Core Media IO Device Abstraction Layer Example

Overview

What is the DAL?

The Core Media IO Device Abstraction Layer (DAL) is analogous to Core Audio's Hardware Abstraction Layer (HAL). Just as the HAL deals with audio streams from audio hardware, the DAL handles video (and muxed) streams from video devices.

Developers who are used to the APIs provided in `<CoreAudio/AudioHardware.h>` will be immediately familiar those detailed in `<CoreMediaIO/CMIOHardware.h>`. Just as Core Audio's HAL made use of a System, Device, Stream, and Control abstraction to represent audio capabilities, Core Media IO's DAL does the same to represent video capabilities.

HAL vs. DAL Architecture

Apart from the types of devices they deal with, the largest architectural difference between the HAL and the DAL is the manner in which data are delivered to interested clients. When doing input, the HAL provides audio samples to clients via an `IOProc` and expects the client to be done with the data by the time the `IOProc` has completed. In contrast, the DAL provides it clients with a queue into which it deposits data, and the clients can pull data from the queue at whatever pace they like. Moreover, once they have pulled data from the queue they might hold onto to it for an arbitrary length of time.

Reusing the Core Audio Utility Classes

Since there is a large overlap between HAL PlugIn and DAL PlugIn developers, the DAL example tries to use many of the items introduced in the Core Audio Utility Classes.

Note: The Core Audio Utility Classes must first be downloaded from developer.apple.com. The Sample.xcodeproj is configured assuming they were placed in the '/Library/Developer/', but can be placed wherever desired (with the Sample.xcodeproj reconfigured accordingly).

- **Core Audio Public Utilities**
  The DAL uses the Core Audio Public Utilities when possible rather than duplicating that functionality. Examples of this would be `CACFString.h`, `CACFNumber.h`, and many others.

- **Extend Core Audio Public Utilities Wrapper Pattern**
  Just as the Core Audio Public Utilities provides convenient C++ wrappers for many OS constructs that are germane to audio, the DAL example provides C++ wrappers for OS constructs useful in the video domain.

Not An API Discussion

This document is not geared towards the DAL API. See `<CoreMediaIO/CMIOHardware.h>` for the API reference.

DAL Example Parts

The DAL and DAL PlugIns

The DAL "proper" implements the <CoreMediaIO/CMIOHardware.h> API and exists in the CoreMediaIO.framework. It hosts a CFPlugIn mechanism for loading plugIns from /Library/CoreMediaIO/Plug-Ins/DAL/. This example provides a sample implementation composed of the following items:

Sample.plugin

This implements the CFPlugIn portion of the driver. This would be the only item needed if the device can be accessed in user space and it is not necessary to shared it across multiple processes simultaneously.

The Sample.plugin has an implementation that allows devices to be shared across multiple processes. It accomplishes this by communicating with the device via an intermediary, the SampleAssistant.

SampleAssistant

It is desirable to allow many applications access to a device at the same time, but sometimes that is precluded by lower level issues. For example, the FireWire and USB families only allow a single process access to a device at a time. Similarly, the IOVideo family can only deliver its frames to a single process. To get around these shortfalls, the SampleAssistant acts as the intermediary to the device and its interested clients. It gains the exclusive access to the device, and then vends frames to the clients, using Mach's copy-on-write facility to minimize the number of buffer copies.

This sample implementation communicates with a KEXT for demonstration purposes, but it is not necessary in all cases. For example, there would be no need to need to use a KEXT for FireWire or USB devices.

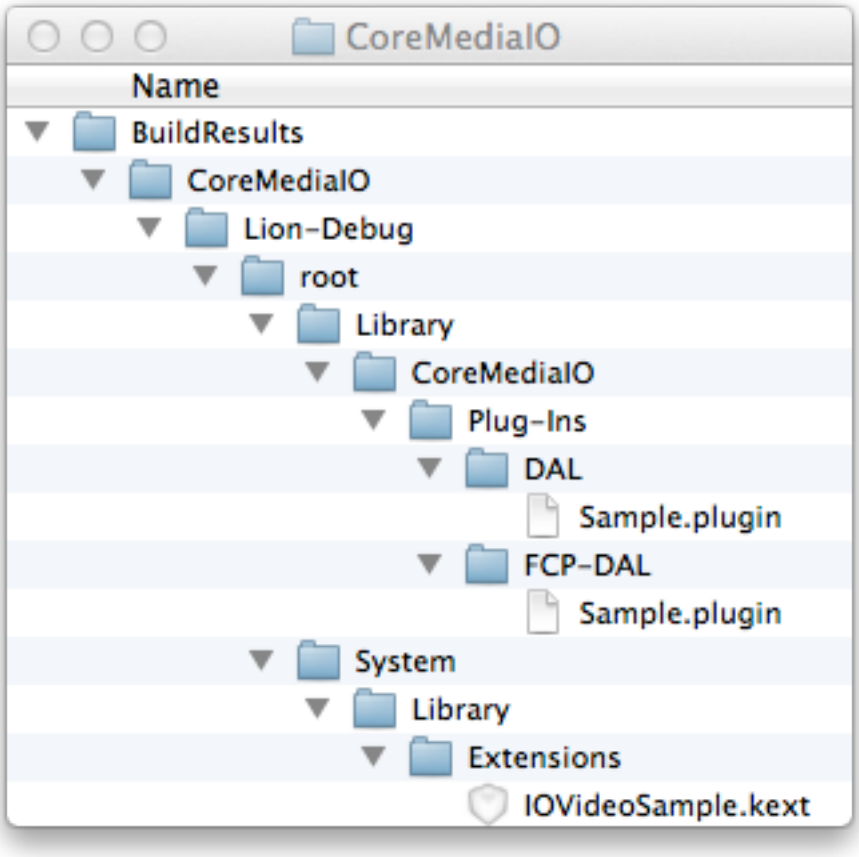
Sample KEXT

Some devices (such as PCI cards) require KEXTs. The Sample KEXT uses a software-only device to demonstrate the usage of the <IOKit/video/IOVideoFamily.h>

Xcode Projects

Structure

The Xcode projects are designed to use `{Path-To-CoreMediaIO-Folder}/BuildResults` as the location for their builds. A root will be built into the BuildResults directory that is suitable for ditto-ing into the System or using as the basis for an Installer. If manually ditto-ing into the System, it will be necessary to explicitly set the permissions. A build of the Lion-Debug configuration would result in the following:



Sample.xcodeproj

This builds the `Sample.plugin`, `IOVideoSample.kext`, and the `SampleAssistant`. The `SampleAssistant` gets embedded in the plugIn's Resources folder (`/Sample.plugin/Contents/Resources/SampleAssistant`).

DAL Example Tidbits

Explicit Namespaces

The DAL example and its associated plugIns make heavy use of nested C++ namespaces. The namespaces prevent collisions from occurring in the global namespace and help define the logical hierarchy. Every file is prefixed by its namespace, allowing its position in the hierarchy to be immediately understood. For example, the file `CMIO_DP_Sample_Device.cpp` is nested 3 layers deep:

NAMESPACE	MEANING	PURPOSE
CMIO	CoreMediaIO	All encompassing namespace
DP	DAL PlugIn	DAL PlugIn related
Sample	Sample	Sample related

File Name Conventions

Throughout the DAL and the DAL PlugIns (and to a lesser extent some of the other CoreMediaIO items), one can determine certain attributes of a file and its contents by whether or not it makes use of an "_" in the file name.

If a file name has "_" in it, you can be assured it is C++, throws exceptions, and each "_" delimits a namespace.

If a file lacks underscores, it is C based and returns errors instead of throwing exceptions.

How Do I Use C++ with Cocoa?

Preferably, just have all the source files be .mm instead of .m.

If for some reason .m file must be present, tell Xcode to compile the file as Objective C++ (instead of plain-old Objective C) through the inspector.

Debug Messages

`CMIODebugMacros.h` is a CoreMediaIO variant of `CADebugMacros.h` (the deltas are discussed in the file). Through the use of the standard defaults mechanism, one can enable/disable debugging on the fly:

```
// CMIODebug()
//   Reports true if 'defaults write com.apple.cmio Debug true' has been set via Terminal in
//   the current user account prior to running the application.
//   This MUST be true for any module specific debugging to be logged.
Boolean CMIODebug(void);

// CMIOModuleDebugLevel()
//   Reports true if 'defaults write com.apple.cmio {moduleKey} {levelValue}' has been set via
//   terminal prior to running the application and true == CMIODebug() and
//   debugLevel <= levelValue.
//   There are two special values that can be used for {levelValue} :
//        -1 == always report true
//        0 == never report true,
Boolean CMIOModuleDebugLevel(CFStringRef moduleKey, CFIndex debugLevel);
```

```
// CMIOModuleDebug()
//   A convenience version of CMIOModuleDebugLevel() where 1 == debugLevel
#define CMIOModuleDebug(moduleKey)  CMIOModuleDebugLevel(moduleKey, 1)
```

For the `Sample.plugin`, its module key is `CMIO_DAL_PlugIn_Sample.Debug` (defined in the target's customized build settings). So the value of that key will determine whether the various `DebugMessage()` modules are printed are not.[1]

PlugIn / Assistant Messaging

MIG .defs File

The majority of the communication that takes place between the plugIn and the Assistant is in the form of Mach messages. The Mach Interface Generator (MIG) creates the stubs to handle the messaging based on a .defs file (in this case, `CMIODPASample.defs`). The syntax for a .defs is somewhat esoteric, so please refer directly to the "Mach 3 Server Writer's Guide" for information.

State Change Messages

The Assistant will message the plugIn when the number of devices changes, or a device's properties or controls change. The plugIn will subsequently query the Assistant (via the MIG mechanism above) to update its state information accordingly.

Frame Arrived Message

When the Assistant has a frame ready, it will invoke `mach_msg()` to message the plugIn. The message uses Mach's copy-on-write mechanism to pass the frame to the plugIn, so if the frame is used and released in a timely fashion by the client no extraneous buffer copies are required.

Output Buffer Requested Message

When the Assistant desires a frame, it again calls `mach_msg()` to message the plugIn, but this time it expects a response from the plugIn. The plugIn either provides a buffer or indicates that no data is available.

[1] Other locally defined debug messaging macros might be noticed as the CoreMediaIO source base is perused, but the use items as defined in `CMIODebugMacros.h`. is encouraged. They have the advantage of being able to melt away to `NOPs()` via a single file change, and be directed to any location (e.g, `syslog` or another file).

Debugging Tips

Debugging the PlugIn

The DAL will try to load any plugIn located in `/Library/CoreMediaIO/Plug-Ins/DAL/`.
To see if the plugIn is loading, the easiest thing to is to put a breakpoint at its factory function (`AppleCMIODPSampleNewPlugIn()` in `CMIO_DP_Sample_PlugIn.cpp`), and launch a video application such as Photo Booth. If you hit the breakpoint, all the mundane mechanical stuff about getting the plugIn built and installed is happening correctly.
The next point of interest would be to put a breakpoint at `CMIO::DP::Sample::PlugIn::InitializeWithObjectID()` (whose location can be inferred from the naming conventions to be `CMIO_DP_Sample_PlugIn.cpp`).
From there, the plugIn can be stepped through as desired.

Comment Out 'waittime' in MIG .defs File

When debugging anything that requires communication between the plugIn and Assistant, it is useful to turn off the Mach message timeouts so the debugger can stop either process.

Lazy Loading of PlugIn

The plugIn's .plist has a `CMIOHardwarePlugInLazyLoadingInfo` key whose value is a matching dictionary (or an array of matching dictionaries) which the DAL will invoke `IOServiceGetMatchingServices()` with. It will only load the plugIn in the event of match. The DAL will load the plugIn every time if the key is absent.

The sample's `Sample-Info.plist` has a '`NoCMIOHardwarePlugInLazyLoadingInfo`' placeholder in place. Remove the '`No`' and populate the matching dictionary as needed should lazy loading be desired.

Keeping the Assistant Alive After All Clients Disconnect

Normally, the Assistant exits when its last client has disconnected. However, this can be undesirable if the Assistant is all configured for debugging. To allow the Assistant to continue, modify the end of `Assistant::ClientDied()` (in `CMIO_DPA_Sample_Server_Assistant.cpp`) as indicated:

```
// If there are still clients connected to the Assistant, simply return
if (true /* not mClientInfoMap.empty() */)
   return;
```

Things to Watch Out For

Device's With Multiple Streams

Though much of the plumbing is in place to handle devices with multiple streams, there are still one or two places that are hard coded to use stream index 0. Until this is resolved, refrain from trying to use devices with more than one active stream.

Vend Your Audio through the HAL

Though a good deal of the plumbing is in place to vend audio streams through the DAL, do not attempt to do so since several key features are still lacking. Additionally, using the HAL allows Core Audio based programs to recognize the audio stream from the device.

Base SDK == Current Mac OS

The project is meant to build with the current System files and does not utilize any of the Xcode SDKs. If having build problems, first verify that this project setting is correct.