OpenML® Media Library Software Development Kit Beginner's Guide

007–4376–001

Record of Revision

Version	Description
001	June 2004 Original publication. Supports the 1.0 release of the OpenML Media Library Software Development Kit (*ML*) and ML 1.1.1 for IRIX 6.5

Contents

About This Guide

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

Related Publications

For more information about ML, see *OpenML Media Library Software Development Kit Programmer's Guide* and the `mlquery`(1ml) man page.

Obtaining Publications

You can obtain SGI documentation as follows:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, enter `infosearch` at a command line or select **Help > InfoSearch** from the Toolchest.
- On IRIX systems, you can view release notes by entering either `grelnotes` or `relnotes` at a command line.
- On Linux systems, you can view release notes on your system by accessing the `README.txt` file for the product. This is usually located in the `/usr/share/doc/`*productname* directory, although file locations may vary.
- You can view man pages by typing `man` *title* at a command line.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`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.

Reader Comments

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

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

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

  http://docs.sgi.com

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

- Send mail to the following address:

  Technical Publications
  SGI
  1500 Crittenden Lane, M/S 535
  Mountain View, California 94043–1351

SGI values your comments and will respond to them promptly.

Introduction

This guide is a quick introduction to the OpenML Media Library Software Development Kit (*ML*). If you are new to ML, you should read this guide and browse the online example programs.

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

ML Terminology

The following terms are used throughout this document, and some are used in the ML code:

Term	Definition
graphics / video	In ML, these terms 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	The hierarchy of all ML devices in the system, containing information about each ML 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 `ML_SYSTEM_LOCALHOST`. Each system contains one or more physical or logical devices.
physical device	A device that corresponds to device-dependent modules in ML. 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.
logical device	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 — it is possible for a single ML 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.
pipe	The connections from memory to the transcoder, and from the transcoder to memory.
transcoder	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. Example transcoders are DV compression and JPEG decompression.
UST	Unadjusted system time. UST is a special system clock that 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 that have passed though a jack. This measure is useful to synchronize media streams.

For More Information

For an in-depth treatment of ML, consult the *OpenML Media Library Software Development Kit Programmer's Guide* as you experiment with your own programs.

Getting Started with ML

The first thing you should do is examine your system with the `mlquery`(1ml) tool. This tool prints a list of all supported ML devices on the system.

Following is an example `mlquery` on the system `linux1`:

```
% mlquery
                SYSTEM: linux1
            active UST: (default software UST source)
               DEVICES:
                        Software DV_MMX Codec [0]
                        OSS audio device [0]
```

This output indicates that there are two installed devices:

- A software DV transcoder
- An audio I/O device (which in this case is built using the Linux OSS driver)

Other options to `mlquery` allow you to gather more information about the installed devices. See the `mlquery`(1ml) 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. The steps are as follows:

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

Step 1: Include the `ml.h` and `mlu.h` Files

To begin, you will need the following files:

File	Description
`ml.h`	Provides the core ML library functionality
`mlu.h`	Provides simple utility functions built on the core library

You may choose to use only the core library or you may find it convenient to use the simpler utility functions.

Include the files as follows:

```
#include <ML/ml.h>
#include <ML/mlu.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 ML capability tree, which contains information on every ML 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, you can use some of the utility functions to find a suitable output path:

```
MLint64 devId=0;
MLint64 jackId=0;
MLint64 pathId=0;

mluFindDeviceByName( ML_SYSTEM_LOCALHOST, argv[1], &devId );
mluFindFirstOutputJack( devId, &jackId );
mluFindPathToJack( jackId, &pathid, memoryAlignment );
```

Step 3: Open the Device Output Path

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

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

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

Note: Sometimes an `mlOpen` 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

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

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

In ML, applications communicate with devices using messages. These messages are known as `MLpv` messages, because they consist of a list of `param/value` pairs. An `MLpv` ends with an `ML_END` to indicate completion.

For example:

```
mlpv controls[5];
MLreal64 gain = -12; /* decibels */

controls[0].param = ML_AUDIO_FORMAT_INT32;
controls[0].value.int32 = ML_AUDIO_FORMAT_S16;
controls[1].param = ML_AUDIO_CHANNELS_INT32;
controls[1].value.int32 = 1;
controls[2].param = ML_AUDIO_GAINS_REAL64_ARRAY;
controls[2].value.pReal64 = &gain
controls[2].length = 1;
controls[3].param = ML_AUDIO_SAMPLE_RATE_REAL64;
controls[3].value.real64 = 44100.0;
controls[4].param = ML_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 you have constructed the `MLpv` controls message, you must set the controls on the open audio path as follows:

```
mlSetControls(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: 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, do the following:

1. Construct an `MLpv` message that describes the buffer. That message must include both a pointer to the buffer and the length of the buffer (in bytes):

   ```
   MLpv msg[2];
    msg[0].param = ML_AUDIO_BUFFER_POINTER;
    msg[0].value.pByte = ourAudioBuffer;
    msg[0].length = sizeof(ourAudioBuffer);
    msg[1].param = ML_END;
   ```

2. Send the buffers message to the opened path:

   ```
   mlSendBuffers(openPath, msg);
   ```

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

Note: Unlike the `mlSetControls` call, the `mlSendBuffers` 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 `mlBeginTransfer` call as follows:

```
mlBeginTransfer(openPath);
```

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

```
sleep(5)
```

Using `sleep` is simple, but the example in Chapter 4, "Realistic Audio Output Program" shows a better approach. See "Step 6: Begin the Transfer" on page 13.

Step 8: Receive the Reply Message

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

```
MLint32 messageType;
MLpv* message;

mlReceiveMessage(openPath, &messageType, &Message );

if( messageType == ML_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:

```
mlClose(openPath);
```

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

Realistic Audio Output Program

The procedure in Chapter 3, "Simple Audio Output Program" on page 5 was for a single audio buffer. In the example in this chapter, you will process millions of audio samples, using the following procedure:

Step 1: Include the `ml.h` and `mlu.h` Files

See "Step 1: Include the `ml.h` and `mlu.h` Files" on page 5.

Step 2: Locate a Device

See "Step 2: Locate a Device" on page 6.

Step 3: Open the Device Output Path

Open the device output path just as in the previous example in "Step 3: Open the Device Output Path" on page 6:

```
mlOpen( 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 4: 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 must allocate space for a small number of buffers and reuse each buffer many times to complete the whole transfer.

Assume that memory has been allocated for 12 audio buffers and that those buffers have been filled with the first few seconds of audio data to be output.

Step 5: Send Buffers to the Open Path

Send each of the 12 buffers to the open path. Here the queue of messages between application and device becomes more interesting. The following code segment enqueues all the audio buffers to the device:

```
int i;
for ( i=0, i < 12; ++i )
  {
    MLpv msg[3];
    msg[0].param = ML_AUDIO_BUFFER_POINTER;
    msg[0].value.pByte = (MLbyte*)buffers[i];
    msg[0].length = bufferSize;
    msg[1].param = ML_AUDIO_UST_INT64;
    msg[1].param = ML_END;
    mlSendBuffers( openPath, msg );
}
```

Notice that each audio buffer is sent in its own message. This is because each message is processed atomically, and therefore 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 6: Begin the Transfer

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:

```
mlBeginTransfer(openPath);
```

At this point, you could tell the program to sleep while the device processes the buffers, as was done in Chapter 3, "Simple Audio Output Program" on page 5. 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 ML terminology, that file descriptor is called a *wait handle* on the receive queue:

```
MLwaitable pathWaitHandle;
 mlGetReceiveWaitHandle(openPath, &pathWaitHandle);
```

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

On IRIX or Linux:

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

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

On Windows:

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

Step 7: Receive Replies from the Device

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

```
MLint32 messageType;
 MLpv* replyMessage;

 mlReceiveMessage(openPath, &messageType, &replyMessage );

 if( messageType == ML_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:

```
MLbyte* audioBuffer = replyMessage[0].value.pByte;
 MLint64 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 8: Refill the Buffer for Further Processing

You can refill the buffer with more audio data and send it back to the device to be processed again with the following:

```
mlSendBuffers(openPath, replyMessage);
```

In this case, you are making a small optimization. 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 9: End the Transfer

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

```
mlEndTransfer(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 `mlEndTransfer` 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, you can send more buffers to the path (see "Step 5: Send Buffers to the Open Path" on page 12).

Step 10: Close the Path

Use the following to close the path:

```
mlClose(openPath);
```

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

Audio/Video Jacks

ML 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
- Pipes to/from transcoders

All share common control, buffer, and queueing mechanisms. This chapter describes these mechanisms in the context of operating on a jack and its associated path. The *OpenML Media Library Software Development Kit Programmer's Guide* discusses the application of these mechanisms to transcoders and pipes.

Open a Jack

To open a connection to a jack, call `mlOpen`:

```
MLstatus mlOpen(const MLint64 objectId, MLpv* options, MLopenid* 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 because an application cannot send data to a jack (it must be sent via a path). Therefore, `mlSendControls` and `mlSendBuffers` are not supported on a jack; you must use `mlSetControls` 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 synchronization lost or gained.

Construct a Message

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

```
MLpv message[3];
message[0].param = ML_VIDEO_FLICKER_FILTER_INT32;
message[0].value.int32 = 1;
message[1].param = ML_VIDEO_NOTCH_FILTER_INT32;
message[1].value.int32 = 1;
message[2].param = ML_END
```

Set Jack Controls

Jack controls deal with external conditions and not processing associated with data transfers. Therefore, applications use `mlSetControls` or `mlGetControls` calls to manipulate these controls. Following is an example of how you can obtain the external synchronization signal (genlock) vertical and horizontal phase immediately:

```
MLpv message[3];
message[0].param = ML_VIDEO_H_PHASE_INT32;
message[1].param = ML_VIDEO_V_PHASE_INT32;
message[2].param = ML_END;
if( mlGetControls( aJackConnection, message))   handleError();
 else
   printf("Horizontal offset is %d, Vertical offset is %d\n",
      message[0].value.int32, message[1].value.int32);
```

`mlSetControls` and `mlGetControls` are blocking calls. If the call succeeds, the message has been successfully processed.

Note: Not all controls may be set via `mlSetControls`. The access privilege in the `param` capabilities can be used to verify when and how controls can be modified.

Close a Jack

When an application has finished using a jack, it should close it with `mlClose`:

```
MLstatus mlClose(MLopenid openId);
```

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

Index