided only a quick introduction to an audio output device. Through a similar interface, the dmSDK also supports audio input, video input, video output, and memory-to-memory transcoding operations.

Audio/Video Jacks

The Digital Media Library is concerned with three types of interfaces: jacks for control of external adjustments, paths for audio and video through jacks in/out of the machine and pipes to/from transcoders. All share common control, buffer, and queueing mechanisms. In this section these mechanisms are described in the context

of operating on a jack and its associated path. In subsequent sections, the application of these mechanisms to transcoders and pipes is discussed.

Opening a Jack

Before setting controls to a jack, a connection must be opened. This is done by calling dmOpen.

```
DMstatus dmOpen(const DMint64 objectId, DMpv* options, DMopenid* openId);
```

A jack is usually an external connection point and most often one end of a path. Jacks may be shared by many paths or they may have other exclusivity inherent in the hardware. For example, a common video decoder may have a multiplexed input shared between composite and S-video. If only one can be in use at a given instance, then there is an implied exclusiveness between them. Many jacks do not support an input message queue since an application cannot send data to a jack (it must be sent via a path). Therefore, the dmSendControls and dmSendBuffers are not supported on a jack, so that dmSetControls must be used to adjust controls. Typically, the adjustments on a path affect hardware registers and can be changed while a data transfer is ongoing (on a path that connects the jack to memory). Examples are brightness and contrast. Some controls are not adjustable during a data transfer. For example, the timing of a jack cannot usually be changed while a data transfer is in effect. Reply messages may be sent by jacks and usually indicate some external condition, such as sync lost or gained.

Constructing a Message

Messages are arrays of parameters, where the last parameter is always `DM_END`. For example, the flicker and notch filters can be adjusted with a message such as the following:

```
DMpv message[3];
message[0].param = DM_VIDEO_FLICKER_FILTER_INT32;
message[0].value.int32 = 1;
message[1].param = DM_VIDEO_NOTCH_FILTER_INT32;
message[1].value.int32 = 1;
message[2].param = DM_END
```

Setting Jack Controls

Since jack controls deal with external conditions and not processing associated with data transfers, applications use `dmSetControls` or `dmGetControls` calls to manipulate these controls. Here is an example of how the genlock vertical and horizontal phase can be obtained immediately:

```
DMpv message[3];
message[0].param = DM_VIDEO_H_PHASE_INT32;
message[1].param = DM_VIDEO_V_PHASE_INT32;
message[2].param = DM_END;
if( dmGetControls( aJackConnection, message))   handleError();
 else
   printf("Horizontal offset is %d, Vertical offset is %d\n",
      message[0].value.int32, message[1].value.int32);
```

`dmSetControls` and `dmGetControls` are blocking calls. If the call succeeds, the message has been successfully processed. Note that not all controls may be set via `dmSetControls`. The access privilege in the param capabilities can be used to verify when and how controls can be modified.

Closing a Jack

When an application has finished using a jack it may close it with dmClose:

```
DMstatus dmClose(DMopenid openId);
```

All controls previously set by this application normally remain in effect though they may be modified by other applications.

Parameters

This chapter describes the dmSDK parameter syntax and semantics. These parameters define a number of variables including control values such as the frame rate or image width, and the location of data such as a single video field.

`param/value` Pairs

The fundamental building block of the dmSDK is the `param/value` pair (`DMpv`), as shown here:

```
typedef struct {
    DMint64 param;
    DMvalue value;
    DMint32 length;
    DMint32 maxLength;
} DMpv;
```

The `param` is a unique numeric identifier for each parameter; and the `value` is a `union` of several possible types, of which the most common are:

```
typedef union {
    DMint32   int32;   /* 32-bit signed integer values */
    DMint64   int64;   /* 64-bit signed integer values */
    DMbyte*   pByte;   /* pointer to an array of bytes */
    DMreal32* real32;  /*32-bit floating point value */
    DMreal64* real64;  /*64-bit floating point value */
    DMint32*  pInt32;  /*pointer to an array of 32-bit signed integer values */
    DMint64*  pInt64;  /*pointer to an array of 64-bit signed integer values */
    DMreal32* pReal32; /*pointer to an array of 32-bit floating point values */
    DMreal64* pReal64; /*pointer to an array of 64-bit floating point values */
    struct_DMpv*pPv;   /*pointer to a message of param/value pairs*/
    struct_DMpv** ppPv;/*pointer to an array of messages */}DMvalue;
```

Messages

In the dmSDK, applications communicate with devices using messages. Each message is a simple array of `param/value` pairs; where the last `param` in the message is DM_END.

For example, the following is a message that sets image width to 1920 and image height to 1080:

```
DMpv controls[3];
controls[0].param = DM_IMAGE_WIDTH_INT32;
controls[0].value.int32 = 1920;
controls[1].param = DM_IMAGE_HEIGHT_INT32;
controls[1].value.int32 = 1080;
controls[2].param = DM_END;
```

Note: A `DMpv` ends with the `DM_END` parameter to indicate completion.

Scalar Values

This section shows you how to set and get scalar values.

Set Scalar Values

To set the values of scalar parameters, you must enter the `param` and `value` fields of each `DMpv` and send the result to a device. If the value is valid, the returned length will be `1`. If the value is invalid, or if the parameter is not recognized by the device, an error status will be returned and length will be set to `-1`.

Note: You do not need to set the `length` or `maxLength` fields — they are ignored when setting scalars. However, on return (`dmReceiveMessage`) a length parameter that equals -1 indicates that this parameter was in error.

For example, to set video timing:

```
DMpv message[2];
message[0].param = DM_VIDEO_TIMING_INT32;
message[0].value.int32 = DM_TIMING_525;
message[1].param = DM_END;
```

```
if( dmSetControls( someOpenVideoPath, message) )
  fprintf(stderr, "Error, unable to set timing\n");
```

Get Scalar Values

To get scalar values, you again construct a `DMpv` list, but here you do not need to set the `value` field. As the device processes the DMpv list, it fills in the `value` and `length` fields. If the `value` is valid, the returned `length` is `1`. If the `value` is invalid, or the parameter is not recognized by the device, an error status will be returned, and `length` is set to `-1`.

For example, to get video timing:

```
DMpv message[2];
message[0].param = DM_VIDEO_TIMING_INT32;
message[1].param = DM_END;
dmGetControls( someOpenVideoPath, message);
if( message[0].length == 1 )
  printf("Timing is %d\n", message[0].value.int32);
else
  fprintf(stderr, "Unable to determine timing\n");
```

Array Values

An array in the dmSDK is much like an array in C:

- `value` of the `DMpv` is a pointer to the first element of the array
- `length` is the number of valid elements in the array
- `maxLength` is the total length of the array

Also, each element increases the length of the array by `1`, so an array of four 32-bit integers would require a `maxLength` of four.

Set the Value of an Array Parameter

To set the value of an array parameter, fill out the `param`, `value`, `length` and `maxLength` fields. If the values are valid, the returned length will be unaltered. If

the values are invalid or if the parameter is not recognized at all by the device, an error status will be returned and length will be set to `-1`.

For example:

```
DMreal64 data[] = { 0, 0.2, 0.4, 0.6, 1.0};
DMpv message[2];
message[0].param = DM_PATH_LUT_REAL64_ARRAY;
message[0].value.pReal64 = data;
message[0].length = sizeof(data)sizeof(DMreal64);
message[1].param = DM_END;
dmSetControls( someOpenPath, message )
```

Note: You do not need to set the `maxLength` field — it is ignored when setting an array parameter.

In the preceding example, you are free to modify the data array at any time before calling `dmSetControls`; and you regain that right as soon as `dmSetControls` returns.

If you have a multithreaded application, your application must ensure the data array is not accessed by some other thread while the `SetControls` call is in progress.

Get the Size of an Array Parameter

To get the size of an array parameter, set `maxLength` to `0`. The device will fill in `maxLength` to indicate the minimal array size to hold that value. If the parameter is not recognized by the device, an error status will be returned, `maxLength` will be set to `0`, and `length` will be set to `-1`.

```
DMpv message[2];
message[0].param = DM_PATH_LUT_REAL64_ARRAY;
message[0].length = 0;
message[0].maxLength = 0;
message[1].param = DM_END;
dmGetControls( someOpenPath, message );
printf("Size of LUT is %d\n", message[0].maxLength);
```

Get the Value of an Array Parameter

To get the value of an array parameter, create an array with `maxLength` entries to hold the result, and set `length` to `0`. The device will fill in no more than `maxLength` array elements and set `length` to indicate the number of valid entries. If the values are invalid or if the parameter is not recognized at all by the device, an error status will be returned and `length` will be set to `-1`.

```
DMint32 data[10];
DMpv message[2];
message[0].param = DM_PATH_LUT_INT32_ARRAY;
message[0].value.pInt32 = data;
message[0].length = 0;
message[0].maxLength = 10;
message[1].param = DM_END;
dmGetControls( someOpenPath, message );
if( message[0].length > 0 )
{
  printf("Received %d array entries\n", message[0].length);
  printf("The first entry is %d\n", data[0]);
}
```

Note: Your application controls memory allocation. If you want to get the whole array, but do not know the maximum size, you must query for `maxLength` first, allocate space for the result, and then query for the value.

Pointer Values

The distinction between array values and pointer values in the dmSDK is subtle, but important. Array values are copied when they are passed to or received from a device. Thus, your application owns the array memory and is nearly always free to modify or free it.

A pointer parameter is a special type of array parameter that is used to send and receive data buffers (as arrays of bytes.) Pointer values are not copied. Instead, only the location of the data is passed to the device. The application sends a buffer by calling `dmSendBuffer`. `dmSendBuffer` places the controls and buffer pointer in the data payload area and inserts a header on the send queue for the device.

This is much more efficient, but it imposes a restriction: after a pointer value is given to a device, that memory cannot be touched until the device has finished processing it.

Note: For efficient processing, all buffers must be pinned in memory.

For example, the following code fragment shows how a pointer parameter might be initialized to send an image to a video input path:

```
DMpv message[2];
message[0].param = DM_IMAGE_BUFFER_POINTER;
message[0].value.pByte = someBuffer;
message[0].maxLength = sizeof(someBuffer);
message[1].param = DM_END;
if( dmSendBuffers( someOpenPath, message ) )
  fprintf(stderr, "Error sending buffers\n");
```

The above `SendBuffers` call places the message on a queue to be processed by the device, and then returns. It does not wait for the device to finish with the buffer. Thus, even after the call to `SendBuffers`, the device still owns the image buffer. Your application must not touch that memory until it is notified that processing is complete.

When you send a buffer to be filled, the device uses `maxLength` to determine how much it may write, and it returns `length` set to indicate the amount of the buffer it actually used.

When you send a buffer for output, the device will interpret the length as the maximum number of bytes of valid data in the buffer. In this case `maxLength` is ignored.

Capabilities

This chapter describes the dmSDK capabilities tree, the repository of information on all installed dmSDK devices. The capabilities tree tells you everything from the hardware location of a physical device, to the range of legal values for supported parameters.

The Capabilities Tree

The capabilities tree forms a hierarchy that describes the installed dmSDK devices in the following order from top to bottom:

1. Physical system
2. Physical devices
3. Logical devices
4. Supported parameters on the logical devices

See "Terms", page 1 for definitions of the elements of the capabilities tree hierarchy.

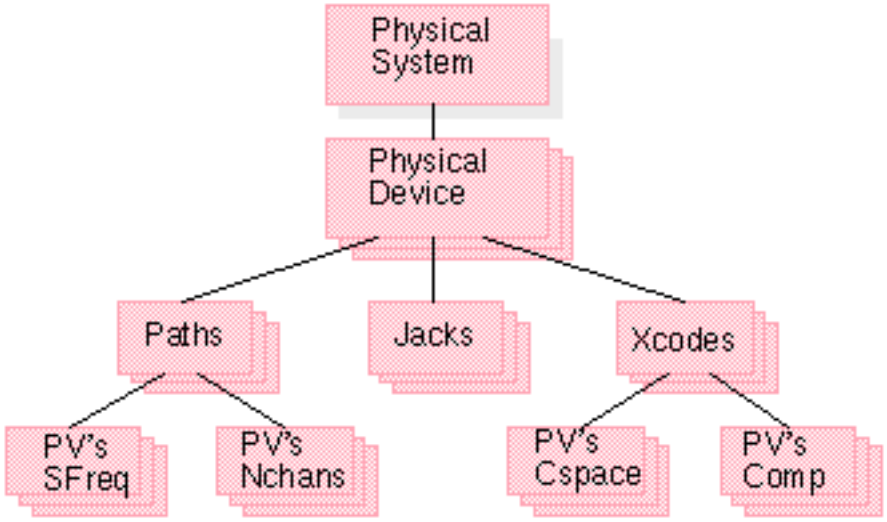

Figure 3-1 The Capabilities Tree

Utility Functions for Capabilities

To access the capability hierarchy, you may either search the capability tree directly, or make use of convenient utility functions to perform the search for you. This section discusses the baseline functionality provided in the core dmSDK library, but you may also wish to examine the utility library and example code for pre-written alternatives.

Manual Access to Capabilities

Direct access to the dmSDK capabilities tree is via three functions:

Function Call	Description
`dmGetCapabilities`	calls the capabilities for a dmSDK object
`dmPvGetCapabilities`	calls the capabilities for a parameter on a given device

`dmFreeCapabilities`	releases a set of capabilities when you have finished using them

Accessing Capabilities

The following code examples show you how to query for the capabilities of your entire capability tree. See "Terms", page 1 for a definition of terms used here.

Note: All objects in the dmSDK are referred to via 64-bit identifying numbers. For example, the 64-bit id number for the system on which your application is running is `DM_SYSTEM_LOCALHOST`.

---

1. You can get the capabilities of the local system as follows. This will give you a DMpv list that includes an array of identifiers for all the physical devices installed on this system:

   Example 3-1 Get System Capabilities

   ```
   DMpv* systemCap;
   dmGetCapabilities( DM_SYSTEM_LOCALHOST, &systemCap);
   ```

2. Use the following to list the number of physical devices on your system:

   Example 3-2 Get Physical Devices

   ```
   DMpv* deviceIds = dmPvFind( systemCap, DM_SYSTEM_DEVICE_IDS);
   printf("There are %d physical devices\n", deviceIds->length );
   if( deviceIds->length > 0 )
     printf("The first device has id %llx\n", deviceIds->value.pInt64[0]);
   dmFreeCapabilities( systemCap );
   ```

3. The following is example code to examine a physical device for its supported I/O paths and transcoders (that is, its logical devices):

   Example 3-3 Get Logical Devices

```
DMpv* deviceCap, *pathIds, *xcodeIds;
dmGetCapabilities( someDeviceId, &deviceCap);
pathIds = dmPvFind( deviceCap, DM_DEVICE_PATH_IDS);
xcodeIds = dmPvFind( deviceCap, DM_DEVICE_XCODE_IDS);
printf("Device supports %d i/o paths and %d transcoders\n",
pathIds->length, xcodeIds->length);
```

```
if ( pathIds->length > 0 )
  printf("The first i/o path has id %llx\n", pathIds->value.pInt64[0]);
dmFreeCapabilities( deviceCap );
```

4. Descending still further down the capability tree, you can obtain the capabilities of any particular logical device by again calling `dmGetCapabilities`. For example, here you find how many parameters are accepted by a path:

Example 3-4 Get Parameters Accepted by a Path

```
DMpv* pathCap, *paramIds;
dmGetCapabilities( somePathId, &pathCap);
paramIds = dmPvFind( pathCap, DM_PARAM_IDS);
printf("Path supports %d parameters\n", paramIds->length);
if (paramIds->length > 0)
  printf("The first parameter has id %llx\n",paramIds->value.pInt64[0]);
dmFreeCapabilities( pathCap );
```

Query Individual Parameters of Logical Devices

At this point, you have descended from the system to the logical device. Still there is one more level: the parameter. Querying the capabilities of a parameter is subtly different because the interpretation of parameters in the dmSDK is device-dependent (for example, the legal values for `DM_IMAGE_WIDTH_INT32` may be `1920` on one device and `720` on another). Thus, you must pass both a logical device ID and a parameter ID as follows:

Example 3-5 Get Capabilities of a Parameter

```
DMpv* paramCap, *paramName;
dmGetCapabilities( someLogicalDeviceId, someParamId, &paramCap );
paramName = dmPvFind( paramCap, DM_NAME_BYTE_ARRAY );
if( paramName != NULL )
    printf("Param has name %s\n", (char *) ( paramName->value.pByte ));
dmFreeCapabilities( paramCap );
```

Note: Since the name of the parameter is being queried on a particular device, the above code will work for all parameters. This includes new device-dependent parameters.

Also, see the `dmPvToString` reference page for a simpler way to find a parameter name.

Query Parameters Which Describe Parameters

In addition to obtaining the capabilities of device parameters, you may also obtain the capabilities of the parameters used to describe capabilities themselves. Since the capabilities parameters are not device-dependent, `deviceID` may be left empty in this case. For example, here we find a text name for the capability parameter `DM_PARENT_ID`:

Example 3-6 Get Capabilities of Parameters that Describe Capabilities

```
DMpv* paramCap, *paramName;
dmGetCapabilities( 0, DM_PARENT_ID, &paramCap);
paramName = dmPvFind( paramCap, DM_NAME_BYTE_ARRAY);
if( paramName != NULL )
    printf("Param has name %s\n", (char *) ( paramName->value.pByte ));
dmFreeCapabilities( paramCap );
```

Again, you can get the same result by using `dmPvToString`, which itself calls `dmPvGetCapabilities`.

Identification Numbers

There are three types of ID numbers in the dmSDK:

ID Number Type	Definition
constant	These have defined names and may be hard-coded. They are system-independent. Examples of constant IDs are `DM_SYSTEM_LOCALHOST`, and `DM_IMAGE_WIDTH_INT32`.

static

Static IDs are allocated by the dmSDK system as new hardware is added. They are machine-dependent and may change after a reboot, or as the system is reconfigured by adding or removing devices. The static ID of a device may change if it is removed from the system and then reconnected.

Note: Static IDs should never be written to a file or passed between machines.

Examples of static IDs are the physical and logical device IDs returned in calls to `dmGetCapabilities`. If you need to share such information between machines, you should use the text names (system-independent) that correspond to the static IDs.

open

These IDs are allocated when logical devices are opened. They are machine-dependent, and have a limited lifetime — from when `dmOpen` is called until `dmClose` is called.

Note: Open IDs should never be written to a file, or passed between machines.

Note: You can call `dmGetCapabilities` (or `dmPvGetCapabilities`) for any type of ID, but the list that is returned will always be static.

System Capabilities

The following sections describe the capabilities of each type of DM object. The capabilities are not necessarily in the order shown. In these tables, the string in the Parameter column is a shortened form of the full parameter name. The full parameter name is of the form `DM_`*parameter_type*, where *parameter* and *type* are the strings listed in the Parameter and Type columns respectively. For example, the full name of `ID` is `DM_ID_INT64`.

Currently, the only defined physical system ID is `DM_SYSTEM_LOCALHOST`. When a system ID is queried, the resulting capabilities list contains the following parameters:

Table 3-1 System Capabilities

Parameter	Type	Description
ID	INT64	Resource ID for this system
NAME	BYTE_ARRAY	NULL-terminated ASCII string containing the hostname for this system.
SYSTEM_DEVICE_IDS	INT64_ARRAY	Array of physical device IDs (these need not be sorted or sequential). For more details on a particular device ID call dmGetCapabilities. This array could be of length zero.

Table 3-2 Physical Device Capabilities

Parameter	Type	Description
ID	INT64	Resource ID for this physical device.
NAME	BYTE_ARRAY	NULL-terminated ASCII description of this physical device (e.g. "HD Video I/O" or "AVC/1394").
PARENT_ID	INT64	Resource ID for the system to which this physical device is attached.
DEVICE_VERSION	INT_32	Version number for this particular physical device.

Parameter	Type	Description
DEVICE_INDEX	BYTE_ARRAY	Index string for this physical device. This is used to distinguish multiple identical physical devices - indexes are generated with a consistent algorithm - identical machine configurations will have identical indexes - e.g. plugging a particular card into the first 64-bit, 66MHz PCI slot in any system will give the same index number. Uniquely identifying a device in a system-independent way requires using both the name and index.
DEVICE_LOCATION	BYTE_ARRAY	Physical hardware location of this physical device (on most platforms this is the hardware graph entry). Makes it possible to distinguish between two devices on the same I/O bus, and two devices each with its own I/O bus.
DEVICE_JACK_IDS	INT64_ARRAY	Array of jack IDs. For more details on a particular jack ID call `dmGetCapabilities`. This array could be of length zero.
DEVICE_PATH_IDS	INT64_ARRAY	Array of path IDs. For more details on a particular path ID call `dmGetCapabilities`. This array could be of length zero.
DEVICE_XCODE_IDS	INT64_ARRAY	Array of transcoder device IDs (these need not be sorted or sequential). For more details on a particular transcoder ID call dmGetCapabilities. This array could be of length zero.

Jack Logical Device Capabilities

The capabilities for a jack logical device contain the following parameters:

Table 3-3 Jack Logical Device Capabilities

Parameter	Type	Description
`ID`	`INT64`	Resource ID for this jack.
`NAME`	`BYTE_ARRAY`	NULL-terminated ASCII description of this jack (e.g. "Purple S-video").
`PARENT_ID`	`INT64`	Resource ID for the physical device to which this jack is attached.

Parameter	Type	Description
JACK_TYPE	INT32	Type of logical jack: DM_JACK_TYPE_AUDIO DM_JACK_TYPE_VIDEO DM_JACK_TYPE_COMPOSITE DM_JACK_TYPE_SVIDEO DM_JACK_TYPE_SDI DM_JACK_TYPE_DUALLINK DM_JACK_TYPE_GENLOCK DM_JACK_TYPE_GPI DM_JACK_TYPE_SERIAL DM_JACK_TYPE_ANALOG_AUDIO DM_JACK_TYPE_AES DM_JACK_TYPE_GFX DM_JACK_TYPE_AUX DM_JACK_TYPE_ADAT Where: AUDIO is a generic audio jack, VIDEO is a generic video jack, COMPOSITE is a composite video jack, SVIDEO is a SVideo jack, SDI is a Serial Digital Interface jack, DUALLINK is a SDI dual link jack, GENLOCK is a genlock jack, GPI is a General Purpose Interface jack, SERIAL is a generic serial control jack, ANALOG_AUDIO is an analog audio jack, AES is a digital AES standard jack, GFX is a digital graphics jack, AUX is a generic auxiliary jack, and ADAT is a digital ADAT standard jack.

Parameter	Type	Description
JACK_DIRECTION	INT32	Direction of data flow through this jack. May be: DM_JACK_DIRECTION_IN DM_JACK_DIRECTION_OUT Where: IN is an input jack with data for memory and OUT is an output jack with data from memory.
JACK_COMPONENT_SIZE	INT32	Maximum number of bits of resolution per component for the signal through this jack. Stored as an integer, so 8 means 8 bits of resolution.
JACK_PATH_IDS	INT64_ARRAY	Array of path IDs that may use this jack. (These need not be sorted or sequential.) For more details on a particular path ID, call dmGetCapabilities. This array could be of length zero.
PARAM_IDS	INT64_ARRAY	List of resource IDs for parameters which may be set and/or queried on this jack.
OPEN_OPTION_IDS	INT64_ARRAY	List of resource IDs for open option parameters which may be used when this jack is opened
JACK_FEATURES	BYTE_ARRAY	Double NULL-terminated list of ASCII features strings. Each string represents a specific feature supported by this jack. Entries are separated by NULL characters (there are 2 NULLs after the last string).

Path Logical Device Capabilities

The capabilities list for a path logical device contains the following parameters:

Table 3-4 Path Logical Device Capabilities

Parameter	Type	Resource ID for this path.
ID	INT64	Resource ID for this path.
NAME	BYTE_ARRAY	NULL-terminated ASCII description of this path (e.g., "Memory to S-Video Out").
PARENT_ID	INT64	Resource ID for the physical device on which this path resides.
PARAM_IDS	INT64_ARRAY	List of resource IDs for parameters which may be set and/or queried on this path.
OPEN_OPTION_IDS	INT64_ARRAY	List of resource IDs for open option parameters which may be used when this path is opened.
PRESET	MSG_ARRAY	Each entry in the array is a message (a pointer to the head of a DMpv list, where the last entry in the list is DM_END). Each message provides a single valid combination of all setable parameters on this path. In particular, it should be possible to call dmSetControls using any of the entries in this array as the control's message. Each path is obligated to provide at least one preset.

Parameter	Type	Resource ID for this path.
PATH_TYPE	INT32	Type of this path: DM_PATH_TYPE_MEM_TO_DEV DM_PATH_TYPE_DEV_TO_MEM DM_PATH_TYPE_DEV_TO_DEV Where: MEM_TO_DEV is a path from memory to a device, DEV_TO_MEM is a path from device to memory and DEV_TO_DEV is a path from device to another device.
PATH_COMPONENT _ALIGNMENT	INT32	The location in memory of the first byte of a component (either an audio sample or a video line), must meet this alignment. Stored as an integer in units of bytes.
PATH_BUFFER _ALIGNMENT	INT32	The location in memory of the first byte of an audio or video buffer must meet this alignment. Stored as an integer in units of bytes
PATH_SRC_JACK_ID	INT64	Resource ID for the jack which is the source of data for this path (unused if path is of type DM_PATH_TYPE_MEM_TO_DEV). For details on the jack ID call dmGetCapabilities.

Parameter	Type	Resource ID for this path.
PATH_DST_JACK_ID	INT64	Resource ID for the jack which is the destination for data from this path (unused if path is of type DM_PATH_TYPE_DEV_TO_MEM). For details on the jack ID call dmGetCapabilities.
PATH_FEATURES	BYTE_ARRAY	Double NULL-terminated list of ASCII features strings. Each string represents a specific feature supported by this path. Entries are separated by NULL characters (there are 2 NULLs after the last string).

Transcoder Logical Device Capabilities

The capabilities list for a transcoder logical device contains the following parameters:

Table 3-5 Transcoder Logical Device Capabilities

Parameter	Type	Description
ID	INT64	Resource ID for this transcoder.
NAME	BYTE_ARRAY	NULL-terminated ASCII description of this transcoder (e.g. "Software DV and DV25").
PARENT_ID	INT64	Resource ID for the physical device on which the transcoder resides.
PARAM_IDS	INT64_ARRAY	List of resource IDs for parameters which may be set and/or queried on this transcoder (May be of length 0).
OPEN_OPTION_IDS	INT64_ARRAY	List of resource IDs for open option parameters which may be used when this xcode is opened

Parameter	Type	Description
PRESET	MSG_ARRAY	Each entry in the array is a message (a pointer to the head of a DMpv list, where the last entry in the list is DM_END). Each message provides a single valid combination of all setable parameters on a transcoder. In particular, it should be possible to call dmSetControls using any of the entries in this array as the controls message. Each transcoder is required to provide at least one preset for each transcoder.
XCODE_ENGINE_TYPE	INT32	Type of the engine in this transcoder. At this time the only defined xcode type is: DM_XCODE_ENGINE_TYPE_NULL
XCODE _IMPLEMENTATION _TYPE	INT32	How this transcoder is implemented: DM_XCODE_IMPLEMENTATION_TYPE_SW DM_XCODE_IMPLEMENTATION_TYPE_HW The implementation of the transcoder could be in either software (SW) or hardware (HW).
XCODE _COMPONENT _ALIGNMENT	INT32	The location in memory of the first byte of a component (either an audio sample or a video pixel), must meet this alignment. Stored as an integer in units of bytes.
XCODE _BUFFER _ALIGNMENT	INT32	The location in memory of the first byte of an audio or video buffer must meet this alignment. Stored as an integer in units of bytes
XCODE_FEATURES	BYTE_ARRAY	Double NULL-terminated list of ASCII features strings. Each string represents a specific feature supported by this xcode. Entries are separated by NULL characters (there are 2 NULLs after the last string)

Parameter	Type	Description
XCODE_SRC_PIPE_IDS	INT64_ARRAY	List of pipe IDs from which the transcode engine may obtain buffers to be processes.
XCODE_DEST_PIPE_IDS	INT64_ARRAY	List of pipe IDs from which the transcode engine may obtain buffers to be filled with the result of its processing.

Pipe Logical Device Capabilities

The capabilities list for a pipe logical device contains the following parameters:

Table 3-6 Pipe Logical Device Capabilities

Parameter	Type	Description
ID	INT64	Resource ID for this path.
NAME	BYTE_ARRAY	NULL-terminated ASCII description of this pipe ("DV Codec Input Pipe").
PARENT_ID	INT64	Resource ID for the transcoder on which this pipe resides.

Parameter	Type	Description
PARAM_IDS	INT64_ARRAY	List of resource IDs for parameters which may be set and/or queried on this transcoder (May be of length 0).
PIPE_TYPE	INT32	Type of this pipe: DM_PIPE_TYPE_MEM_TO_ENGINE DM_PIPE_TYPE_ENGINE_TO_MEM MEM_TO_ENGINE is the trancoder input pipe with data flow from memory to engine. ENGINE_TO_MEM is the transcoder output pipe with data flow from engine to memory.

Finding a Parameter in a Capabilities List

A parameter within a message or capabilities list may be found using

`DMpv* dmPvFind(DMpv*` *msg*, `DMint64` *param*);

msg points to the first parameter in an `DM_END` terminated array of parameters and param is the 64-bit unique identifier of the parameter to be found. `dmPvFind` returns the address of the parameter if successful; otherwise it returns NULL.

Obtaining Parameter Capabilities

All objects in DM are referred to via 64–bit identifying numbers. For example, the 64–bit ID number for the system running the application is `DM_SYSTEM_LOCALHOST`. Details on the interpretation of a particular device dependent parameter are obtained using:

`DMstatus dmPvGetCapabilities(DMint64` *objectId*, `DMint64` *parameterId*, `DMpv**` *capabilities*);

objectId is the 64-bit unique identifier for the object whose parameter is being queried. An example is the `openId` returned from a call to `dmOpen`. The status

DM_STATUS_INVALID_ID is returned if the specified object ID was invalid. *parameterId* is the 64-bit unique identifier for the parameter whose capabilities are being queried. The status DM_STATUS_INVALID_ARGUMENT is returned if the capabilities pointer is invalid. Capabilities is a pointer to the head of the resulting capabilities list. This list should be treated as read-only by the application. If the call was successful, then the status dm_STATUS_NO_ERROR is returned.

objectid may be either a static ID (obtained from a previous call to dmGetCapabilities) or an open ID (obtained by calling dmOpen.) Querying the capabilities of an opened object is identical to querying the capabilities of the corresponding static object.

It is also possible to get the capabilities of the capabilities parameters themselves. Those parameters are not tied to any particular object and so the *objectId* should be 0.

The list returned in *capabilities* contains the following parameters, though not necessarily in this order. The string in the Parameter column is a shortened form of the full parameter name. The full parameter name is of the form DM_*parameter_type*, where *parameter* and *type* are the strings listed in the Parameter and Type columns respectively. For example, the full name of ID is DM_ID_INT64

Table 3-7 Parameters returned by dmPvGetCapabilities

Parameter	Type	Description
ID	INT64	Resource ID for this parameter.
NAME	BYTE_ARRAY	NULL-terminated ASCII name of this parameter. This is identical to the enumerated value. For example, if the value is DM_XXX, then the name is "DM_XXX".
PARENT_ID	INT64	Resource ID for the logical device (video path or transcoder pipe) on which this parameter is used.

Parameter	Type	Description
PARAM_TYPE	INT32	Type of this parameter: DM_TYPE_INT32 DM_TYPE_INT32_POINTER DM_TYPE_INT32_ARRAY DM_TYPE_INT64 DM_TYPE_INT64_POINTER DM_TYPE_INT64_ARRAY DM_TYPE_REAL32 DM_TYPE_REAL32_POINTER DM_TYPE_REAL32_ARRAY DM_TYPE_REAL64 DM_TYPE_REAL64_POINTER DM_TYPE_REAL64_ARRAY DM_TYPE_BYTE_POINTER DM_TYPE_BYTE_ARRAY
PARAM_ACCESS	INT32	Access controls for this parameter. Bitwise "or" of the following flags: DM_ACCESS_READ DM_ACCESS_WRITE DM_ACCESS_OPEN_OPTION DM_ACCESS_IMMEDIATE(use in set/get) DM_ACCESS_QUEUED (usein send/query) DM_ACCESS_SEND_BUFFER (only in dmSendBuffers) DM_ACCESS_DURING_TRANSFER DM_ACCESS_PASS_THROUGH (ignored by device)
PARAM_DEFAULT	same type as param	Default value for this parameter of type DM_PARAM_TYPE. (This parameter may be of length 0 if there is no default).

Parameter	Type	Description
PARAM_MINS	array of same type as param	Array of minimum values for this parameter (may be missing if there are no specified minimum values). Each set of min/max values defines one allowable range of values. If min equals max then the allowable range is a single value. If the array is of length one, then there is only one legal range of values. (will be of length 0 if there are no specified minimum values).
PARAM_MAXS	array of same type as param	Array of maximum values for this parameter (may be of length 0 if there are no specified maximum values). There is one entry in this array for each entry in the mins array.
PARAM_INCREMENT	same type as param	Legal param values go from min to max in steps of increment. (will be of length 0 if there is no increment).

Parameter	Type	Description
PARAM_ENUM_VALUES	same type as param	Array of enumerated values for this parameter (will be of length 0 if there are no enumeration values).
PARAM_ENUM_NAMES	BYTE_ARRAY	Array of enumeration names for this parameter (must have one entry for each element in the values array). The array is a double NULL-terminated list of ASCII strings. Each string represents a specific enumeration name corresponding to the enumerated value in the same position in the PARAM_ENUM_VALUES array. Entries are separated by NULL characters (there are 2 NULLs after the last string).

Freeing Capabilities Lists

A capabilities list *capabilities* obtained from either `dmGetCapabilities` or `dmPvGetCapabilities` is returned to the system using `DMstatus dmFreeCapabilities (DMpv *capabilities);`.

The status `DM_STATUS_INVALID_ARGUMENT` is returned if the capabilities pointer is invalid. The `DM_STATUS_NO_ERROR` is returned if the call was successful.

Audio/Visual Paths

In the dmSDK, the logical connections between jacks and memory are called *paths*. For example, a video output path provides the means to transfer video information from buffers in memory, through a video output jack.

Opening a Logical Path

Before you send messages to a device, you must open a processing path that goes through it. This is done by calling `dmOpen`(3dm) as follows:

```
DMstatus dmOpen (DMint64 pathId, DMpv* options,
                    DMopenid* openid);
```

Think of a path as a logical device - a physical device (for example, a PCI card) may simultaneously support several such paths. A side effect of opening a path is that space is allocated for queues of messages from your application to the device and replies from the device back to your application. All of the messages sent to a queue share a common payload area and are required to observe a strictly ordered relationship. That is, if message A is sent before message B, then the reply to A must arrive before the reply to B.

Constructing a Message

Messages are arrays of parameters, where the last parameter is always `DM_END`. For example, set the image width to be 720 and the image height to be 480 as follows:

```
DMpv message[3];
message[0].param = DM_IMAGE_WIDTH_INT32;
message[0].value.int32 = 720;
message[1].param = DM_IMAGE_HEIGHT_INT32;
message[1].value.int32 = 480;
message[2].param = DM_END
```

Processing Out-of-Band Messages

In some cases, an application wishes to influence a device without first waiting for all previously enqueued messages to be processed. Borrowing a term from UNIX communications, we term these cases *out-of-band messages*. They are performed with the `dmSetControls`(3dm) or `dmGetControls`(3dm) calls.

Here is an example of how you can immediately get the width and height of an image:

```
DMpv message[3];
message[0].param = DM_IMAGE_WIDTH_INT32;
message[1].param = DM_IMAGE_HEIGHT_INT32;
message[2].param = DM_END
if( dmGetControls( somePath, message))
   handleError();
else
  printf("Image size is %d x %d\n",
          message[0].value.int32,
          message[1].value.int32);
```

Out-of-band messages work well for simple sets and queries. They are blocking calls. If the call succeeds, the message has been successfully processed.

Sending In-Band Messages

Out-of-band messages are appropriate for simple control changes, but they provide no buffering between your application and the device. For most applications, processing real-time data will require using a queuing communication model. The dmSDK supports this with the `dmSendControls`(3dm), `dmSendBuffers`(3dm), and `dmReceiveMessage`(3dm) calls.

For example, to send a controls message to a device input queue use:

```
DMstatus dmSendControls( DMopenid openId, DMpv* message);
```

Devices interpret messages in the order in which they are enqueued. Because of this, the time relationship is explicit between, for example, video buffers and changes in video modes.

Note: The `send` call does not wait for a device to process the message. Rather, it copies it to the device input queue and then returns.

When your application sends a message, it is copied into the send queue. The message is then split between a small fixed header on the input list, and a larger, variable-sized space in the data area.

Sometimes, there is not enough space in the data area and/or send list for new messages. In that case the return code indicates that the message was not enqueued. As a rule, a full input queue is not a problem — it simply indicates that the application is generating messages faster than the device can process them.

For some devices, the system may use device-specific knowledge to best manage messaging transactions. For example, when you call `sendBuffers` the system may copy the message exactly as described above, or it may send part or all of the message directly to the hardware. Regardless of what happens, the system always looks to your application as described here.

Each message you send is guaranteed to result in at least one reply message from the device. This is how you know when your message is interpreted and what is the result.

- In the case of control parameters you should check the return message to make sure your control executed correctly.
- In the case of video buffers, you should allocate buffer space in your application and then send an indirect reference to that buffer in a message. Once your application receives a reply message you can be certain the device has completed your request and finished with the memory, so you are free to reuse it.

Some devices can send messages to advise your application of important events (for example some video devices can notify you of every vertical retrace). However, no notification messages will be generated unless and until you explicitly request them.

Processing In-Band Messages

The device processes messages as follows:

Removes the message header from the send queue, processes the message and writes any response into the payload area. It then places a reply header on the receive queue.

In general, your application must allow space in the message for any reply you expect to be returned.

Note: The device performs no memory allocation, but rather uses the memory allocated when the application enqueued the input message. This is important because it guarantees there will never be any need for the device to block because it did not have enough space for the reply.

Processing Exception Events

In some cases an exception event occurs which requires that the device pass a message back to your application. Your application must explicitly ask for such events.

Possible exception events are:

```
DM_EVENT_DEVICE_ERROR  Device encountered an error and is unable to recover.
DM_EVENT_DEVICE_UNAVAILABLE  The device is not available for use.
DM_EVENT_VIDEO_SEQUENCE_LOST  A video buffer was not available for an I/O transfer.
DM_EVENT_VIDEO_SYNC_LOST  Device lost the output genlock sync signal.
DM_EVENT_VIDEO_SYNC_GAINED  Device detected a valid output genlock.
DM_EVENT_VIDEO_SIGNAL_LOST  Device lost the video input signal.
DM_EVENT_VIDEO_SIGNAL_GAINED Device detected a valid input video signal.
DM_EVENT_VIDEO_VERTICAL_RETRACE  A video vertical retrace occurred.
DM_EVENT_AUDIO_SEQUENCE_LOST  An audio buffer was not available for an I/O transfer.
DM_EVENT_AUDIO_SAMPLE_RATE_CHANGED  The audio input sampling frequency changed.
```

If you ask for events, your application must read its receive queue frequently enough to prevent the device running out of space for messages which you have asked it to enqueue. If the queue starts to fill up, then the device will enqueue an event message advising that it is stopping notification of exception events.

Note: The device never needs to allocate space in the data area for reply messages. It will automatically stop sending notifications of events if the output list starts to fill up. Space is reserved in the receive queue for a reply to every message your application enqueues; but if there is insufficient space, attempts to send new messages will fail.

Processing In-Band Reply Messages

To receive a reply message from a device use `dmReceiveMessage`(3dm) as follows:

```
DMstatus dmReceiveMessage(DMopenid openId,
                          DMint32* messageType,
                          DMpv** reply);
```

This call returns the earliest unread message sent from the device back to your application. The `messageType` parameter indicates why this reply was generated. It could come from a call to `sendControls`, `sendBuffers`, or it could have been generated spontaneously by the device as the result of an event. The reply pointer is guaranteed to remain valid until you attempt to receive a subsequent message. This allows a small optimization — you can read the current message "in place" without needing to first copy it off the queue. It is acceptable to overwrite a value in a reply message and then send that as a new message.

Beginning and Ending Transfers

Devices do not begin to process enqueued messages until explicitly instructed to by an application.

This is done with the `dmBeginTransfer`(3dm) call:

```
DMstatus dmBeginTransfer( DMopenid openId);
```

This call frees the device to begin processing enqueued messages. It also commands the device to begin generating exception events. Typically, an application will open a device, enqueue several buffers (priming the input queue) and then call `BeginTransfer`. In this way, it avoids the underflow which could otherwise occur if the application were swapped out immediately after enqueueing the first buffer to the device.

To stop a transfer, call `dmEndTransfer`(3dm):

```
DMstatus dmEndTransfer( dmopenid openId);
```

This causes the device to do the following:

- Stop processing messages containing buffers
- Flush its input queue
- Stop notification of exception events

Closing a Logical Path

When your application has finished using an open path, it may close it (see `dmClose`(3dm)):

```
DMstatus dmClose( DMopenId openId);
```

This causes an implicit `EndTransfer` on any device with an active transfer. It then frees any resources used by the device. If you wish to have pending messages processed prior to closing a device, you must identify a message (perhaps by adding a piece of user data or by remembering its MSC number) and make sure it is the last thing you enqueue. When it appears on the output queue, you will know all messages have been processed. At that point you can close the device.

Transcoders

A *transcoder* provides a means to process data in memory. Support for transcoders may be implemented entirely in software, or it may be performed with hardware assistance. In either case, the software interfaces are consistent.

Each dmSDK transcoder device consists of the following:

- A transcoder *engine* which performs the actual processing
- A number of source pipes and destination pipes

The engine takes data from buffers in the source pipes, processes it, and stores the result in buffers in the destination pipes. Each pipe acts much like a path which provides two things: a way for your application to send buffers containing bits to be processed, and a way to send empty buffers to hold the results of that processing.

Finding a Suitable Transcoder

Use `dmGetCapabilites`(3dm) to obtain details of all transcoders on the system.

Opening a Logical Transcoder

Open a transcoder in much the same way as a path, but using `dmOpen`(3dm):

```
DMstatus dmOpen (DMint64 xcodeId, DMpv* options, DMopenid* openid);
```

When a transcoder is opened, it creates any required source and destination pipes. Just as for a path, an open transcoder is a logical entity — as such, a single physical device may support several transcoders simultaneously.

Controlling the Transcoder

The transcoder engine is controlled indirectly through the source and destination pipes:

- Controls on the source pipe describe what you will be sending the transcoder for input.

- Controls on the destination pipe describe the results you want.

The difference between the source and destination controls dictates what operations the transcoder should perform.

For example, if the DM_IMAGE_CODING is UNCOMPRESSED on the source and DVCPRO_50 on the destination, then you are requesting the transcoder to:

- Take uncompressed data from the source pipe.
- Apply a DVCPRO_50 compression.
- Write the results to the destination pipe.

To set controls on a transcoder, construct a controls message as you would for a video path. The only difference is that you must explicitly direct controls to a particular pipe. This is done through the DM_SELECT_ID parameter, which directs all following controls to a particular ID (in this case, the ID of a pipe on the transcoder).

For example, here is code to set image width and height on both the source and destinations pipes:

Example 5-1 Set Image Width/Height on Pipes

```
msg[0].param = DM_SELECT_ID_INT64;
msg[0].value.int64 = DM_XCODE_SRC_PIPE;
msg[1].param = DM_IMAGE_WIDTH_INT32;
msg[1].value.int32 = 1920;
msg[2].param = DM_IMAGE_HEIGHT_INT32;
msg[2].value.int32 = 1080;
msg[3].param = DM_SELECT_ID_INT64;
msg[3].value.int64 = dm_XCODE_DST_PIPE;
msg[4].param = DM_IMAGE_WIDTH_INT32;
msg[4].value.int32 = 1920;
msg[5].param = DM_IMAGE_HEIGHT_INT32;
msg[5].value.int32 = 1280;
msg[6].param = DM_END;

dmSetControls(someOpenXcode, msg);
```

Sending Buffers

Once the controls on a pipe have been set, you may begin to send buffers to it for processing. Do this with the `dmSendBuffers`(3dm) call.

Call `dmSendBuffers` once for all the buffers corresponding to a single instant in time. For example, if the transcoder expects both an image buffer and an audio buffer, you must send both in a single `sendBuffers` call.

For example, here is code to send a source buffer to the source pipe, and a destination buffer to the destination pipe:

Example 5-2 Send Source/Destination Buffers to Source/Destination Pipes

```
msg[0].param = DM_SELECT_ID_INT64;
msg[0].value.int64 = DM_XCODE_SRC_PIPE;
msg[1].param = DM_IMAGE_BUFFER_POINTER;
msg[1].value.pByte = srcBuffer;
msg[1].length = srcImageSize;
msg[2].param = DM_SELECT_ID_INT64;
msg[2].value.int64 = DM_XCODE_DST_PIPE;
msg[3].param = DM_IMAGE_BUFFER_POINTER;
msg[3].value.pByte = dstBuffers;
msg[3].maxLength = dstImageSize;
msg[4].param = DM_END;

dmSendBuffer(someOpenXcode, msg);
```

Starting a Transfer

The `sendBuffers` call places buffer messages on a pipe queue to the device. You must then call `dmBeginTransfer`(3dm) to tell the transcoder engine to start processing messages.

Note: The `beginTransfer` call may fail if the source and destination pipe settings are inconsistent.

Changing Controls During a Transfer

During a transfer, you could attempt to change controls by using `dmSetControls`(3dm), but this is often undesirable since the effect of the control change on buffers currently being processed is undefined. A better method is to send control changes in the same queue as the buffer messages. Do this with the same `dmSendControls`(3dm) call as on a path, again using `DM_SELECT_ID` to direct particular controls to a particular pipe.

Note: Parameter changes sent with `sendControls` are guaranteed to only affect buffers sent with subsequent send calls.

Note: Some hardware transcoders may be unable to accommodate control changes during a transfer. If in doubt, examine the capabilities of particular parameter to determine if it may be changed while a transfer is in progress.

In a transcoder, it is possible to get the following exception event:

`DM_EVENT_XCODE_FAILED`

Transcoder was unable to process data.

Receiving a Reply Message

Whenever you pass buffer pointers to the transcoder (by calling `sendBuffers`) you give up all rights to that memory until the transcoder has finished using it. As the transcoder finishes processing each buffers message, it will enqueue a reply message back to your application. You may read these reply messages in exactly the same way as on a path by calling `dmReceiveMessage`(3dm).

The transcoder queue maintains a strict first-in, first-out ordering. If buffer A is sent before buffer B, then the reply to A will come before the reply to B. This is guaranteed even on transcoders which parallelize across multiple physical processors.

Ending Transfers

To end a transfer, call `dmEndTransfer`(3dm). This is a blocking call which:

- Allows all buffers currently in the engine to complete
- Marks any remaining messages as "aborted"

By examining the reply to each message, your application can determine whether or not it was successfully processed.

It is also acceptable to call `endTransfer` before `beginTransfer` has been called. In that case any messages in the queue are aborted and returned to the application.

Note: If you are not interested in the result of any pending buffers, you can simply close the transcoder without bothering to first end the transfer.

Closing a Transcoder

When your application has finished using a transcoder it may close it, see `dmClose`(3dm) :

```
DMstatus dmClose( DMopenId openId);
```

This causes an implicit `EndTransfer`. It then frees any resources used by the device.

Work Functions

In most cases, the difference between hardware and software transcoders is transparent to an application. Software transcoders may have more options and may run more slowly, but for many applications these differences are not significant.

One notable difference between hardware and software transcoders is that software transcoders will attempt to use as much of the available processor time as possible. This may be undesirable for some applications. To counter this, an application has the option to do the work of the transcoder itself, in its own thread. This is achieved with the `dmXcodeWork`(3dm) function.

If you open a software transcoder while setting the `DM_XCODE_MODE_SYNCHRONOUS` option, the transcoder will not spawn any threads and will not do any processing on its own. To perform a unit of transcoding work, your application must now call the `dmXcodeWork`(3dm) function.

Note: This only applies to software transcoders, and only if you set the `DM_XCODE_MODE_SYNCHRONOUS` option when opening the transcoder.

Multi-Stream Transcoders

This chapter has described the operation of a single-stream transcoder (one in which all controls/buffers can be sent to the transcoder engine using the `DM_SELECT_ID` parameter). Some transcoders, however, particularly those which need to consume source and destination buffers at different rates, will not work efficiently with this programming model. For those cases, it is possible to access each transcoder pipe individually, sending/receiving buffers on the source pipe at a different rate than on the destination pipe. This will be supported in a future revision of the dmSDK.

Video Parameters

The processing of a video input/output path is described by two sets of parameters:

- Video parameters describe how to interpret and generate the signal as it arrives and leaves, as discussed in this chapter.
- Image parameters describe how to write/read the resulting bits to/from the device (see Chapter 7, "Image Parameters", page 67).

Not all parameters may be supported on a particular video jack or path. Note that some parameters may be adjusted on both a path and a jack, or may be adjustable on just one or the other. Use `dmGetCapabilities` to obtain a list of parameters supported by a jack or path. In addition, not all values may be supported on a particular parameter. Use `dmPvGetCapabilities` to obtain a list of the values supported by the parameter.

Note: This chapter, as well as Chapter 7, "Image Parameters", page 67 assume a working knowledge of digital video concepts. Readers unfamiliar with terms like *video timing*, *422* or *CbYCr* may wish to consult a text devoted to this subject. A good resource is *A Technical Introduction to Digital Video*, by Charles Poynton, published by John Wiley & Sons, 1996 (ISBN 0-471-12253-X, hardcover).

Video Sampling

There are two kinds of video sampling, spatial and temporal. Our concern here is with temporal sampling, of which there are two techniques:

- *progressive* sampling is frame-based (for example, from film).
- *interlaced* sampling is field-based.

Progressive Sampling

In progressive, frame-based sampling, a picture at a specified resolution is sampled at constant rate. Film is a progressive sampling source for video.

Imagine an automatic film advance camera that can take 60 pictures-per-second, with which you take a series of pictures of a moving ball. Figure 6-1, page 56 shows 10 pictures from that sequence (different colors emphasize the different positions of the ball in time). The time delay between each picture is a 60th of a second, so this sequence lasts 1/6th of a second.

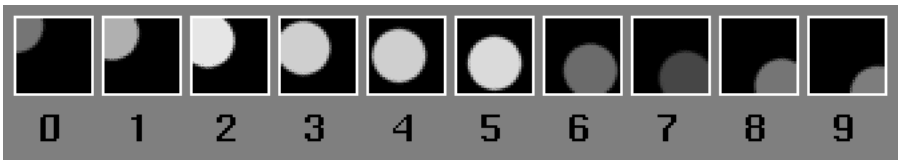

Figure 6-1 Film at 60 Frames-per-Second

Interlaced Sampling

Interlaced sampling is more involved than progressive sampling. Here, the video is sampled in a periodicity of two *sample fields*, called F1 and F2, such that half of the display lines of the picture are scanned at a time. Like window blinds, every other line in a sample field is blank.

Pairs of sample fields are superimposed on each other or "interlaced" to create the *video frame*. In the video frame, the sample frames, while consecutive, appear coincident to the eye. This effect is aided by the persistence of phosphors on the display screen which hold the impression of the first set of scanned lines as the second set displays. (This sequence is made visible if you videotape a computer monitor display.)

Most video signals in use today, including several high-definition video formats, are field-based (interlaced) rather than frame-based (progressive). In the dmSDK, the value of the Video Timing parameter `DM_VIDEO_TIMING_INT32` defines the specific video standard, and each standard is defined as progressive or interlaced.

Example of Interlaced Sampling

Suppose you shoot the moving ball with an NTSC video camera. NTSC video has 60 fields-per-second, so you might think that the video camera would record the same series of pictures as shown in Figure 6-1, page 56, but it does not. The video camera does record 60 images per second, but each image consists of only half of the scanned

lines of the complete picture at a given time, as shown in Figure 6-2, page 57, rather than a filmstrip of 10 complete images.

Note how the image lines alternate between odd- and even-numbered images.

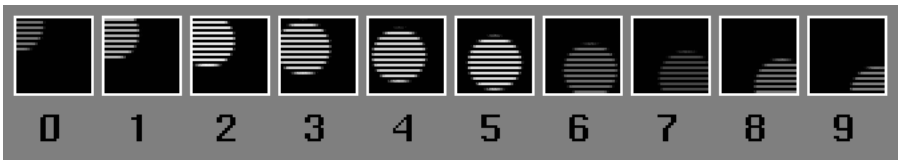

Figure 6-2 Video at 60 Frames-per-Second

Video Parameters

Parameter	Purpose
`DM_VIDEO_TIMING_INT32`	video timing parameter
`DM_VIDEO_COLORSPACE_INT32`	video colorspace parameter
`DM_VIDEO_PRECISION_INT32`	video precision parameter
`DM_TIMING_UNKNOWN`	Timing of input signal cannot be determined.
`DM_TIMING_NONE`	No signal present.

`DM_VIDEO_TIMING_INT32`

This parameter sets the timing on an input or output video path. Not all timings may be supported on all devices. On devices which can auto-detect, the timing may be read-only on input. (Details of supported timings may be obtained by calling `dmPvGetCapabilites` on this parameter). Figure B-1, page 139 and Figure B-2, page 140 illustrate details of the 601 standard.

Supported Timings

Note: See Appendix B, "Common Video Standards", page 139 for diagrams of common video standards.

These format for these timings are as follows:

`DM_TIMING_`*xxxx*`_`*yyyy*`x`*zzzz*`_`*nnn*`[i|p|PsF]`

where:

xxxx	Total number of lines.
yyyy `x` *zzzz*	Width by height of the active video region (high definition).
nnn`[i|p|PsF]`	The frame rate, followed by `i`, `p`, or `PsF` to indicate interlaced, progressive, or segmented Frame, respectively.

Standard Definition (SD) Timings

`DM_TIMING_525` (NTSC)
`DM_TIMING_625` (PAL)

High Definition (HD) Timings

```
DM_TIMING_1125_1920x1080_60p
DM_TIMING_1125_1920x1080_5994p
DM_TIMING_1125_1920x1080_50p
DM_TIMING_1125_1920x1080_60i
DM_TIMING_1125_1920x1080_5994i
DM_TIMING_1125_1920x1080_50i
DM_TIMING_1125_1920x1080_30p
DM_TIMING_1125_1920x1080_2997p
DM_TIMING_1125_1920x1080_25p
DM_TIMING_1125_1920x1080_24p
DM_TIMING_1125_1920x1080_2398p
DM_TIMING_1125_1920x1080_24PsF
DM_TIMING_1125_1920x1080_2398PsF
DM_TIMING_1125_1920x1080_30PsF
DM_TIMING_1125_1920x1080_2997PsF
DM_TIMING_1125_1920x1080_25PsF
DM_TIMING_1250_1920x1080_50p
DM_TIMING_1250_1920x1080_50i
DM_TIMING_1125_1920x1035_60i
DM_TIMING_1125_1920x1035_5994i
DM_TIMING_750_1280x720_60p
```

```
DM_TIMING_750_1280x720_5994p
```

DM_VIDEO_COLORSPACE_INT32

Sets the colorspace at the video jack. For input paths, this is the colorspace you expect to receive at the jack. For output paths, it is the colorspace you desire at the jack.

See "DM_IMAGE_COLORSPACE_INT32", page 76 for a detailed description of colorspace values.

Supported Colorspace Values

```
DM_COLORSPACE_RGB_601_FULL
DM_COLORSPACE_RGB_601_HEAD
DM_COLORSPACE_CbYCr_601_FULL
DM_COLORSPACE_CbYCr_601_HEAD
DM_COLORSPACE_RGB_240M_FULL
DM_COLORSPACE_RGB_240M_HEAD
DM_COLORSPACE_CbYCr_240M_FULL
DM_COLORSPACE_CbYCr_240M_HEAD
DM_COLORSPACE_RGB_709_FULL
DM_COLORSPACE_RGB_709_HEAD
DM_COLORSPACE_CbYCr_709_FULL
DM_COLORSPACE_CbYCr_709_HEAD
```

DM_VIDEO_SAMPLING_INT32

Sets the sampling at the video jack. (See "DM_IMAGE_SAMPLING_INT32", page 78 for a detailed description of sampling values.)

Supported Sampling Values

```
DM_SAMPLING_422
DM_SAMPLING_4224
DM_SAMPLING_444
DM_SAMPLING_4444
```

DM_VIDEO_PRECISION_INT32

Sets the precision (number of bits of resolution) in the signal at the jack. This is an integer. A precision value of `10`, indicates a 10-bit signal. A value of 8 indicates an 8-bit signal.

DM_VIDEO_GENLOCK_SIGNAL_PRESENT_INT32

Used to query the incoming genlock signal for an output path. Not all devices may be able to sense genlock timing, but those that do will support this parameter. Common values match those for `DM_VIDEO_TIMING`, with two additions: `DM_TIMING_NONE` (there is no signal present) and `DM_TIMING_UNKNOWN` (the timing of the genlock cannot be determined).

DM_VIDEO_SIGNAL_PRESENT_INT32

Used to query the incoming signal on an input path. Not all devices may be able to sense timing, but those that do will support this parameter. Common values match those for `DM_VIDEO_TIMING`, with two additions: `DM_TIMING_NONE` (there is no signal present) and `DM_TIMING_UNKNOWN` (the timing of the input signal cannot be determined).

DM_VIDEO_GENLOCK_SOURCE_TIMING_INT32

Describes the genlock source timing. Only accepted on output paths. Each genlock source is specified as an output timing on the path and corresponds to the same timings as available with `DM_VIDEO_TIMING_INT32.`

DM_VIDEO_GENLOCK_TYPE_INT32

Describes the genlock signal type. Only accepted on output paths. Each genlock type is specified as either a 32–bit resource Id or `DM_VIDEO_GENLOCK_TYPE_INTERNAL`.

DM_VIDEO_BRIGHTNESS_INT32

Set or get the video signal brightness.

DM_VIDEO_CONTRAST_INT32

Set or get the video signal contrast.

DM_VIDEO_HUE_INT32

Set or get the video signal HUE.

DM_VIDEO_SATURATION_INT32

Set or get the video signal color saturation.

DM_VIDEO_RED_SETUP_INT32

Set or get video signal RED channel setup.

DM_VIDEO_GREEN_SETUP_INT32

Set or get the video signal GREEN channel setup.

DM_VIDEO_BLUE_SETUP_INT32

Set or get the video signal BLUE channel setup.

DM_VIDEO_ALPHA_SETUP_INT32

Set or get the video signal ALPHA channel setup.

DM_VIDEO_H_PHASE_INT32

Set or get the video signal horizontal phase genlock offset.

DM_VIDEO_V_PHASE_INT32

Set or get the video signal vertical phase genlock offset.

DM_VIDEO_FLICKER_FILTER_INT32

Set or get the video signal filter.

DM_VIDEO_DITHER_FILTER_INT32

Set or get the video signal dither filter.

DM_VIDEO_NOTCH_FILTER_INT32

Set or get the video signal notch filter.

DM_VIDEO_INPUT_DEFAULT_SIGNAL_INT64

Set or get the video signal default input signal.

DM_VIDEO_OUTPUT_DEFAULT_SIGNAL_INT64

Sets the default signal at the video jack when there is no active output. The only allowable are:

`DM_SIGNAL_NOTHING` indicates that output signal shall cease without generation of sync.

`DM_SIGNAL_BLACK` indicates that output shall generate a black picture complete with legal sync values.

`DM_SIGNAL_COLORBARS` indicates that output should use an internal colorbar generator.

`DM_SIGNAL_INPUT_VIDEO` indicates that output should use the default input signal as a pass through.

DM_VIDEO_START_Y_F1_INT32

Sets the start vertical location on F1 fields of the video signal. For progressive signals it specifies the start of every frame.

DM_VIDEO_OUTPUT_REPEAT_INT32

If the application is doing output and fails to provide buffers fast enough (the queue to the device underflows), then this control determines the device behavior. Allowable options are:

DM_VIDEO_REPEAT_NONE	The device does nothing, usually resulting in black output.
DM_VIDEO_REPEAT_FIELD	The device repeats the last field. For progressive signals or interleaved formats, this is the same as DM_VIDEO_REPEAT_FRAME.
DM_VIDEO_REPEAT_FRAME	The device repeats the last two fields. This output capability is device dependent and the allowable settings should be queried via the get capabilities of the DM_VIDEO_OUTPUT_REPEAT_INT32 parameter.

DM_VIDEO_FILL_Cr_REAL32

The Cr value for any pixel outside the clipping region. This is a real number: a value of 0.0 is the minimum legal value, 1.0 is the maximum legal value. Default is 0.

DM_VIDEO_FILL_Cb_REAL32

The Cb value for any pixel outside the clipping region. This is a real number: a value of 0.0 is the minimum legal value, 1.0 is the maximum legal value. Default is 0.

DM_VIDEO_FILL_RED_REAL32

The red value for any pixel outside the clipping region. This is a real number: a value of 0.0 is the minimum legal value (black), 1.0 is the maximum legal value. Default is 0.

DM_VIDEO_FILL_GREEN_REAL32

The green value for any pixel outside the clipping region. This is a real number: a value of 0.0 is the minimum legal value, 1.0 is the maximum legal value. Default is 0.

DM_VIDEO_FILL_BLUE_REAL32

The blue value for any pixel outside the clipping region. This is a real number: a value of 0.0 is the minimum (fully transparent), 1.0 is the maximum (fully opaque). Default is 1.0.

DM_VIDEO_START_X_INT32

Sets the start horizontal location on each line of the video signal.

DM_VIDEO_START_Y_F2_INT32

Sets the start vertical location on F2 fields of the video signal. Ignored for progressive timing signals.

DM_VIDEO_WIDTH_INT32

Sets the horizontal width of the clipping region on each line of the video signal.

DM_VIDEO_HEIGHT_F1_INT32

Sets the vertical height for each F1 field of the video signal. For progressive signals it specifies the height of every frame.

DM_VIDEO_HEIGHT_F2_INT32

Sets the vertical height for each F2 field of the video signal. For progressive signals, it always has value 0.

DM_VIDEO_FILL_Y_REAL32

The luminance value for any pixel outside the clipping region. This is a real number: a value of 0.0 is the minimum legal value (black), 1.0 is the maximum legal value. Default is 0.

DM_VIDEO_FILL_A_REAL32

The alpha value for any pixel outside the clipping region. This is a real number: a value of 0.0 is the minimum (fully transparent), 1.0 is the maximum (fully opaque). Default is 1.0.

Examples

Here is an example that sets the video timing and colorspace for an HDTV signal:

```
DMpv message[3]
message[0].param = DM_VIDEO_TIMING_INT32
message[0].value.int32 = DM_TIMING_1125_1920x1080_5994;
message[1].param = DM_VIDEO_COLORSPACE_INT32;
message[1].value.int32 = DM_COLORSPACE_CbYCr_709_HEAD;
message[2].param = DM_END;
dmSetControls( device, message);
```

Image Parameters

This chapter describes in detail the dmSDK image parameters and gives examples of the resulting in-memory pixel formats.

Introduction

An *image buffer* is memory allocated for a frame or field of pixels. Since the dmSDK itself does not allocate memory for buffers, the application must do the allocation. This means that each buffer requires a dedicated memory allocation call (`malloc`, for example.)

Buffers must be in contiguous virtual memory and should be pinned in memory for optimum performance. Once a buffer has been created, the pointer to the buffer is passed to the dmSDK with the parameter `DM_IMAGE_BUFFER_POINTER`. Pointer to the first byte of an image buffer in memory. The buffer address must comply with the alignment constraints for buffers on the particular path or transcoder to which it is being sent. See `dmGetCapabilities` for details on determining alignment requirements with `DM_PATH_BUFFER_ALIGNMENT_INT32`. For example, if `DM_PATH_BUFFER_ALIGNMENT_INT32` is 8, this means that the value of the buffer pointer must be a multiple of 8 bytes. The same applies to `DM_PATH_COMPONENT_ALIGNMENT_INT32` where the beginning of each line (the first pixel of each line) must be a multiple of the value of the `DM_PATH_COMPONENTALIGNMENT_INT32` parameter.

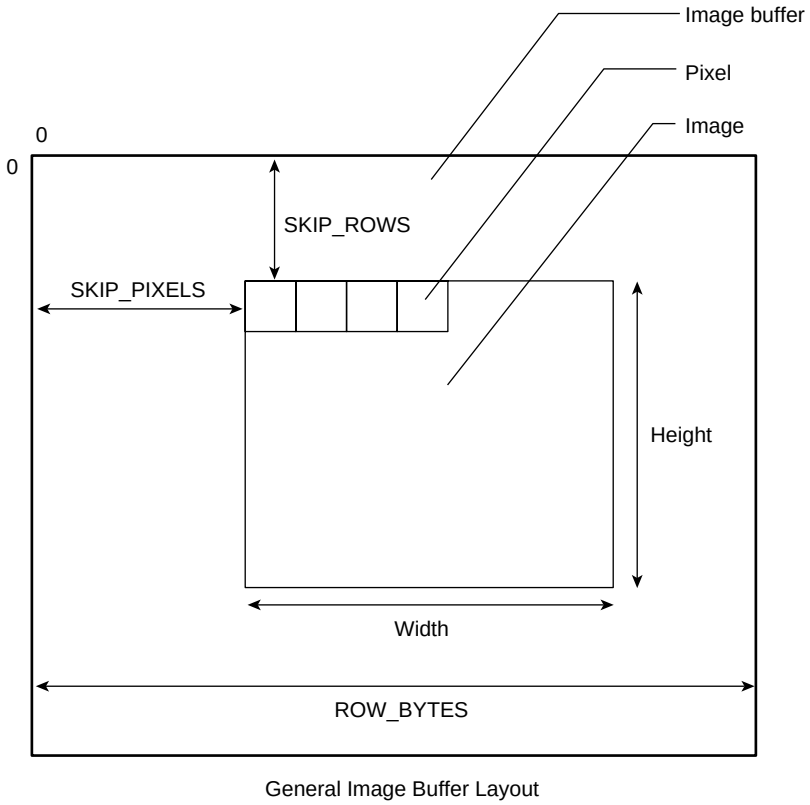

Figure 7-1 General Image Buffer Layout

In Figure 7-1, page 68 an image is mapped into a image buffer in a very general form.

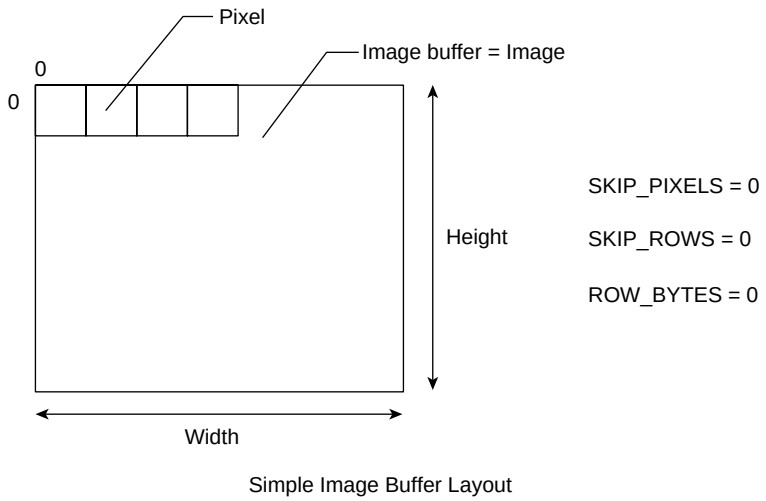

Figure 7-2 Simple Image Buffer Layout

Figure 7-2, page 69 shows the more common simple image buffer layout.

Image Buffer Parameters

The following subsections list and describe all image parameters.

DM_IMAGE_BUFFER_POINTER

Pointer to the first byte of an image buffer in memory. The buffer address must comply with the alignment constraints for buffers on the particular path or transcoder to which it is being sent. See `dmGetCapabilities` for details on determining alignment requirements with `DM_PATH_BUFFER_ALIGNMENT_INT32`. For example, if `DM_PATH_BUFFER_ALIGNMENT_INT32` is 8, this means that the value of the buffer pointer must be a multiple of 8 bytes. The same applies to `DM_PATH_COMPONENT_ALIGNMENT_INT32` where the beginning of each line (the first pixel of each line) must be a multiple of the value of the `DM_PATH_COMPONENT_ALIGNMENT_INT32` parameter.

DM_IMAGE_WIDTH_INT32

The width of the image in pixels.

DM_IMAGE_HEIGHT_1_INT32

For progressive or interleaved buffers (depending on parameter DM_IMAGE_INTERLEAVE_MODE_INT32), this represents the height of each frame. For interlaced and non-interleaved signals, this represents the height of each F1 field. Measured in pixels.

DM_IMAGE_HEIGHT_2_INT32

The height of each F2 field in an interlaced non-interleaved signal. Otherwise it has value 0.

DM_IMAGE_ROW_BYTES_INT32

The number of bytes along one row of the image buffer. If this value is 0, each row is exactly DM_IMAGE_WIDTH_INT32 pixels wide. Default is 0.

Note: In physical memory there is no notion of two dimensions, the end of the first row continues directly at the start of the second row. An image buffer contains either one frame or one field. For interlaced image data the two fields can be stored in two separate image buffers or they can be stored in interleaved form in one image buffer.

DM_IMAGE_SKIP_PIXELS_INT32

The number of pixels to skip at the start of each line in the image buffer. Default is 0. Must be 0 if DM_IMAGE_ROW_BYTES_INT32 is 0. Default is 0.

DM_IMAGE_SKIP_ROWS_INT32

The number of rows to skip at the start of each image buffer. Default is 0.

DM_IMAGE_TEMPORAL_SAMPLING_INT32

Specifies whether the image source is progressive or interlaced. Set to DM_TEMPORAL_SAMPLING_FIELD_BASED or DM_TEMPORAL_SAMPLING_PROGRESSIVE. Default is device-dependent. If the image data is field based, the parameter DM_IMAGE_INTERLEAVE_MODE_INT32 defines how the two fields are stored in an image buffer.

DM_IMAGE_INTERLEAVE_MODE_INT32

Only used in interlaced images. This parameter specifies whether the two fields have been interleaved into a single image (and reside in a single buffer) or are stored in two separate fields (hence in two separate buffers). Set to DM_INTERLEAVE_MODE_INTERLEAVED or DM_INTERLEAVE_MODE_SINGLE_FIELD. This is ignored for signals with progressive timing. Default is interleaved.

In DM_INTERLEAVE_MODE_INTERLEAVED each pair of fields is interleaved into a single buffer. In this case the parameter DM_IMAGE_HEIGHT_2_INT32 is set to zero.

For DM_INTERLEAVE_MODE_SINGLE_FIELD the two fields are stored separately. This means that each field has its own image buffer, use DM_IMAGE_HEIGHT_1_INT32 for the F1 buffer and DM_IMAGE_HEIGHT_2_INT32 for the F2 buffer.

DM_IMAGE_DOMINANCE_INT32

Sets the dominance of the video signal. The allowable values are DM_DOMINANCE_F1 (default), and DM_DOMINANCE_F2. Ignored for progressive signals. Field dominance defines the order of fields in a frame and can be either *F1–dominant* or *F2-dominant*. F1-dominant specifies a frame as an F1 field followed by an F2 field. F2-dominant specifies a frame as an F2 field followed by an F1 field. Notice also that for the same sequent of fields there are two valid interpretations which of the two fields belong together. Changing the field dominance is most significant when external devices (for example, a tape deck) can only operate on frame boundaries.

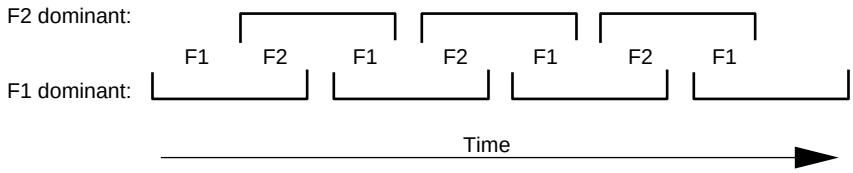

Figure 7-3 Field Dominance

DM_IMAGE_ORIENTATION_INT32

The orientation of the image.

DM_ORIENTATION_TOP_TO_BOTTOM "natural video order" pixel [0,0] is at the top left of the image.

DM_ORIENTATION_BOTTOM_TO_TOP "natural graphics order" pixel [0,0] is at the bottom left of the image.

DM_IMAGE_COMPRESSION_INT32

An image buffer can also store a compressed image, for example this could be the output of a codec. If the image data is compressed, then one of the following values are used:

```
DM_COMPRESSION_UNCOMPRESSED
DM_COMPRESSION_BASELINE_JPEG
DM_COMPRESSION_DV_625
DM_COMPRESSION_DV_525
DM_COMPRESSION_MPEG2I
DM_COMPRESSION_DVCPRO_625
DM_COMPRESSION_DVCPRO_525
DM_COMPRESSION_DVCPRO50_625
DM_COMPRESSION_DVCPRO50_525
DM_COMPRESSION_MPEG2
```

If the image data is in uncompressed format the value of this parameter is DM_COMPRESSION_UNCOMPRESSED.

Note: In case of a compressed bit stream, all parameters that describe the image data (that is, height, width, color space, etc.) might not be known. The only parameters that might be known are the compression type `DM_IMAGE_COMPRESSION_INT32` and the size of the bit stream `DM_IMAGE_SIZE_INT32`. The image buffer layout parameters (`DM_IMAGE_SKIP_ROWS`, `DM_IMAGE_SKIP_PIXELS`, and `DM_IMAGE_ROW_BYTES`) do not apply to compressed images.

For more infomation on JPEG, refer to W. B. Pennebaker and J. L. Mitchell, JPEG: Still Image Data Compression Standard, New York, NY: Van Nostrand Reinhold, 1993.

For more information on DV compression, refer to Specification of Consumer-Use Digital VCRs using 6.3mm magnetic tape.

For more information on DVCPRO and DVCPRO50 compression, refer to SMPTE 314M Television - Data Structure for DV-Based Audio, Data and Compressed Video - 25 and 50 Mb/s.

For more information on MPEG2, refer to ISO/IEC 13818-2 GENERIC CODING OF MOVING PICTURES AND ASSOCIATED AUDIO: SYSTEMS.

DM_IMAGE_SIZE_INT32

Size of the image buffer in bytes. This is a read-only parameter and is computed in the device using the current path control settings. This value represents the worst-case buffer size.

DM_IMAGE_COMPRESSION_FACTOR_REAL32

For compressed images only, this parameter describes desired compression factor. A value of `1` indicates no compression, a value of *x* indicates that approximately *x* compressed buffers require the same space as 1 uncompressed buffer.

Note: The size of the uncompressed buffer depends on image width, height, packing and sampling. The default value is implementation-dependent, but should represent a reasonable trade-off between compression time, quality and bandwidth. *x* is a number larger than 1.

DM_IMAGE_PACKING_INT32

For recommendations on packing and component ordering, see Appendix A: "Pixels in Memory."

The image packing parameter describes the pixel storage in detail as follows:

`DM_PACKING_`*type_size_order*

- *type* is the base type of each component. Leave blank for an unsigned integer, use `S` for a signed integer. (In the future, the dmSDK may also support `R` for real numbers.)
- *size* defines the number of bits per component. The size may refer to simple, padded or complex packings.

  For the simplest formats every component is the same size and there is no additional space between components. Here, a single numeric value specifies the number of bits per component. The first component consumes the first *size* bits, the next consumes the next *size* bits, and so on. Within each component, the most significant bits always precede the least-significant bits. For example, a size of `12` means that the first byte in memory has the most significant 8 bits of the first component, the second byte holds the remainder of the first component and the most significant 4 bits of the second component, and so on.

  Space is only allocated for components which are in use (that depends on the sampling mode, see later). For these formats the data must always be interpreted as a sequence of bytes. For example, `DM_PACKING_8` describes a packing in which each component is an unsigned 8–bit quantity. `DM_PACKING_S8` describes the same packing except that each component is a signed 8–bit quantity.

  For padded formats, each component is padded and may be treated as a short 2–byte integer. When this occurs, the *size* takes the form: {*bits*}in{*size*}{*alignment*} where:

bits	is the number of bits of space per component
space	is the total size of each component
alignment	`L` or `R` indicates, respectively, whether the information is left or right-shifted in that space

  In this case, each component in use consumes *space* bits and those bits must be interpreted as a short integer. (Unused components consume no space).

For example, here are some common packings (note that the signed-ness of the component values does matter):

```
          15  int  short 0
Packing   +--------------+
12in16R   0000iiiiiiiiiiii
S12in16R  ssssiiiiiiiiiiii
12in16L   iiiiiiiiiiiipppp
S12in16L  iiiiiiiiiiiipppp
S12in16L0 iiiiiiiiiiii0000
```

where `s` indicates sign-extension, `i` indicates the actual component information, and `p` indicates padding (replicated from the most significant bits of information).

Note: These bit locations refer to the locations when the 16–bit component has been loaded into a register as a 16–bit integer quantity.

For the most complex formats, the size of every component is specified explicitly, and the entire pixel must be treated as a single 4–byte integer. The *size* takes the form *size1_size2_size3_size4*, where *size1* is the size of component 1, *size2* is the size of component 2, and so on. In this case, the entire pixel is a single 4–byte integer of length equal to the sum of the component sizes. Any space allocated to unused components must be zero-filled. The most common complex packing occurs when 4 components are packed within a 4-byte integer. For example, `DM_PACKING_10_10_10_2` is:

```
            31             int             0
Packing     +-------------------------------+
10_10_10_2  11111111112222222222333333333344
```

where `1` is the first component, `2` is the second component, and so on. The bit locations refer to the locations when this 32–bit pixel is loaded into a register as a 32–bit integer quantity. If only three components were in use (determined from the sampling), then the space for the fourth component would be zero-filled.

- *order* is the order of the components in memory. Leave blank for natural ordering (1,2,3,4), use `R` for reversed ordering (4,3,2,1). For all other orderings, specify the component order explicitly. For example, `4123` indicates that the fourth

component is stored first in memory, followed by the remaining three components. Here, we compare a normal, a reversed, and a 4123 packing:

```
             31             int             0
Packing      +-----------------------------+
10_10_10_2   11111111112222222223333333333344
10_10_10_2_R 44333333333322222222211111
10_10_10_2_4123 44111111111122222222233333333333
```

where `1` is the first component, `2` is the second component, and so on. Since this is a complex packing, the bit locations refer to the locations when this entire pixel is loaded into a register as a single integer.

DM_IMAGE_COLORSPACE_INT32

The colorspace parameters describe how to interpret each component. The full colorspace parameter is:

`DM_COLORSPACE_`*representation_standard_range*

where:

- *representation* is either `DM_REPRESENTATION_RGB` or `DM_REPRESENTATION_CbYCr`.

  This controls how to interpret each component. Table 7-1, page 76 shows this mapping (assuming for now that every component is sampled once per pixel):

Table 7-1 Mapping colorspace *representation* parameters

Colorspace Representation	Component 1	Component 2	Component 3	Component 4
RGB	Red	Green	Blue	Alpha
CbYCr	Cb	Y	Cr	Alpha

Remember, the packing dictates the size and order of the components in memory, while the colorspace describes what each component represents. For example, here we show the effect of colorspace and packing combined (again assuming a 4444 sampling, see later).

```
Color                    31            int              0
Standard  Packing        +-----------------------------+
RGB       10_10_10_2     RRRRRRRRRRGGGGGGGGGGBBBBBBBBBBAA
RGB       10_10_10_2_R   AABBBBBBBBBBGGGGGGGGGGRRRRRRRRRR
CbYCr     10_10_10_2     bbbbbbbbbbYYYYYYYYYYrrrrrrrrrrAA
CbYCr     10_10_10_2_R   AAbbbbbbbbbbYYYYYYYYYYrrrrrrrrrr
```

- *standard* indicates how to interpret particular values as actual colors. Choosing a different standard alters the way the system converts between different color representations. The current standards supported are Rec. 601, Rec. 709 and SMPTE 240M.

- *range* is either `FULL`, where the smallest and largest values are limited only by the available packing size, or `HEAD`, where the smallest and largest values are somewhat less than the theoretical min/max values to allow some "headroom". Full range is common in computer graphics. Headroom range is common in video, particularly when sending video signals over a wire (for example, values outside the legal component range may be used to mark the start or end of a video frame). When constructing a colorspace, you must specify a representation, a standard and a range.

  In Rec. 601 video, the black level (blackest black) is 16 for 8–bit video and 64 for 10–bit video, but in computer graphics, 0 is blackest black. If a picture with 16 for blackest black is displayed by a system that uses 0 as blackest black, the image colors are all grayed-out as a result of shifting the colors to this new scale. Similarly, the brightest level is 235 for 8–bit video and 940 for 10–bit video. The best results are obtained by choosing the correct colorspace.

Example 7-1 `DM_COLORSPACE_RGB_709_FULL`

`DM_COLORSPACE_RGB_709_FULL` is shorthand for the following:

```
DM_REPRESENTATION_RGB
+
DM_STANDARD_709
+
DM_RANGE_FULL
```

where:

- representation is RGB
- the standard is 709
- full-range data is used

DM_IMAGE_SAMPLING_INT32

The sampling parameters take their names from common terminology in the video industry. They describe how often each component is sampled for each pixel. In computer graphics, its normal for every component to be sampled once per pixel, but in video that need not be the case.

For all RGB colorspaces, the only legal samplings are:

- DM_SAMPLING_444 indicates that the R, G and B components are each sampled once per pixel, and only the first 3 channels are used. If used with an image packing that provides space for a fourth component, then those bits should have value 0 on an input path and will be ignored on an output path.
- DM_SAMPLING_4444 indicates that the R, G, B and A components are sampled once per pixel.

For all CbYCr colorspaces, the legal samplings include:

- DM_SAMPLING_444 indicates that Cb, Y, and Cr are each sampled once per pixel and only the first 3 channels are used. If a packing provides space for a 4th channel then those bits should have value 0.
- DM_SAMPLING_4444 indicates that Cb, Y, Cr and Alpha are each sampled once per pixel.
- DM_SAMPLING_422 indicates that Y is sampled once per pixel and Cb/Cr are sampled once per pair of pixels. In this case Cb and Cr are interleaved on component 1 (Cb is first, Cr is second) and the Y occupies component 2. If used with an image packing that provides space for a third or fourth component, then those bits should have value 0 on an input path and will be ignored on an output path.
- DM_SAMPLING_4224 indicates that Y and Alpha are sampled once per pixel and Cb/Cr are sampled once per pair of pixels. In this case Cb and Cr are interleaved on component 1, Y is on component 2, component 3 contains the alpha channel

and component 4 is not used (and will have value `0` if space is allocated for it in the packing).

- `DM_SAMPLING_411` indicates that Y is sampled once per pixel and Cb/Cr are sampled once per 4 pixels. In this case Cb, Y is component 2 and Cr occupies component 3. If used with an image packing that provides space for a 4th component then those bits should have value 0 on an input path and will be ignored on an output path.
- `DM_SAMPLING_420` indicates that Y is sampled once per pixel and Cb or Cr is sampled once per pair of pixels on alternate lines. In this case Cb or Cr is interleaved on component 1 and the Y occupies component 2. If used with an image packing that provides space for a 3rd or 4th component then those bits should have value 0 on an input path and will be ignored on an output path.
- `DM_SAMPLING_400` indicates that only Y is sampled per pixel (a greyscale image). Here Y is stored on component 1, all other components are unused. If used with an image packing that provides space for additional components, then those bits should have value 0 on an input path and will be ignored on an output path.
- `DM_SAMPLING_0004` indicates that only Alpha is sampled per pixel. If used with an image packing that provides space for additional components, then those bits should have value 0 on an input path and will be ignored for an output path.

Table 7-2, page 79 shows the combined effect of sampling and colorspace on the component definitions:

Table 7-2 Effect of sampling and colorspace on component definitions.

Sampling	Colorspace Representation	Comp 1	Comp 2	Comp 3	Comp 4
4444	RGB	Red	Green	Blue	Alpha
444	RGB	Red	Green	Blue	
0004	RGB	Alpha	Y	Cr	Alpha
444	CbYCr	Cb	Y	Cr	0

4224	CbYCr	Cb/Cr	Y	Alpha	0
422	CbYCr	Cb/Cr	Y		
400	CbYCr	Y			
420	CbYCr	Cb/Cr*	Y		
411	CbYCr	Y	Cr		
0004	CbYCr	Alpha			

1

DM_SWAP_BYTES_INT32

Parameter `DM_IMAGE_SWAP_BYTES` may be available on some devices. When set to `0` (the default) this has no effect. When set to `1`, the device reorders bytes as a first step when reading data from memory, and as a final step when writing data to memory. The exact reordering depends on the packing element size. For simple and padded packing formats (see packings, below) the element size is the size of each component. For complex packing formats, the element size is the sum of the four component sizes.

The swap-bytes parameter reorders bits as follows:

Element Size	Default ordering	Modified ordering
16 bit	[15..0]	[7..0][15..8]
32 bit	[31..0]	[7..0][15..8][23..16][31..24]
other	[n..0]	[n..0] (no change)

1 * Cb and Cr components are multiplexed with Y on alternate lines (not pixels.)

Audio Parameters

This chapter describes the dmSDK audio parameters and buffers.

Audio Buffer Layout

The digital representation of an audio signal is generated by periodically sampling the amplitude (voltage) of the audio signal. The samples represent periodic "snapshots" of the signal amplitude. The sampling rate specifies the number of samples per second. The audio buffer pointer points to the source or destination data in an audio buffer for processing a fragment of a media stream. For audio signals, a fragment typically corresponds to between 10 milliseconds and 1 second of audio data. An audio buffer is a collection of sample frames. A *sample frame* is a set of audio samples that are coincident in time. A sample frame for mono data is a single sample. A sample frame for stereo data consists of a left-right sample pair.

Stereo samples are interleaved; left-channel samples alternate with right-channel samples. 4-channel samples are also interleaved, with each frame usually having two left/right sample pairs, but there can be other arrangements.

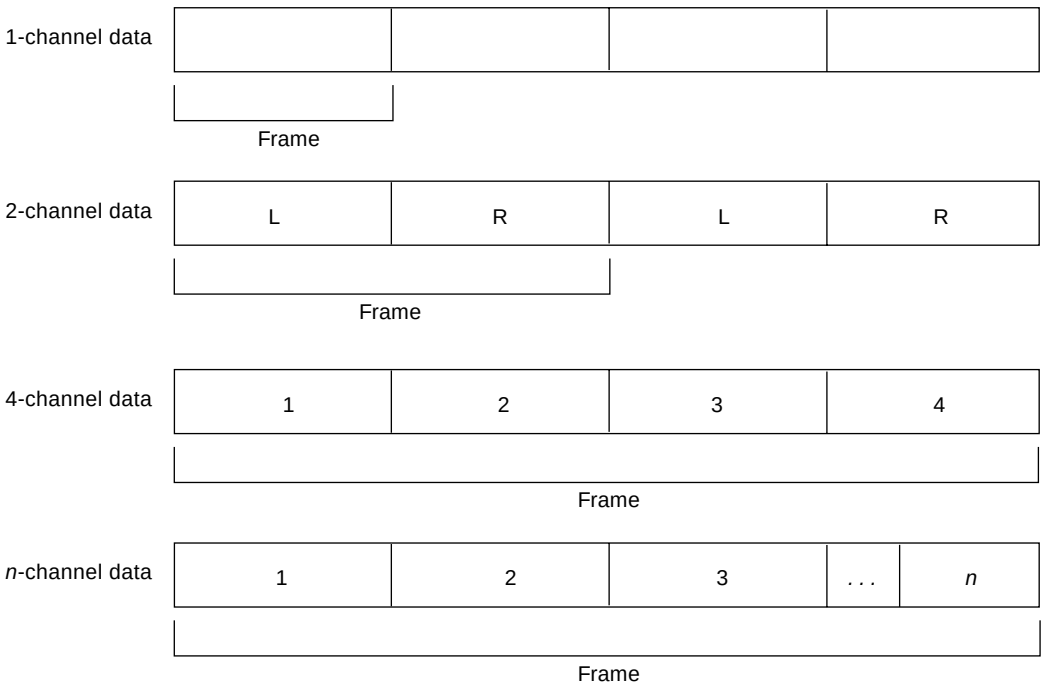

Figure 8-1 Different Audio Sample Frames

Figure 8-1, page 82 shows the relationship between the number of channels and the frame size of audio sample data.

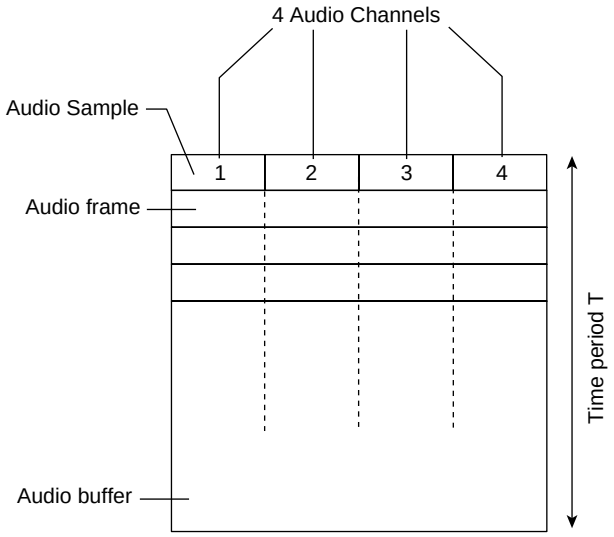

Figure 8-2 Layout of an Audio Buffer with 4 Channels

Figure 8-2, page 83 shows the layout of an audio buffer in memory.

Audio Parameters

The parameters discussed in the following sections are as follows:

`DM_AUDIO_BUFFER_POINTER`	Pointer to the audio buffer
`DM_AUDIO_FRAMESIZE_INT32`	Size of a audio sample frame in bytes
`DM_AUDIO_SAMPLE_RATE_REAL64`	Sample rate in Hz
`DM_AUDIO_PRECISION_INT32`	Precision at audio jack
`DM_AUDIO_FORMAT_INT32`	Format of the data in the audio buffer
`DM_AUDIO_GAINS_REAL64_ARRAY`	Audio gain controls
`DM_AUDIO_COMPANDING_INT32`	Sample quantization method
`DM_AUDIO_CHANNELS_INT32`	Number of audio channels

`DM_AUDIO_COMPRESSION_INT32`	Audio compression format

DM_AUDIO_BUFFER_POINTER

A pointer to the first byte of an in-memory audio buffer. The buffer address must comply with the alignment constraints for buffers on the particular path to which it is being sent. (See `dmGetCapabilities`(3dm) for details of determining alignment requirements).

DM_AUDIO_FRAME_SIZE_INT32

The size of an audio sample frame in bytes. This is a read-only parameter and is computed in the device using the current path control settings.

DM_AUDIO_SAMPLE_RATE_REAL64

The sample rate of the audio data in Hz. The sample rate is the frequency at which samples are taken from the analog signal. Sample rates are measured in hertz (Hz). A sample rate of 1 Hz is equal to one sample per second. For example, when a mono analog audio signal is digitized at a 44.1 kilohertz (kHz) sample rate, 44,100 digital samples are generated for every second of the signal. Values are dependent on the hardware, but are usually between 8,000.0 and 96,000.0. Default is hardware-specific. Common sample rates are:

8,000.0
16,000.0
32,000.0
44,100.0
48,000.0
96,000.0

The Nyquist theorem defines the minimum sampling frequency required to accurately represent the information of an analog signal with a given bandwidth. According to Nyquist, digital audio information is sampled at a frequency that is at least double the highest interesting analog audio frequency. The sample rate used for music-quality audio, such as the digital data stored on audio CDs is 44.1 kHz. A 44.1 kHz digital signal can theoretically represent audio frequencies from 0 kHz to 22.05 kHz, which adequately represents sounds within the range of normal human hearing.

Higher sample rates result in higher-quality digital signals; however, the higher the sample rate, the greater the signal storage requirement.

DM_AUDIO_PRECISION_INT32

The maximum width in bits for an audio sample at the input or output jack. For example, a value of 16 indicates a 16-bit audio signal. Query only. DM_AUDIO_PRECISION_INT32 specifies the precision at the Audio I/O jack, whereas DM_AUDIO_FORMAT_INT32 specifies the packing of the audio samples in the audio buffer. If DM_AUDIO_FORMAT_INT32 is different than DM_AUDIO_PRECISION_INT32, the system will convert between the two formats. Such a conversion might include padding and/or truncation.

DM_AUDIO_FORMAT_INT32

Specifies the format in which audio samples are stored in memory. The interpretation of format values is:

DM_FORMAT_[*type*][*bits*]

- [*type*] is U for unsigned integer samples, S for signed (2's compliment) integer samples, R for real (floating point) samples.
- [*bits*] is the number of significant bits per sample.

For sample formats in which the number of significant bits is less than the number of bits in which the sample is stored, the format of the values is:

DM_FORMAT_*{type}{bits}* in*{size}{alignment}*

- *{size}* is the total size used for the sample in memory, in bits.
- *{alignment}* is either R or L depending on whether the significant bits are right- or left-shifted within the sample. For example, here are three of the most common audio buffer formats:

```
DM_FORMAT_U8          7 char 0
                      +------+
                      iiiiiiii

DM_FORMAT_S16         15  short int  0
                      +--------------+
                      iiiiiiiiiiiiiiii
```

```
DM_FORMAT_S24in32R     31            int              0
                       +------------------------------+
                       ssssssssiiiiiiiiiiiiiiiiiiiiiiii
```

where `s` indicates sign-extension, and `i` indicates the actual component information. The bit locations refer to the locations when the 8–, 16–, or 32–bit sample has been loaded into a register as an integer quantity. If the audio data compression parameter `DM_AUDIO_COMPRESSION_INT32` indicates that the audio data is in compressed form, the `DM_AUDIO_FORMAT_INT32` indicates the data type of the samples after decoding. Common formats are:

`DM_FORMAT_U8`
`DM_FORMAT_S16`
`DM_FORMAT_S24in32R`
`DM_FORMAT_R32`

Default is hardware-specific.

DM_AUDIO_GAINS_REAL64_ARRAY

The gain factor in decibels (dB) on the given path. There will be a value for each audio channel. Negative values represent attenuation. Zero represents no change of the signal. Positive values amplify the signal. A gain of negative infinity indicates infinite attenuation (mute).

DM_AUDIO_COMPANDING_INT32

Describes the quantization method of the audio sample value. For `DM_COMPANDING_MU_LAW` and `DM_COMPANDING_A_LAW`, the output voltage changes exponentially with linear sample values changes. The purpose of this method is to use a wider dynamic volume range with the same number of sample bits. *Companding* is a neologism that combines “compressing” and “expanding”. It is different than Audio Compression, where a set of audio samples are compressed in order to get a smaller file size.

Common values are:

`DM_COMPANDING_NONE` (default, if supported by the hardware)
`DM_COMPANDING_MU_LAW`
`DM_COMPANDING_A_LAW`

DM_AUDIO_CHANNELS_INT32

The number of channels of audio data in the buffer. Multi-channel audio data is always stored interleaved, with the samples for each consecutive audio channel following one another in sequence. For example, a 4–channel audio stream will have the form:

```
123412341234...
```

where 1 is the sample for the first audio channel, 2 is for the second, and so on.

Common values are:

```
DM_CHANNELS_MONO
DM_CHANNELS_STEREO
DM_CHANNELS_4
DM_CHANNELS_8
```

DM_AUDIO_COMPRESSION_INT32

In case the audio data is in compressed form, this parameter specifies the compression format. The compression format may be an industry standard such as MPEG-1 audio, or it may be no compression at all.

Common values include the following:

```
DM_COMPRESSION_UNCOMPRESSED
DM_COMPRESSION_MU_LAW
DM_COMPRESSION_A_LAW
DM_COMPRESSION_IMA_ADPCM
DM_COMPRESSION_MPEG1
DM_COMPRESSION_MPEG2
DM_COMPRESSION_AC3
```

When the data is uncompressed, the value of this parameter is `DM_COMPRESSION_UNCOMPRESSED.`

Uncompressed Audio Buffer Size Computation

The following equation shows how to calculate the number of bytes for an uncompressed audio buffer given the sample frame size, sampling rate and the time period representing the audio buffer:

$N = F \cdot R \cdot T$

where:

N	audio buffer size in bytes
F	the number of bytes per audio sample frame (`DM_AUDIO_FRAMESIZE_INT32`)
R	the sample rate in Hz (`DM_AUDIO_SAMPLE_RATE_REAL64`)
T	the time period the audio buffer represents in seconds

Example 8-1 Buffer Size Computation

If:

- *F* is 4 bytes (if *packing* is S16 and there are two channels)
- *R* (sample rate) is 44,100 Hz
- *T* = 40 ms = 0.04 s.

then the resulting buffer size (*N*) is 7056 bytes.

DM Processing

The DM library is concerned with two types of interfaces: Paths for digital media through jacks into and out of the machine, and pipes for digital media to and from transcoders. Both share common control, buffer, and queueing mechanisms. These mechanisms are first described in the context of a complete program example. Subsequently, the individual functions are presented.

DM Program Structure

DM programs are composed of the following structure. Each of the functions are described later in this chapter (except where noted).

```
// get list of available dmedia devices
dmGetCapabilities( systemid,  );

// search the devices to find the desired jack, path, or xcode to open
 // (See Chapter 7: DM Capabilities for function description)
dmGetCapabilities( deviceid, & capabilities  );

 // query the jack, path, or xcode to discover allowable open options and parameters
 // (See Chapter 7: DM Capabilities for function description)
dmGetCapabilities( objectid, & capabilities  );
 // query for individual parameter characteristics
 // (See Chapter 7: DM Capabilities for function description)
dmPvGetCapabilities( deviceid,& capabilities  );

// free memory associated with any of the above get capabilities:
 // (See Chapter 7: DM Capabilities for function description)
dmFreeCapabilities( capabilities );

 // open a logical connection todmhe desired object
dmOpen( objectId, options,  );

 // get and set any necessary immediate controls
dmGetControls( openid, controls );
dmSetControls( openid, controls );
```

```
 // send any synchronous controls
dmSendControls( openid, controls );

// pre-roll buffers
dmSendBuffers( openid, buffers );

 // prepare for asynchronous processing by getting a wait handle
dmGetWaitHandle( openid,  );

 // start the path or xcode transferring
dmBeginTransfer( openid );

// perform synchronous work
dmXcodeWork( openid );

// check on the status of the queues
dmGetSendMessageCount( openid,  );
dmGetReceiveMessageCount( openid,  );

// process return messages
dmReceiveMessage( openid,  );

// find specific returned parameters
dmPvFind( msg, param );

// repeat dmSendControls, dmSendBuffers, dmXcodeWork, etc. as required

// stop the transfer
dmEndTransfer( openid );

// close the logical connection
dmClose( openid );

// other useful functions:
dmGetVersion( ,  );
dmGetSystemUST( systemId,  );
dmStatusName( status );
dmMessageName( messageType );
```

DMstatus Return Value

Note that all DM API functions return an `DMstatus` value. This provides a consistent error checking interface. (Certain "Convenience Functions" do not adhere to this standard.) See descriptions below. The various DMstatus return values are:

`DM_STATUS_NO_ERROR`

The operation succeeded without error.

`DM_STATUS_NO_OPERATION`

The function resulted in no operation.

`DM_STATUS_OUT_OF_MEMORY`

The operation was aborted due to lack of memory resources.

`DM_STATUS_INVALID_ID`

One of the arguments representing an ID is invalid.

`DM_STATUS_INVALID_ARGUMENT`

One of the arguments in the function call is invalid.

`DM_STATUS_INVALID_VALUE`

The value of a parameter is invalid.

`DM_STATUS_INVALID_PARAMETER`

The specified parameter ("param" field) is invalid for the requested operation.

`DM_STATUS_INVALID_CONFIGURATION`

Since control messages may be incomplete, and each individual set or send controls may be valid, there exists a point in time where the processing of buffers must be accomplished using those aggregate controls. If for some reason, the "combination of controls" is invalid, the processing is aborted and the `DM_STATUS_INVALID_CONFIGURATION` error (for `dmSetControls`) or the event (for `dmSendControls`) is returned

`DM_STATUS_RECEIVE_QUEUE_EMPTY`

The receive queue was empty when an dmReceiveMessage function was processed.

`DM_STATUS_SEND_QUEUE_OVERFLOW`

Too many dmSendControls and/or dmSendBuffers have been issued.

`DM_STATUS_RECEIVE_QUEUE_OVERFLOW`

The receive queue will not accept the return message if the current message is enqueued on the send queue.

`DM_STATUS_INSUFFICIENT_RESOURCES`

Not all the resources required to complete the operation are available.

`DM_STATUS_DEVICE_UNAVAILABLE`

The requested device has become unavailable, possibly by being powered down or removed from the system.

`DM_STATUS_ACCESS_DENIED`

The requested open access conflicts with a previous access already established or the requested parameter cannot be modified during the current operation mode.

`DM_STATUS_INTERNAL_ERROR`

An operation was aborted due to a system or device I/O error.

Device States

For audio and video paths and transcoders, the device transitions through well-known states, known as *Device States*. These states are listed below:

`DM_DEVICE_STATE_READY`

Indicates that the device is in a quiescent state and can accept messages, but will not process them until it enters the `DM_DEVICE_STATE_TRANSFERRING` state.

`DM_DEVICE_STATE_TRANSFERRING`

Indicates that the device has accepted a `dmBeginTRansfer` and is now processing messages.

`DM_DEVICE_STATE_WAITING`

Indicates that the device is currently waiting for an external event such as the `DM_WAIT_FOR_AUDIO_MSC_INT64` predicate control. Messages may still be enqueued, but will not be processed until the wait condition is removed.

DM_DEVICE_STATE_ABORTING

Indicates that the device has terminated message processing, usually by accepting a dmEndTransfer. All messages remaining on the input queue will be flushed to the output queue with the message type indicating that the message was aborted. (DM_CONTROLS_ABORTED, DM_QUERY_CONTROLS_ABORTED, and DM_BUFFERS_ABORTED.)

DM_DEVICE_STATE_FINISHING

Indicates that the device is terminating the transfer, but will complete processing of the remaining messages in the input queue.

Opening a Jack, Path or Xcode

In order to communicate with a Jack, Path, or Xcode, a connection must be opened. A physical device (e.g. a PCI card) may simultaneously support several such connections. These connections are done by calling dmOpen:

```
DMstatus dmOpen (const DMint64 objectId, DMpv* options,DDMopenid* openid);
```

objectId is the 64-bit unique identifier for the object (jack, path or transcoder) to be opened. The parameters in options specify the initial configuration of the device to be opened. These parameters are described in Table 10.1, where the string in the Parameter column is a shortened form of the full parameter name. The full parameter name is of the form DM_parameter_type, where parameter and type are the strings listed in the Parameter and Type columns respectively. For example, the full parameter name of OPEN_MODE is DM_OPEN_MODE_INT32.

STATUS RETURN

This function returns one of the following:

DM_STATUS_NO_ERROR

The call succeeded and the handle of the open instance of the object has been returned in openid.

DM_STATUS_INVALID_ID

The argument objectid is invalid.

DM_STATUS_INVALID_ARGUMENT

One of the arguments is otherwise invalid.

`DM_STATUS_INVALID_PARAMETER`

One of the parameters in the options list is invalid.

`DM_STATUS_INVALID_VALUE`

One of the parameters in the options list has an invalid value.

`DM_STATUS_OUT_OF_MEMORY`

Insufficient memory is available to perform the operation, including the space needed to allocate the queues for messages between the application and the device.

`DM_STATUS_INSUFFICIENT_RESOURCES`

Some other required resource is not available, possibly by being already in use by this or another application.

`DM_STATUS_DEVICE_UNAVAILABLE`

The requested device has gone off-line, possibly by being disconnected.

`DM_STATUS_ACCESS_DENIED`

The requested open access mode is not available.

`DM_STATUS_INTERNAL_ERROR`

An operating system error has occurred.

Jack Open Parameters

The following open parameters are supported when opening a jack.

Table 9-1 Jack, dmOpen Options

Parameter	Type	Description
OPEN_MODE	INT32	Application's intended use for the device. Defined values are: DM_MODE_RO for read only access. DM_MODE_RWS for shared read/write access. DM_MODE_RWE for exclusive access. The default is defined by the device's capabilities.
OPEN _RECEIVE _QUEUE_COUNT	INT32	Applications' preferred size (number of messages) for the receive queue. This influences the amount of memory allocated for this queue when the device is opened. Default is device-dependent A null value indicates that the application does not expect to receive any events from the jack.
OPEN _EVENT _PAYLOAD_COUNT	INT32	Application's preferred size (number of messages) for the queue event payload area. This payload area holds the contents of event messages on the receive queue. Default is device dependent. A null value indicates that the application does not expect to receive any events from the jack.

The DM_OPEN_OPTION_IDS returned by a dmGetCapabilities call using the JACK ID, returns a list of these parameters. dmPvGetCapabilities can then be used to discover allowable values.

Path Open Parameters

The following open parameters are supported when opening a path: The DM_OPEN_OPTION_IDS

Table 9-2 dmOpen Options

Parameter	Type	Description
OPEN_MODE	INT32	Application's intended use for the device. Defined values are: DM_MODE_RO for read only access DM_MODE_RWS for shared read/write access DM_MODE_RWE for exclusive access. The default is defined by the device's capabilities.
OPEN _SEND _QUEUE_COUNT	INT32	Application's preferred size (number of messages) for the send queue. This influences the amount of memory allocated for this queue when the device is opened. Default is device-dependent.
OPEN _RECEIVE _QUEUE_COUNT	INT32	Applications' preferred size (number of messages) for the receive queue. This influences the amount of memory allocated for this queue when the device is opened. Default is device-dependent
OPEN _MESSAGE _PAYLOAD_SIZE	INT32	Application's preferred size (in bytes) for the queue message payload area. The payload area holds messages on both the send and receive queues. Default is device-dependent.

Parameter	Type	Description
OPEN_EVENT_PAYLOAD_COUNT	INT32	Application's preferred size (number of messages) for the queue event payload area. This payload area holds the contents of event messages on the receive queue. Default is device-dependent.
OPEN_SEND_SIGNAL_COUNT	INT32	Application's preferred low-water level (number of empty message slots) in the send queue. When the device dequeues a message and causes the number of empty slots to exceed this level, then the device will signal the send queue event. Default is device-dependent.

Xcode Open Parameters

Table 9-3 dmOpen Options

Parameter	Type	Description
OPEN_MODE	INT32	Application's intended use for the device. Defined values are: DM_MODE_RO for read only access DM_MODE_RWS for shared read/write access DM_MODE_RWE for exclusive access. The default is defined by the device's capabilities.
OPEN _SEND _QUEUE_COUNT	INT32	Application's preferred size (number of messages) for the send queue. This influences the amount of memory allocated for this queue when the device is opened. Default is device-dependent.
OPEN _RECEIVE _QUEUE_COUNT	INT32	Applications' preferred size (number of messages) for the receive queue. This influences the amount of memory allocated for this queue when the device is opened. Default is device-dependent
OPEN _MESSAGE _PAYLOAD_SIZE	INT32	Application's preferred size (in bytes) for the queue message payload area. The payload area holds messages on both the send and receive queues. Default is device-dependent.

Parameter	Type	Description
OPEN _EVENT _PAYLOAD_COUNT	INT32	Application's preferred size (number of messages) for the queue event payload area. This payload area holds the contents of event messages on the receive queue. Default is device-dependent.
OPEN _SEND _SIGNAL_COUNT	INT32	Application's preferred low-water level (number of empty message slots) in the send queue. When the device dequeues a message and causes the number of empty slots to exceed this level, then the device will signal the send queue event. Default is device-dependent.

Parameter	Type	Description
OPEN_XCODE_MODE	INT32	Application's preferred mode for controlling a software transcoder. This parameter does not apply to paths. Defined values are: DM_XCODE_MODE_SYNCHRONOUS when processing by a software transcoder is to be initiated by the application. DM_XCODE_MODE_AYNCHRONOUS when processing by a software transcoder is to be initiated by DM. Default is DM_XCODE_MODE_ASYNCHRONOUS.
OPEN_XCODE_STREAM	INT32	Selects between single and multi-stream transcoders. In single stream mode, source and destination buffers are processed at the same rate. In multi-stream mode, the source and destination pipes each have their own queue of buffers and may run at different rates (this is more complicated to program, but may be more efficient for some intra-frame codecs). Defined values are: DM_XCODE_STREAM_SINGLE DM_XCODE_STREAM_MULTI Default is DM_XCODE_STREAM_SINGLE In a future release, DM_XCODE_STREAM_MULTI transcoders will be supported.

Set Controls

Some controls on a logical connection are "asynchronous" in nature and do not affect an ongoing data transfer. These controls may be set in an "out of band" message using the `dmSetControls` operation:

```
DMstatus dmSetControls( DMopenid openid, DMpv* controls );
```

`openid` is the 64-bit unique identifier returned by the `dmOpen function`. The controls parameter is a message containing various parameters as described elsewhere in this document. Note that this call blocks until the device has processed the message. To identify an invalid value specification, the device will set the length component of the erroneous `DMpv` to -1, otherwise the controls array will not be altered in any way and may be reused. The controls message is not enqueued on the send queue but instead is sent directly to the device. The device will attempt to process the message "as soon as possible".

Enqueueing entails a copy operation,

`NOTES`

This call returns as soon as the control array has been processed. This does not mean that buffers have been affected by the parameter change. Rather, it means that the parameters have been validated and sent to the device (i.e. in most cases this means that they reside in registers).

`STATUS RETURN`

This function returns one of the following:

`DM_STATUS_NO_ERROR`

The control values were set successfully.

`DM_STATUS_INVALID_ID`

The specified `openid` is invalid.

`DM_STATUS_INVALID_PARAMETER`

At least one of the parameters in the controls array was not recognized (the first such offending control will be marked with length -1, remaining controls will be skipped and the entire message will be ignored).

`DM_STATUS_INVALID_VALUE`

At least one of the parameters in the controls array has a value which is invalid. This may be because the parameter value is outside the legal range, or it may be that parameter value is inconsistent (the entire message will be ignored and the system will attempt to flag the first offending value by setting the length to -1).

Get Controls

Control on a logical connection may be queried asynchronously to an ongoing transfer:

```
DMstatus dmGetControls (DMopenid openid, DMpv* controls);
```

`openid` is the identifier, returned by `dmOpen`, of the jack, path, or transcoder whose parameters are to be queried. The controls parameter is a message consisting of parameters to be queried. The device will place its reply in the controls array argument (overwriting existing values). Control values that were obtained successfully will have non-negative lengths. `GetControls` returns the state of the controls at the time the call is made. If `GetControls` is called before a control has been explicitly set, then generally the returned value is undefined (exceptions are noted in the definitions the controls, see `DM_UST`).

`STATUS RETURN`

This function returns one of the following:

`DM_STATUS_NO_ERROR` The control values were obtained successfully.

`DM_STATUS_INVALID_ID` The specified open device id is invalid.

`DM_STATUS_INVALID_PARAMETER` At least one of the parameters in the controls array was not recognized (the offending control will be marked with length –1; remaining controls will still be processed).

`DM_STATUS_INVALID_VALUE` At least one of the parameters in the controls array has a value which is invalid (the offending control will be marked with length –1; remaining controls will still be processed).

Send Controls

Other controls on a logical connection are "synchronous" in nature and do affect the processing of subsequent data buffers. These controls should be set in an "in band" message using the dmSendControls operation:

```
DMstatus dmSendControls( DMopenid openid, DMpv* controls );
```

`openid` is the 64-bit unique identifier returned by the `dmOpen` function. The controls parameter is a message containing various parameters as described in the preceding chapters.

The dmSendControls sends a message containing a list of control parameters to a previously-opened digital media device. These controls are enqueued on the send queue in sequence with any other messages to that device. Any control changes are thus guaranteed not to have any effect until all previously enqueued messages have been processed.

This call returns as soon as the control change has been enqueued to the device. It does not wait until the control change has actually taken effect.

All the control changes within a single message are considered occur atomically. If any one control change in the message fails, then the entire message has no effect. A successful return does not guarantee that resources will be available to support the requested control change at the time it is processed by the device.

As each message is processed by the device, a reply message will be enqueued for return to the application. By examining that reply, the application may obtain the result of attempting to process the requested controls. Note that a device may take an arbitrarily long time to generate a reply (it may, for example, wait for several messages before replying to the first). If an application requires an immediate response, consider using the set controls operation instead.

Enqueueing entails a copy operation, so the application is free to delete/alter the message array as soon as the call returns. Any error return value indicates the control change has not been enqueued and will thus have no effect.

`STATUS RETURN`

This function returns one of the following:

`DM_STATUS_NO_ERROR`

The control values were set successfully.

DM_STATUS_INVALID_ID

The specified openid is invalid.

DM_STATUS_SEND_QUEUE_OVERFLOW

There was not enough space on the path send queue for this latest message. Try again later after the device has had time to catch up, or specify a larger send queue size on open.

DM_STATUS_RECEIVE_QUEUE_OVERFLOW

There is not currently enough space on the receive queue to hold the reply to this message. Read some replies from the receive queue and then try to send again, or specify a larger receive queue size on open.

DM_STATUS_INVALID_PARAMETER

At least one of the parameters in the controls array was not recognized (the first such offending control will be marked with length -1, remaining controls will be skipped and the entire message will be ignored).

DM_STATUS_INVALID_VALUE

At least one of the parameters in the controls array has a value which is invalid. This may be because the parameter value is outside the legal range, or it may be that parameter value is inconsistent (the entire message will be ignored and the system will attempt to flag the first offending value by setting the length to -1).

RETURN EVENT

The event returned from processing a dmSendControls may be one of the following:

DM_CONTROLS_COMPLETE

The controls were processed without error.

DM_CONTROLS_ABORTED

The processing of the controls were aborted due to another asynchronous event, such as the dmEndTransfer function was requested.

DM_CONTROLS_FAILED

The processing of the controls failed because the values were not accepted at the time of processing.

Send Buffers

This function sends a message containing a list of buffers to a previously-opened digital media device. These buffers are enqueued on the send queue in sequence with any other messages to that device. All the buffers within a single message are considered to apply to the same point in time. For example, a single buffers message could contain image, audio, HANC and VANC buffers, each specified with its own buffer parameter in the buffers message.

```
DMstatus dmSendBuffers( DMopenid openid, DMpv* buffers );
```

`openid` refers to a previously-opened logical connection as returned from dmOpen, while buffers is a message containing a list of buffer parameters.

As each message is processed by the path, a reply message will be enqueued for return to the application. By examining that reply, the application may obtain the result of attempting to process the buffers.

A successful return value from the dmSendBuffers guarantees only that the requested buffers have been enqueued to the device. Any error return value indicates the buffers have not been enqueued and will thus have no effect.

The memory for the buffers is designated by the POINTER value, and is always owned by the application. However, after a buffer has been sent, it is on loan to the system and must not be touched by the application. After the buffer has been returned via ReceiveMessage, then the application is again free to delete and/or modify it.

When sending a buffer to be output, the application must set the buffer `length` to indicate the number of valid bytes in the buffer. In this case `maxLength` is ignored by the device (it doesn't matter how much larger the buffer may be, since the device won't read past the last valid byte).

When sending a buffer to be filled (on input) the application must set the buffer `maxLength` to indicate the maximum number of bytes which may be written by the device to the buffer. As the device processes the buffer, it will write no more than the `maxLength` bytes and then set the returned length to indicate the last byte written. The `maxLength` is returned without change. It is acceptable to send the same buffer multiple times.

Enqueueing entails a copy operation, so the application is free to delete/alter the message array as soon as the call returns. Any error return value indicates that the buffer has not been enqueued and will thus have no effect.

```
STATUS RETURN
```

This function returns one of the following:

DM_STATUS_NO_ERROR

The buffers message was enqueued successfully.

DM_STATUS_INVALID_ID

The specified `openid` is invalid.

DM_STATUS_SEND_QUEUE_OVERFLOW

There was not enough space on the path send queue for this latest message. Try again later after the device has had time to catch up, or specify a larger send queue size on open.

DM_STATUS_RECEIVE_QUEUE_OVERFLOW

There is not currently enough space on the receive queue to hold the reply to this message. Read some replies from the receive queue and then try to send again, or specify a larger receive queue size on open.

DM_STATUS_INVALID_PARAMETER

At least one of the parameters in the message was not recognized (the first such offending control will be marked with length -1, remaining controls will be skipped and the entire message will be ignored).

DM_STATUS_INVALID_VALUE

At least one of the parameters in the message has a value which is invalid. This may be because the parameter value is outside the legal range, or it may be that parameter value is inconsistent (the entire message will be ignored and the system will attempt to flag the first offending value by setting the length to -1).

RETURN EVENT

The event returned from processing a `dmSendControls` may be one of the following:

DM_BUFFERS_COMPLETE

The buffers were processed without error.

DM_BUFFERS_ABORTED

The processing of the buffers was aborted due to another asynchronous event, such as the `dmEndTransfer` function was requested.

`DM_BUFFERS_FAILED`

The processing of the buffers failed because the values were not accepted at the time of processing. This can occur both because parameters in the buffers message were invalid or due to the current control settings at the time of processing (because of previous `dmSendControls` messages), the processing of buffers would be invalid. Since preceding control messages may be incomplete, and each of the individual set or send controls may be valid, there still exists a point in time where the processing of buffers must be accomplished using those aggregate controls. If for some reason, the "combination of controls" is invalid, the processing is aborted and the event `DM_BUFFERS_FAILED` is returned.

Query Controls

To obtain the control values on a logical connection that are synchronous via an "in band" message, use the `dmQueryControls` operation:

```
DMstatus dmQueryControls( DMopenid openid, DMpv* controls );
```

`openid` is the 64-bit unique identifier returned by the `dmOpen` function. The controls parameter is a message containing various parameters as described in the preceding chapters.

The `dmQueryControls` sends a message containing a list of control parameters to a previously-opened digital media device. These controls are enqueued on the send queue in sequence with any other messages to that device. The control values returned are thus guaranteed to reflect any and all previously enqueued `dmSendControls` messages that have been processed.

This call returns as soon as the message has been enqueued to the device. It does not wait until the control value is available.

As each message is processed by the device, a reply message will be enqueued for return to the application. By examining that reply, the application may obtain the result of attempting to process the requested controls. Note that a device may take an arbitrarily long time to generate a reply (it may, for example, wait for several messages before replying to the first). If an application requires an immediate response, consider using the get controls operation instead.

Enqueueing entails a copy operation, so the application is free to delete/alter the message array as soon as the call returns. Any error return value indicates the control change has not been enqueued and will thus have no effect.

STATUS RETURN

This function returns one of the following:

DM_STATUS_NO_ERROR

The control values were set successfully.

DM_STATUS_INVALID_ID

The specified `openid` is invalid.

DM_STATUS_SEND_QUEUE_OVERFLOW

There was not enough space on the path send queue for this latest message. Try again later after the device has had time to catch up, or specify a larger send queue size on open.

DM_STATUS_RECEIVE_QUEUE_OVERFLOW

There is not currently enough space on the receive queue to hold the reply to this message. Read some replies from the receive queue and then try to send again, or specify a larger receive queue size on open.

DM_STATUS_INVALID_PARAMETER

At least one of the parameters in the controls array was not recognized (the first such offending control will be marked with length -1, remaining controls will be skipped and the entire message will be ignored).

RETURN EVENT

The event returned from processing a `dmQueryControls` may be one of the following:

DM_QUERY_CONTROLS_COMPLETE

The query controls were processed without error.

DM_QUERY_CONTROLS_ABORTED

The processing of the query controls were aborted due to another asynchronous event, such as the `dmEndTransfer` function was requested.

Get Wait Handle

When processing a number of digital media streams asynchronously, there exists a need for the application to know when processing is required on each individual stream. The `dmGetSendWaitHandle` and `dmGetReceiveWaitHandle` functions are provided to facilitate this processing:

```
DMstatus dmGetSendWaitHandle( DMopenid openid, DMwaitable* WaitHandle );
DMstatus dmGetReceiveWaitHandle( DMopenid openid, DMwaitable* WaitHandle );
```

The `openid` is a previously-opened digital media object as returned by a `dmOpen` call and the `WaitHandle` is the requested returned wait handle. This function returns an event handle on which an application may wait. On IRIX, UNIX and Linux, `DMwaitable` is a file descriptor for use in select(). On Windows, `DMwaitable` is a HANDLE which may be used in the win32 functions `WaitForSingleDevice` or `WaitForMultipleDevices`.

The send queue handle is signaled whenever the device dequeues a message and the message count drops below a preset level (set by the parameter `DM_OPEN_SEND_SIGNAL_COUNT` specified when the object was opened). Thus, if the send queue is full, an application may wait on this handle for notification that space is available for additional messages.

The receive queue handle is signaled whenever the device enqueues a reply message. Thus, if the receive queue is empty, the application may wait on this handle for notification that additional reply messages are ready.

The returned handles were created when the device was opened and are automatically destroyed when the path is closed.

`STATUS RETURN`

This function returns one of the following:

`DM_STATUS_NO_ERROR`

The wait handle was obtained successfully.

`DM_STATUS_INVALID_ID`

The specified open device handle is invalid.

Begin Transfer

`dmBeginTransfer` starts the actual transferring of buffers to the logical media connection:

```
DMstatus dmBeginTransfer (DMopen id openid);
```

The `openid` is a previously-opened digital media object as returned by an `dmOpen` call.

This function begins a continuous transfer on the specified Path or Xcode. It is not used on a logical connection to a Jack. This call advises the device to begin processing buffers and returning messages to the application. As stated earlier, sending a buffer to a device that has not yet begun transfers will cause the send queue to stall until the transfers have started. Typically applications will open a device, send several buffers and then call `dmBeginTransfer`. This call returns as soon as the device has begun processing transfers. It does not block until the first buffer has been processed. It is an error to call this function more than once without an intervening call to `dmEndTransfer`.

NOTES

The delay between a call to dmBeginTransfer and the transfer of the first buffer is implementation-dependent. To begin sending data at a particular time, an application should start the transfer early (enqueueing blank buffers) and use the UST/MSC mechanism to synchronize the start of real data.

STATUS RETURN

This function returns one of the following:

DM_STATUS_NO_ERROR

The device agreed to begin transfer on the path.

DM_STATUS_INVALID_ID

The specified open device id is invalid.

DM_STATUS_NO_OPERATION

The call had no effect (transfers have already been started).

XCode Work

For software-only transcoders opened with the DM_XCODE_MODE_INT32 open option set to DM_XCODE_MODE_SYNCHRONOUS, this function allows an application to control exactly when (and in which thread) the processing for that codec takes place.

```
DMstatus dmXcodeWork( DMopenid openid );
```

`openid` refers to a previously-opened digital media transcoder. This function performs one unit of processing for the specified codec. The processing is done in the thread of the calling process, and the call does not return until the processing is complete. For most codecs a "unit of work" is the processing of a single buffer from the source queue and the writing of a single resulting buffer on the destination queue.

Note: Note - the default behavior for all codecs is for processing to happen automatically as a side effect of enqueueing messages to the device. This function only applies to software codecs and only applies if they are opened with the DM_XCODE_MODE_SYNCHRONOUS open option.

STATUS RETURN

This function returns one of the following:

DM_STATUS_NO_ERROR

The software transcoder performed one unit of work successfully.

DM_STATUS_INVALID_ID

The specified `openid` is invalid.

DM_STATUS_NO_OPERATION

There was no work to be done.

RETURN EVENTS

There are no return events associated with this function.

Get Message Count

During the processing of messages it is sometimes necessary to inquire as to the "fullness" of the message queues. These functions provide that capability:

```
DMstatus dmGetSendMessageCount ( DMopenid openid, DMint32* messageCount);
DMstatus dmGetReceiveMessageCount ( DMopenid openid, DMint32* messageCount );
```

`openid` is a previously-opened digital media object returned by `dmOpen`. `MessageCount` is the resulting returned count.

These functions return a count of the number of messages in the send or receive queues of a device. The send queue contains messages queued by the application for processing by the device while the receive queue holds messages which have been processed and are waiting to be read by the application. A message is considered to reside in the send queue from the moment it is enqueued by the application until the moment the device begins processing it. A message resides in the receive queue from the moment the device enqueues it, until the moment the application dequeues the corresponding reply message (all messages in the receive queue are counted, regardless of whether or not they were successfully processed). The message counts are intended to aid load-balancing in sophisticated applications. They are not a reliable method for predicting `UST/MSC` pairs.

Some devices can begin processing one or more following messages before the first has been completed. Thus, the sum of the send and receive queue counts may be less than the difference between the number of messages which have enqueued and dequeued by the application. Note also that the time lag between a message being removed from the send queue, and the time at which it affects data passing though a physical jack, is implementation dependent. The message counts are not a reliable method for timing or synchronizing media streams.

`STATUS RETURN`

This function returns one of the following: `DM_STATUS_NO_ERROR`

The message count was obtained successfully.

`DM_STATUS_INVALID_ID`

The specified open device id is invalid.

Receive Messsage

In order for applications to obtain the results of previous digital media requests, the `dmReceiveMessage` function is used.

```
DMstatus dmReceiveMessage( DMopenid openid, DMint32* messageType, DMpv *reply );
```

`openid` is a previously-opened digital media object. `messageType` is an integer to be filled in by the device, indicating the type of message received. `reply` is a pointer to the head of the reply message.

This function reads the oldest message from the receive queue. The receive queue holds reply messages sent from a digital media device back to an application.

Messages on the receive queue may be the result of processing a message sent with `dmSendControls`, or they may result from processing a message sent with `dmSendBuffers`, or they may be generated spontaneously by the device to advise the application of some exceptional event.

Each message sent with an `dmSendBuffers` or `dmSendControls` generates a single reply message with `messageType` indicating whether or not the message was processed successfully and a pointer to a list of parameters holding the reply.

The contents of the reply array are guaranteed to remain valid until the next call to `dmReceiveMessage`. It is acceptable for an application to modify the reply and then send it to the same or to another device by calling `dmSendControls` or `dmSendBuffers`.

Note that, on some devices, triggering of the receive wait handle does not guarantee that a message is waiting on the receive queue. Thus applications must accept a status return of `DM_STATUS_RECEIVE_QUEUE_EMPTY` from an `dmReceiveMessage` function.

Get Returned Parameters

In returned messages, the application often wants to query specific parameters. The `dmPvFind` convenience function is provided for this use;

```
DMpv* dmPvFind( DMpv* msg, DMint64 param );
```

`Msg` is a message for which the parameter being searched is to be found. The `param` argument is the parameter that is being searched.

End Transfer

For an application to invoke an orderly shutdown of a digital media stream, the `dmEndTransfer` function should be issued.

`openid` is a previously-opened digital media object.

This function ends a continuous transfer on the specified path or transcoder. This call advises the device to stop processing buffers and aborts any remaining messages on its input queue. This is a blocking call. It does not return until transfers have stopped and any messages remaining on the device input queue have been aborted and flushed to the device output queue. Calling `dmEndTransfer` on a device which has not begun transfers is legal (it still causes the queue to be flushed). Any messages which are flushed will be marked to indicate they were aborted. Buffer messages are marked `DM_BUFFERS_ABORTED`, while controls messages are marked `DM_CONTROLS_ABORTED`.

`STATUS RETURN`

`DM_STATUS_NO_ERROR`

The device agreed to end transfer on the path.

`DM_STATUS_INVALID_ID`

The specified open device handle is invalid.

Close Processing

After an application is finished with a digital media connection, it should terminate that connection. The `dmClose` function is provided for that use. Note that an `dmClose` is implied if an application terminates (for any reason) before an `dmClose` function is called. A previously opened digital media object can be closed using:

```
DMstatus dmClose(DMopenid openid);
```

`openid` is the handle of the device to be closed. When a digital media object is closed, all messages in the message queues of the device are discarded. The device handle `openid` becomes invalid; any subsequent attempt to use it to refer to the closed object will result in an error.

dmClose returns DM_STATUS_INVALID_ID if openid is invalid. Otherwise it returns DM_STATUS_NO_ERROR after the device has been closed and associated resources have been freed.

The pipes opened as a side-effect of opening a transcoder are also closed as a side-effect of closing a transcoder. Pipes should not be closed explicitly.

Utility Functions

There are a number of other useful functions available in the DM API. They are described here.

Get Version

```
DMstatus dmGetVersion( DMint32 majorVersion, DMint32 minorVersion );
```

Use to obtain the version number for the Digital Media Library. The major version number is the first digit in the version. For example, the 1.0 release will have a major number of 1 and a minor number of 0. Changes in major numbers indicate a potential incompatibility, while changes in minor numbers indicate small backward-compatible enhancements. Within a particular major version, all the minor version numbers will start at 0 and increase monotonically. Note that this is the version number of the DM core library, the version numbers for device-dependent modules are available in the capabilities list for each physical device.

STATUS RETURN

This function returns one of the following:

DM_STATUS_NO_ERROR

The version numbers were obtained successfully.

DM_STATUS_INVALID_ARGUMENT

At least one of the pointers passed in is invalid.

Status Name

```
const char *dmStatusName( DMstatus status );
```

Intended mainly as an aid in debugging, this call converts the integer DM status value into a C string. The converted string is exactly the same as the status enum. value. For example, the value DM_STATUS_NO_ERROR, is converted to the string ''DM_STATUS_NO_ERROR''.

FUNCTION RETURN

This function returns a valid C string, or NULL if the status value is invalid.

Message Name

```
const char *dmMessageName( DMint32 messageType );
```

Intended mainly as an aid in debugging, this call converts the integer DM message type into a C string. The converted string is exactly the same as the message enum values. For example, the value DM_CONTROLS_FAILED, is converted to the string "DM_CONTROLS_FAILED".

FUNCTION RETURN

This function returns a valid C string, or NULL if the message value is invalid.

DMpv String Conversion Routines

```
DMstatus  dmPvValueToString(DMint64 objectId, DMpv* pv, char* buffer, DMint32* bufferSize);
DMstatus  dmPvParamToString(DMint64 objectId, DMpv* pv,char* buffer, DMint32* bufferSize);
DMstatus  dmPvToString(DMint64 objectId, DMpv* pv,  char* buffer, DMint32* bufferSize);
DMstatus  dmPvValueFromString(DMint64 objectId, const char* buffer,DMint32* bufferSize,
                              DMpv* pv, DMbyte* arrayData, DMint32 arraySize);
DMstatus  dmPvParamFromString(DMint64 objectId, const char* buffer, DMint32* size, DMpv* pv);
DMstatus  dmPvFromString(DMint64 objectId, const char* buffer, DMint32* bufferSize, DMpv* pv,
                         DMbyte* arrayData, DMint32 arraySize);
```

Parameter

objectID is the 64–bit ID number for the digital media library on which the parameter is interpreted.

pv is a pointer to the DMpv for use in the conversion.

buffer is a pointer to a buffer to hold the string.

bufferSize initially contains the size of the buffer (in bytes). Upon completion, this is overwritten with the actual number of bytes processed.

arrayData is a pointer to a buffer to hold any array data resulting from the conversion.

arraySize initially contains the size of the array buffer (in bytes).

Description

These routines convert between `DMpv` param/value pairs and strings. They are of benefit to applications writing lists of parameters to/from files, but are most commonly used as an aid to debugging.

These routines make use of the parameter capability data (see `dmPvGetCapabilities`) to generate and interpret human-readable ASCII strings.

`dmPvParamToString` converts pv->param into a string. The resulting value for *bufferSize* is the length of the string (excluding the terminating '\0').

`dmPvValueToString` converts pv->value into a string. The resulting value for *bufferSize* is the length of the string (excluding the terminating '\0').

`dmPvToString` converts the `DMpv` into a string. It writes the parameter name and value separated by '='. The resulting value for bufferSize is the length of the string (excluding the terminating '\0').

`dmPvParamFromString` interprets a string as a parameter name and writes the result in pv->param. It expects the string was created by `dmPvParamToString`.

`dmPvValueFromString` interprets a string as the value of a `DMpv` and writes the result in pv->value. It expects the string was created by `dmPvValueToString`. For scalar parameters, the result is returned in the value field of the `DMpv` structure and the array arguments are not used. For array parameters, additional space is required for the result. In this case, the contents of the array are returned inside the *arrayData* buffer and *arraySize* is set to indicate the number of bytes written.

`dmPvFromString` interprets a string as a DMpv. It expects the string was created by `dmPvToString`.

Note that the interpretation of a param/value pair depends on the parameter, its value, and the device on which it will be used. Thus, all these functions require both a param/value pair and a 64-bit device identifier. That identifier may be a static id (obtained from a call to `dmGetCapabilities`), it may be the open id of a jack, path

or transcoder (obtained from a call to `dmOpen`), or it may be the id of an open pipe (obtained by calling `dmXcodeGetOpenPipe`).

Status Return

These functions return one of the following status codes:

`DM_STATUS_NO_ERROR`

The conversion was performed successfully.

`DM_STATUS_INVALID_ID`

The specified id is invalid.

`DM_STATUS_INVALID_ARGUMENT`

The arguments could not be interpreted correctly. Perhaps the bufferSize or arraySize is too small to hold the result of the operation.

`DM_STATUS_INVALID_PARAMETER`

The parameter name is invalid. When converting to a string, the parameter name was not recognized on this device. When converting from a string, the string could not be interpreted as a valid parameter for this device.

`DM_STATUS_INVALID_VALUE`

The parameter value is invalid. When converting to a string, the parameter value was not recognized on this device. When converting from a string, the string could not be interpreted as a valid parameter value for this device.

Examples

This example prints the interpretation of a video timing parameter by a previously-opened video path. Note that the calls could fail if that path did not accept the particular timing value we have chosen here. Note also that, since the interpretation is coming from the device, this will work for device-specific parameters.

char buffer[200]; DMpv control;

control.param = DM_VIDEO_TIMING_INT32; control.value = DM_TIMING_1125_1920x1080_5994i;

```
dmPvParamToString(someOpenPath, &control;, buffer, sizeof(buffer));
printf("control.param is %s\n", buffer);
dmPvValueToString(someOpenPath, &control;, buffer, sizeof(buffer));
printf("control.value is %s\n", buffer);
dmPvToString(someOpenPath, &control;, buffer, sizeof(buffer));
printf("control is %s\n", buffer);
```

The output created by this example would be:

control.param is `DM_VIDEO_TIMING_INT32` control.value is `DM_TIMING_1125_1920x1080_5994i` control is `DM_VIDEO_TIMING_INT32 = DM_TIMING_1125_1920x1080_5994i`

Synchronization

This chapter describes DM support for synchronizing digital media streams. The described techniques are designed to enable accurate synchronization even when there are large (and possibly unpredictable) processing delays.

UST

To timestamp each media stream, some convenient representation for time is needed. In DM, time is represented by the value of the Unadjusted System Time (UST) counter. That counter starts 0 when the system is reset, and increases continuously (without any adjustment) while the system is running.

Each process and/or piece of hardware may have its own view of the system UST counter. That view is an approximation to the real system UST counter. The difference between any two views is bounded for any implementation.

Each UST timestamp is a signed 64-bit integer value with units of nanoseconds representing a recent view of the system UST counter. A current view of the system UST is obtained by using the `dmGetSystemUST` function call.

```
DMstatus dmGetSystemUST(DMint64 systemId, DMint64* ust);
```

Currently *systemId* must be `DM_SYSTEM_LOCALHOST`, otherwise the status `DM_STATUS_INVALID_ID` is returned. The resulting UST value is placed at the address UST. The status `DM_STATUS_INVALID_ARGUMENT` is returned if UST is invalid. The status `DM_STATUS_NO_ERROR` is returned on a successful execution.

Get System UST

```
DMstatus dmGetSystemUST( systemId,  );
```

Use to obtain the current UST (Universal System Time) on a particular system. At this time, the only legal system id is `DM_SYSTEM_LOCALHOST`.

`STATUS RETURN`

This function returns one of the following:

`DM_STATUS_NO_ERROR`

The system UST was obtained successfully.

`DM_STATUS_INVALID_ID`

The specified systemid is invalid.

`DM_STATUS_INVALID_ARGUMENT`

The UST was not returned successfully (perhaps an invalid pointer?).

UST/MSC/ASC Parameters

Basic support for synchronization requires that the application know exactly when video or audio buffers passed through a jack. In DM this is achieved with the UST/MSC buffer parameters:

`DM_AUDIO_UST_INT64, DM_VIDEO_UST_INT64`

The unadjusted system time (UST) is the timestamp for the most recently processed slot in the audio/video stream. For video devices, the UST time corresponds to the time at which the field/frame starts to pass through the jack. For audio devices, the UST time corresponds to the time at which the first sample in the buffer passed through the jack.

Typically, an application will pass `dmSendBuffers` a video message containing a `DM_IMAGE_BUFFER`, a `DM_VIDEO_MSC` and a `DM_VIDEO_UST` (and possibly an `DM_VIDEO_ASC` - see below), or an audio message containing a `DM_AUDIO_IMAGE_POINTER`, a `DM_AUDIO_UST`, and a `DM_AUDIO_MSC`. In some cases, a message can contain both audio and video parameters.

Each message is processed as a single unit, and a reply is returned to the application via `dmReceiveMessage`. That reply will contain the completed buffer and the UST/MSC(/ASC) corresponding to the time at which the data in the buffers passed in or out of the jack. Note that, due to hardware buffering on some cards, it is possible to receive a reply message before the data has finished flowing through an output jack.

`DM_AUDIO_MSC_INT64, DM_VIDEO_MSC_INT64`

The media stream count (MSC) is the most recently processed slot in the audio/video stream. This is snapped at the same instant as the UST time described above. Note that MSC increases by one for each potential slot in the media stream through the jack. For interlaced video timings, each slot contains one video field, for progressive timings, each slot contains one video frame. This means that when 2 fields are interlaced into one frame and sent as one buffer, then the MSC will increment by 2

(one for each field). Furthermore, the system guarantees that the least significant bit of the MSC will reflect the state of the Field Bit, being 0 for Field 1 and 1 for Field 2. For audio, each slot contains one audio frame.

`DM_AUDIO_ASC_INT64, DM_VIDEO_ASC_INT64`

The application stream count (ASC) is provided to aid the developer in predicting when the audio or video data will pass through an output jack. See the "UST/MSC for Output" section below for further information on the use of the ASC parameter.

UST/MSC Example

For example, here we send an audio buffer and video buffer to an I/O path and request both UST and MSC stamps:

```
DMpv message[7];
message[0].param = DM_IMAGE_BUFFER_POINTER;
message[0].value.pByte = someImageBuffer;
message[0].length = sizeof(someImageBuffer);
message[0].maxLength = sizeof(someImageBuffer);
message[1].param = DM_VIDEO_UST_INT64;
message[2].param = DM_VIDEO_MSC_INT64;
message[3].param = DM_AUDIO_BUFFER_POINTER;
message[3].value.pByte = someAudioBuffer;
message[3].length = sizeof(someAudioBuffer);
message[3].maxLength = sizeof(someAudioBuffer);
message[4].param = DM_AUDIO_UST_INT64;
message[5].param = DM_AUDIO_MSC_INT64;
message[6].param = DM_END;
dmSendBuffers( device, message);
```

After the device has processed the buffers, it will enqueue a reply message back to the application. That reply will be an exact copy of the message sent in, with the exception that the MSC and UST values will be filled in. (For input, the buffer parameter length will also be set to the number of bytes written into it). Note that a `dmSendBuffers` call can only have one `DM_IMAGE_BUFFER_POINTER.`

UST/MSC For Input

On input the application can detect if any data is missing by looking for breaks in the MSC sequence. This could happen if an application did not provide buffers fast enough to capture all of the signal which arrived at the jack. (An alternative to

looking at the MSC numbers, is to turn on the events `DM_AUDIO_SEQUENCE_LOST` or `DM_VIDEO_SEQUENCE_LOST`. Those will fire whenever the queue from application to device overflows.)

Given the UST/MSC stamps for two different buffers (UST1,MSC1) and (UST2,MSC2), the input sample rate in samples per nanosecond can be computed as:

$$sampleRate = \frac{(MSC_2 - MSC_1)}{UST_2 - UST_1}$$

Equation 10-1

One common technique for synchronizing different input streams is to start recording early, stop recording late, and then use the UST/MSC stamps in the recorded data to find exact points for trimming the input data.

An alternative way to start recording several streams simultaneously is to use predicate controls (see later).

UST/MSC For Output

On output, the actual output sample rate can be computed in exactly the same way as the input sample rate:

$$sampleRate\ \frac{(MSC_2 - MSC_1)}{(UST_2 - UST_1)}$$

Equation 10-2

Some applications must determine exactly when the next buffer sent to the device will actually go out the jack. Doing this requires two steps. First, the application must maintain its own field/frame count. This parameter is called the ASC. The ASC may start at any desired value and should increase by one for every audio frame or video field enqueued. (For convenience, the application may wish to associate the ASC with the buffer by embedding it in the same message. The parameters `DM_AUDIO_ASC_INT32` and `DM_VIDEO_ASC_INT32` are provided for this use.)

Now, assume the application knows the (UST,MSC,ASC) for two previously-output buffers, then the application can detect if there was any underflow by comparing the number of slots the application thought it had output, with the number of slots which the system actually output.

```
if (ASC_2 - ASC_1) == (MSC_2 - MSC_1) then all is well.
```

Assuming all is well, and that the application knows the current ASC, then the next data the application enqueues may be predicted to have a system sequence count of:

$$currentMSC / currentASC \rightarrow (MSC2 - ASC2)$$

Equation 10-3

and may be predicted to hit the output jack at time:

$$currentUST = UST2 \rightarrow \frac{(currentASC - ASC2)}{sampleRate}$$

Equation 10-4

Note that the application should periodically recompute the actual sample rate based on measured MSC/UST values. It is not sufficient to rely on a nominal sample rate since the actual rate may drift over time.

So, in summary: given the above mechanism, the application knows the UST/MSC pair for every processed buffer. Using the UST/MSC's for several processed buffers we can compute the frame rate. Given a UST/MSC pair in the past, a prediction of the current MSC, and the frame rate, the application can predict the UST at which the next buffer to be enqueued will hit the jack.

Predicate Controls

Predicate controls allow an application to insert conditional commands into the queue to the device. Using these we can pre-program actions, allowing the device to respond immediately, without needing to wait for a round-trip through the application.

Unlike the UST/MSC timestamps, predicate controls are not required to be supported on all audio/video devices. To see if they are supported on any particular device, look for the desired parameter in the list of supported parameters on each path (see dmGetCapabilities). The simplest predicate controls are:

`DM_WAIT_FOR_AUDIO_MSC_INT64`and

`DM_WAIT_FOR_VIDEO_MSC_INT64`

When the message containing this control reaches the head of the queue it causes the queue to stall until the specified MSC value has passed. Then that message, and subsequent messages, are processed as normal.

For example, here is code that uses WAIT_FOR_AUDIO_MSC to send a particular buffer out after a specified stream count:

```
DMpv message[3];
message[0].param = DM_WAIT_FOR_AUDIO_MSC_INT64;
message[0].value.int64 = someMSCInTheFuture;
message[1].param = DM_AUDIO_BUFFER_POINTER;
message[1].value.pByte = someBuffer;
message[1].value.length = sizeof(someBuffer);
message[2].param = DM_END;
dmSendBuffers( someOpenPath, message);
```

This places a message on the queue to the path and then immediately returns control to the application. As the device processes that message, it will pause until the specified media MSC value has passed before allowing the buffer to flow through the jack.

Using this technique an application can program several media streams to start in-sync by simply choosing some MSC count to start in the future.

Note: If both DM_IMAGE_DOMINANCE and DM_WAIT_FOR_VIDEO_MSC controls are set and do not correspond to the same starting output field order, the DM_WAIT_FOR_VIDEO_MSC_INT64 control will override DM_IMAGE_DOMINANCE_INT32 control settings.

Another set of synchronization predicate controls are:

DM_WAIT_FOR_AUDIO_UST_INT64 and DM_WAIT_FOR_VIDEO_UST_INT64

When the message containing this control reaches the head of the queue it causes the queue to stall until the specified UST value has passed. Then that message, and subsequent messages, are processed as normal. Note that the accuracy with which the system is able to implement the WAIT_FOR_UST command is device-dependent - see device-specific documentation for limitations. For example, here is code that uses WAIT_FOR_AUDIO_UST to send a particular buffer out after a specified time:

```
MLpv message[3];
message[0].param = ML_WAIT_FOR_AUDIO_UST_INT64;
message[0].value.int64 = someUSTtimeInTheFuture;
message[1].param = ML_AUDIO_BUFFER_POINTER;
message[1].value.pByte = someBuffer;
message[1].value.length = sizeof(someBuffer);
```

```
message[2].param = ML_END;
mlSendBuffers( someOpenPath, message);
```

This places a message on the queue to the path and then immediately returns control to the application. As the device processes that message, it will pause until the specified video UST time has passed before allowing the buffer to flow through the jack.

Using this technique an application can program several media streams to start in-sync by simply choosing some UST time in the future and program each to start at that time.

`DM_IF_VIDEO_UST_LT` or

`DM_IF_AUDIO_UST_LT`

When included in a message, this control will cause the following logical test: if the UST is less than the specified time, then the entire message is processed as normal. Otherwise, the entire message is simply skipped.

Regardless of the outcome, any following messages are processed as normal. Skipping over a message takes time, so there is a limit to how many messages a device can skip before the delay starts to become noticeable. All media devices will support skipping at least one message without noticeable delay.

Pixels in Memory

This appendix provides examples of the more common in-memory pixel formats, along with their corresponding dmSDK parameters.

Greyscale Examples

8–bit greyscale (1 byte per pixel)

```
byte 0
7      0
+------+
YYYYYYYY
```

Parameters:

- `DM_PACKING_8`
- `DM_COLORSPACE_CbYCr_*`
- `DM_SAMPLING_400`

Padded 12–bit greyscale (1 short per pixel)

```
short 0
15             0
+--------------+
ssssYYYYYYYYYYYY
```

Parameters:

- `DM_PACKING_S12in16R`
- `DM_COLORSPACE_CbYCr_*`
- `DM_SAMPLING_400`

RGB Examples

8–bit RGB (3 bytes per pixel)

```
byte 0    byte 1    byte 2
7      0  7      0  7      0
+------+  +------+  +------+
RRRRRRRR  GGGGGGGG  BBBBBBBB
```

Parameters:

- DM_PACKING_8
- DM_COLORSPACE_RGB_*
- DM_SAMPLING_444

8–bit BGR (3 bytes per pixel)

```
byte 0    byte 1    byte 2
7      0  7      0  7      0
+------+  +------+  +------+
BBBBBBBB  GGGGGGGG  RRRRRRRR
```

Parameters:

- DM_PACKING_8_R
- DM_COLORSPACE_RGB_*
- DM_SAMPLING_444

8–bit RGBA (4 bytes per pixel)

```
byte 0    byte 1    byte 2    byte 3
7      0  7      0  7      0  7      0
+------+  +------+  +------+  +------+
RRRRRRRR  GGGGGGGG  BBBBBBBB  AAAAAAAA
```

Parameters:

- DM_PACKING_8

- DM_COLORSPACE_RGB_*
- DM_SAMPLING_4444

8–bit ABGR (4 bytes per pixel)

```
byte 0    byte 1    byte 2    byte3
7      0  7      0  7      0  7      0
+------+  +------+  +------+  +------+
AAAAAAAA  BBBBBBBB  GGGGGGGG  RRRRRRRR
```

Parameters:

- DM_PACKING_8_R
- DM_COLORSPACE_RGB_*
- DM_SAMPLING_444

10–bit RGB (one 32–bit integer per pixel)

```
31            int              0
+-------------------------------+
RRRRRRRRRRGGGGGGGGGGBBBBBBBBBB00
```

Parameters:

- DM_PACKING_10_10_10_2
- DM_COLORSPACE_RGB_*
- DM_SAMPLING_444

10–bit RGBA (one 32–bit integer per pixel)

```
31            int              0
+-------------------------------+
RRRRRRRRRRGGGGGGGGGGBBBBBBBBBBAA
```

Parameters:

- DM_PACKING_10_10_10_2

- DM_COLORSPACE_RGB_*
- DM_SAMPLING_4444

12–bit RGBA (6 bytes per pixel)

```
byte 0    byte 1    byte 2    byte 3    byte 4    byte 5
7      0  7      0  7      0  7      0  7      0  7      0
+------+  +------+  +------+  +------+  +------+  +------+
RRRRRRRR  RRRRGGGG  GGGGGGGG  BBBBBBBB  BBBBAAAA  AAAAAAAA
```

Parameters:

- DM_PACKING_S12
- DM_COLORSPACE_RGB_*
- DM_SAMPLING_4444

Padded 12–bit RGB (three 16–bit shorts per pixel)

```
short 0           short 1           short 2
15             0  15             0  15             0
+--------------+  +--------------+  +--------------+
ssssRRRRRRRRRRR  ssssGGGGGGGGGGG  ssssBBBBBBBBBBB
```

Parameters:

- DM_PACKING_S12in16R
- DM_COLORSPACE_RGB_*
- DM_SAMPLING_444

Padded 12–bit RGBA (four 16–bit shorts per pixel)

```
short 0           short 1           short 2           short 3
15             0  15             0  15             0  15             0
+--------------+  +--------------+  +--------------+  +--------------+
ssssRRRRRRRRRRR  ssssGGGGGGGGGGG  ssssBBBBBBBBBBB  ssssAAAAAAAAAAA
```

Parameters:

- DM_PACKING_S12in16R
- DM_COLORSPACE_RGB_*
- DM_SAMPLING_4444

CbYCr Examples

8–bit CbYCr (3 bytes per pixel)

```
byte 0    byte 1    byte 2
7      0  7      0  7      0
+------+  +------+  +------+
bbbbbbbb  YYYYYYYY  rrrrrrrr
```

Parameters:

- DM_PACKING_8
- DM_COLORSPACE_CbYCr_*
- DM_SAMPLING_444

8–bit CbYCrA (4 bytes per pixel)

```
byte 0    byte 1    byte 2    byte 3
7      0  7      0  7      0  7      0
+------+  +------+  +------+  +------+
bbbbbbbb  YYYYYYYY  rrrrrrrr  AAAAAAAA
```

Parameters:

- DM_PACKING_8
- DM_COLORSPACE_CbYCr_*
- DM_SAMPLING_4444

10–bit CbYCr (one 32–bit integer per pixel)

```
31            int              0
+------------------------------+
bbbbbbbbbbYYYYYYYYYYrrrrrrrrrr00
```

Parameters:

- DM_PACKING_10_10_10_2
- DM_COLORSPACE_CbYCr_*
- DM_SAMPLING_444

10–bit CbYCrA (one 32–bit integer per pixel)

```
31             int             0
+------------------------------+
bbbbbbbbbbYYYYYYYYYYrrrrrrrrrrAA
```

Parameters:

- DM_PACKING_10_10_10_2
- DM_COLORSPACE_CbYCr_*
- DM_SAMPLING_4444

Padded 12–bit CbYCrA (four 16–bit shorts per pixel)

```
short 0           short 1           short 2           short 3
15             0  15             0  15             0  15             0
+--------------+  +--------------+  +--------------+  +--------------+
ssssbbbbbbbbbbbb  ssssYYYYYYYYYYYY  ssssrrrrrrrrrrrr  ssssAAAAAAAAAAAA
```

Parameters:

- DM_PACKING_S12in16R
- DM_COLORSPACE_CbYCr_*
- DM_SAMPLING_4444

422*x* CbYCr Examples

10–bit 422 CbYCr (5 bytes per 2 pixels)

```
      byte 0    byte 1    byte 2    byte 3    byte 4
      7      0  7      0  7      0  7      0  7      0
      +------+  +------+  +------+  +------+  +------+
      bbbbbbbb  bbYYYYYY  YYYYrrrr  rrrrrrYY  YYYYYYYY
```

pixel 1

```
      ++++++++  ++++++++  ++++++++  ++++++
```

pixel 2

```
      ++++++++  ++        ++++      ++++++++  ++++++++
```

Parameters:

- `DM_PACKING_10`
- `DM_COLORSPACE_CbYCr_*`
- `DM_SAMPLING_422`

10–bit 422 CbYCr (5 bytes per 2 pixels)

```
      byte 0    byte 1    byte 2    byte 3    byte 4
      7      0  7      0  7      0  7      0  7      0
      +------+  +------+  +------+  +------+  +------+
      bbbbbbbb  bbYYYYYY  YYYYrrrr  rrrrrrYY  YYYYYYYY
```

pixel 1

```
      ++++++++  ++++++++  ++++++++  ++++++
```

pixel 2

```
      ++++++++  ++        ++++      ++++++++  ++++++++
```

Parameters:

- `DM_PACKING_10_R`

- DM_COLORSPACE_CbYCr_*
- DM_SAMPLING_422

Padded 12–bit 422 CbYCr (four 16–bit shorts per 2 pixels)

```
short 0           short 1           short 2           short 3
15             0  15             0  15             0  15             0
+--------------+  +--------------+  +--------------+  +--------------+
ssssbbbbbbbbbbbb  ssssYYYYYYYYYYYY  ssssrrrrrrrrrrr  ssssYYYYYYYYYYYY
```

pixel 1

```
++++++++++++++++  ++++++++++++++++  ++++++++++++++++
```

pixel 2

```
++++++++++++++++                    ++++++++++++++++  ++++++++++++++++
```

Parameters:

- DM_PACKING_S12in16R
- DM_COLORSPACE_CbYCr_*
- DM_SAMPLING_422

10–bit 4224 CbYCrA (two 32–bit integers per 2 pixels)

```
31          int 0              0  31          int 1              0
+------------------------------+  +------------------------------+
bbbbbbbbbbYYYYYYYYYYAAAAAAAAAA00  rrrrrrrrrrYYYYYYYYYYAAAAAAAAAA00
```

pixel 1

```
++++++++++++++++++++++++++++++    ++++++++++
```

pixel 2

```
++++++++++                        ++++++++++++++++++++++++++++++
```

Parameters:

- DM_PACKING_10_10_10_2
- DM_COLORSPACE_CbYCr_*
- DM_SAMPLING_4224

Common Video Standards

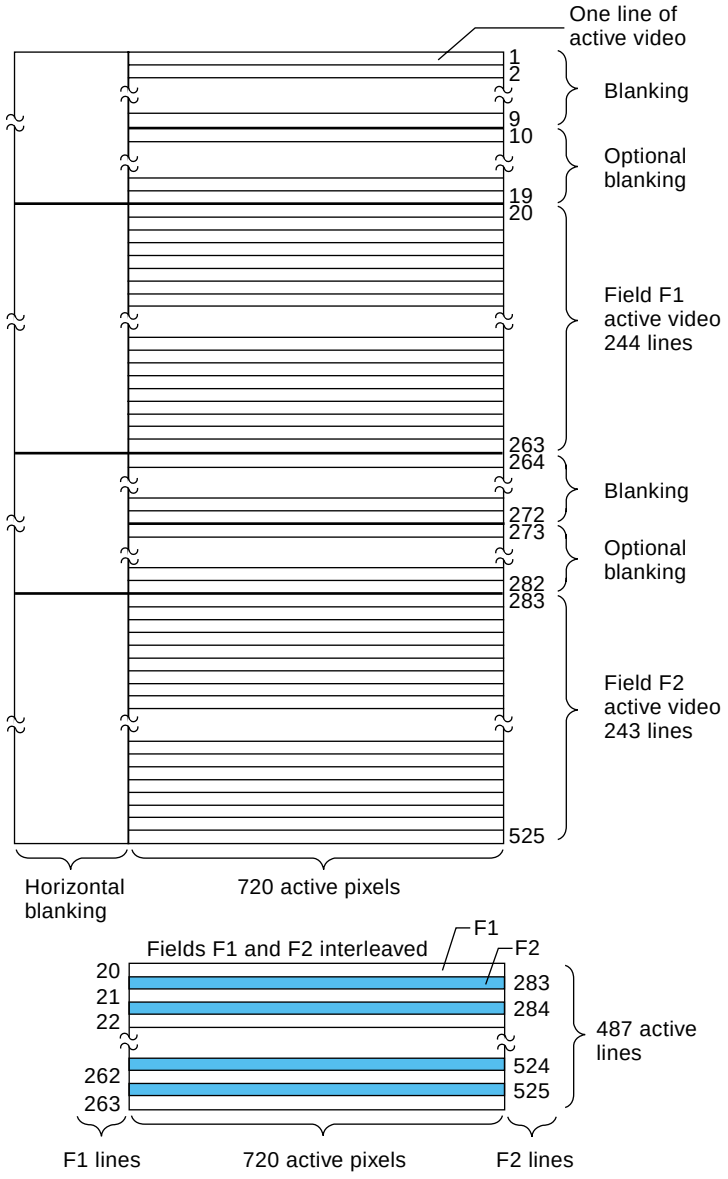

Figure B-1 525/60 Timing (NTSC)

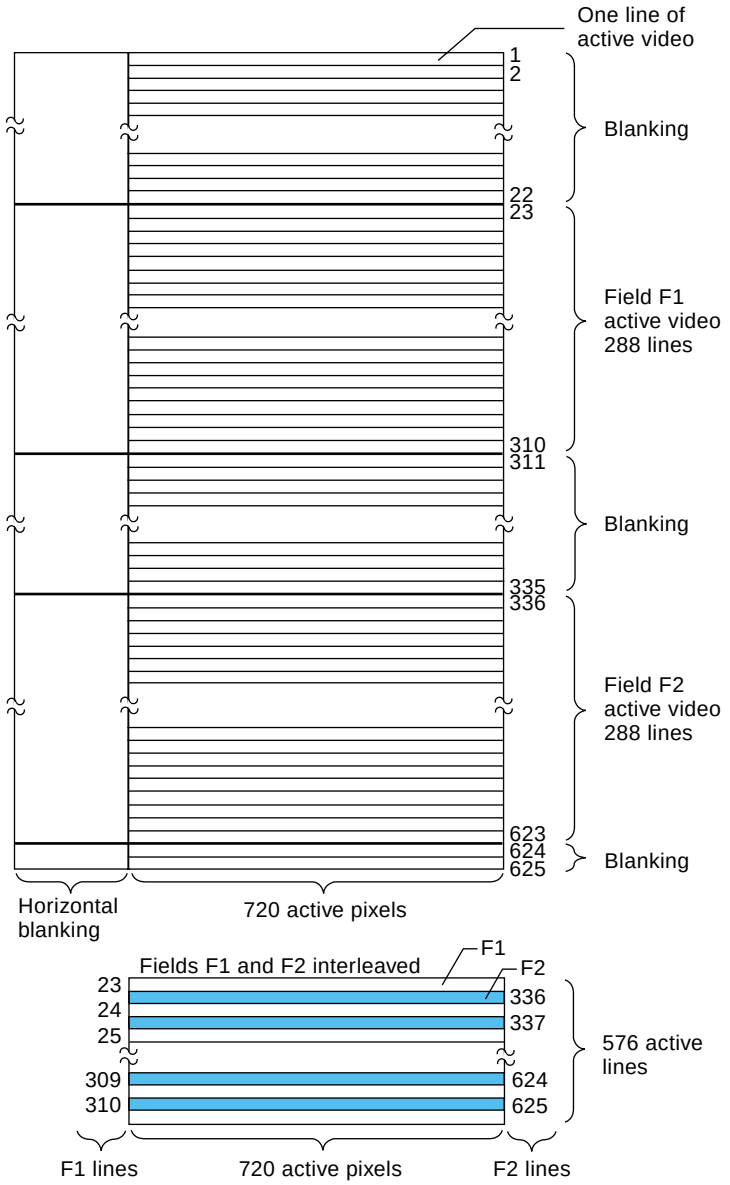

Figure B-2 625/50 Timing (PAL)

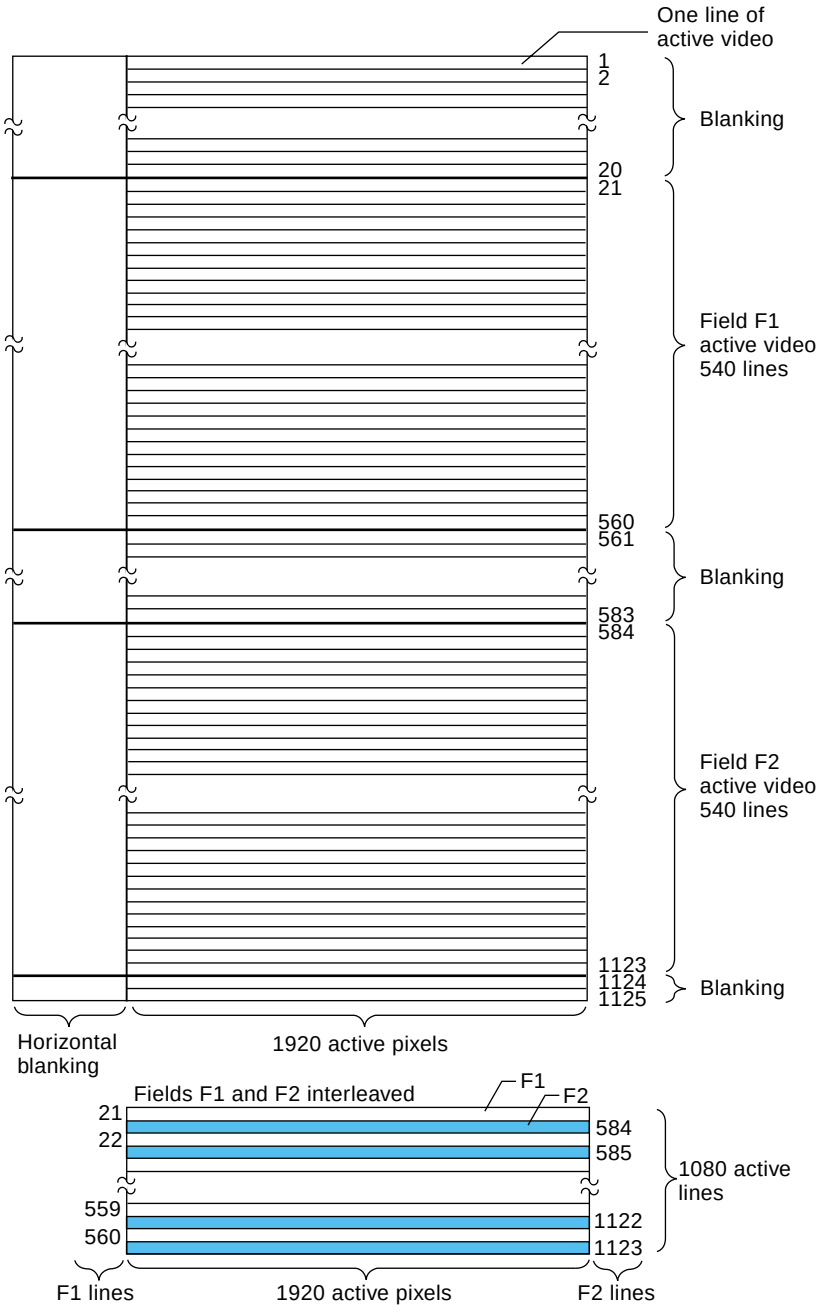

Figure B-3 1080i Timing (High Definition)

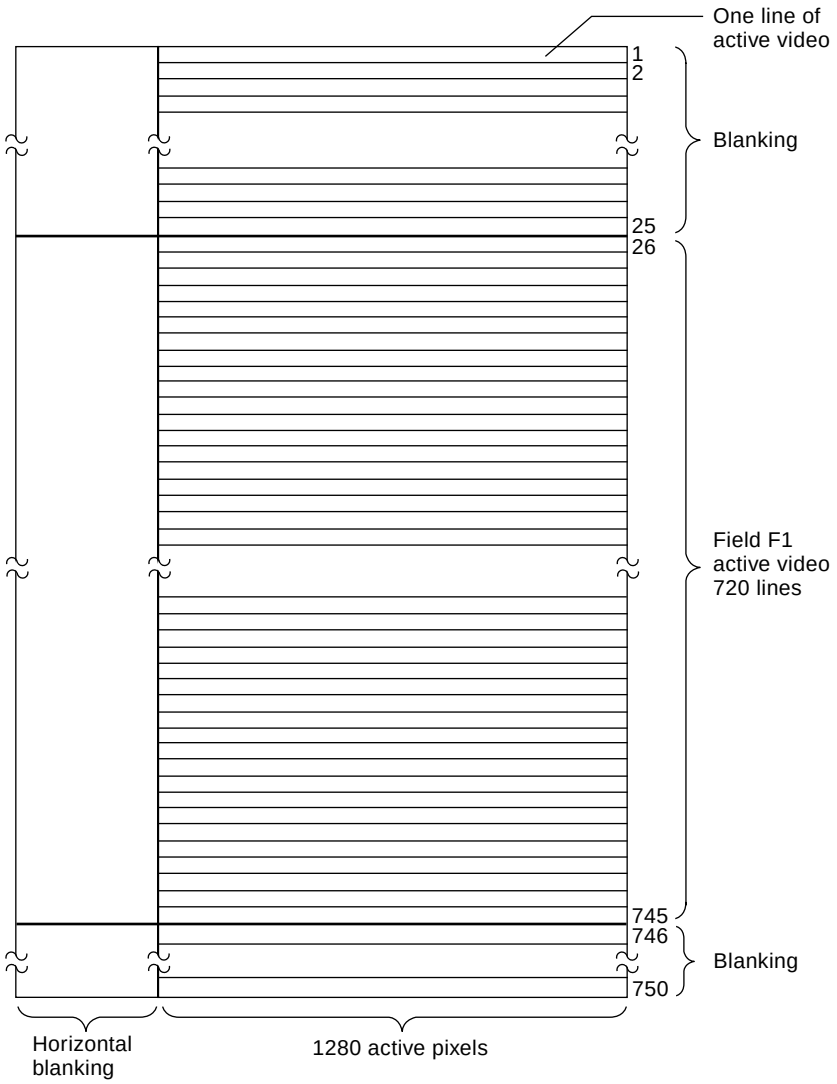

Figure B-4 720p Timing (High Definition)

Index

D

E

F

G

I

T

U

V