A Guide to the CoreMediaIO Example Sample DAL Plug-In

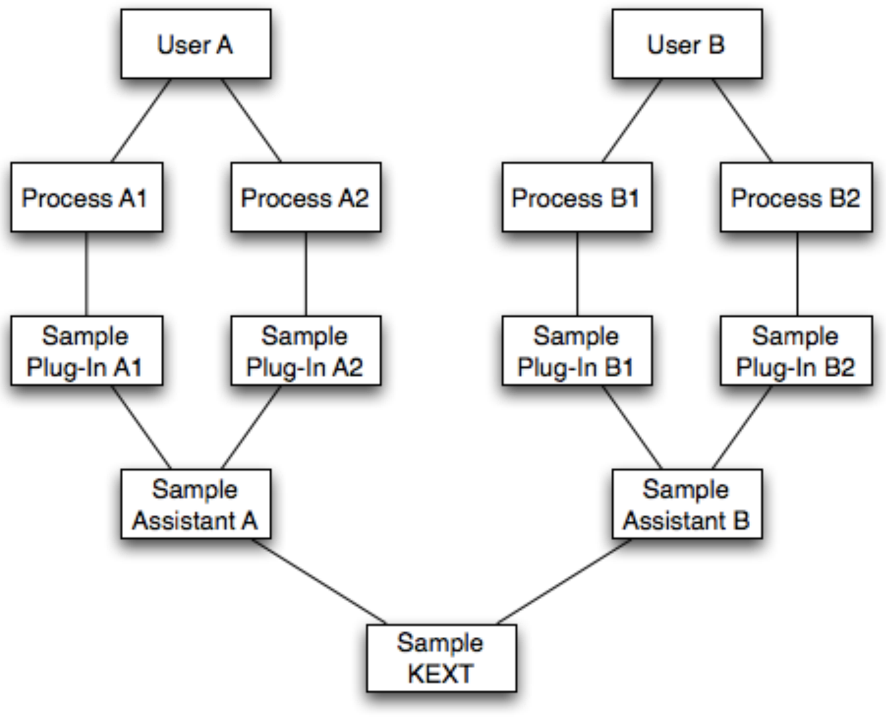
Introduction

The purpose of this document is to fully explore the Sample DAL plug-in that ships with the CoreMediaIO example. It is designed to be read after CoreMediaIO DAL Example.

The sample contains all three layers that are common to full-featured CoreMediaIO device support:

1. A user-level DAL plug-in.
2. A user-level “assistant” server process that allows the device to vend its video data to several processes at once (for example, device could be sending frames to an iChat video session and also be captured in a movie by Quick Time Player).
3. A kernel extension (KEXT) for manipulating the device’s hardware. Because its hard to download hardware from a website, the Sample KEXT simulates a video card by generating test-pattern video frames in the 2vuy format.

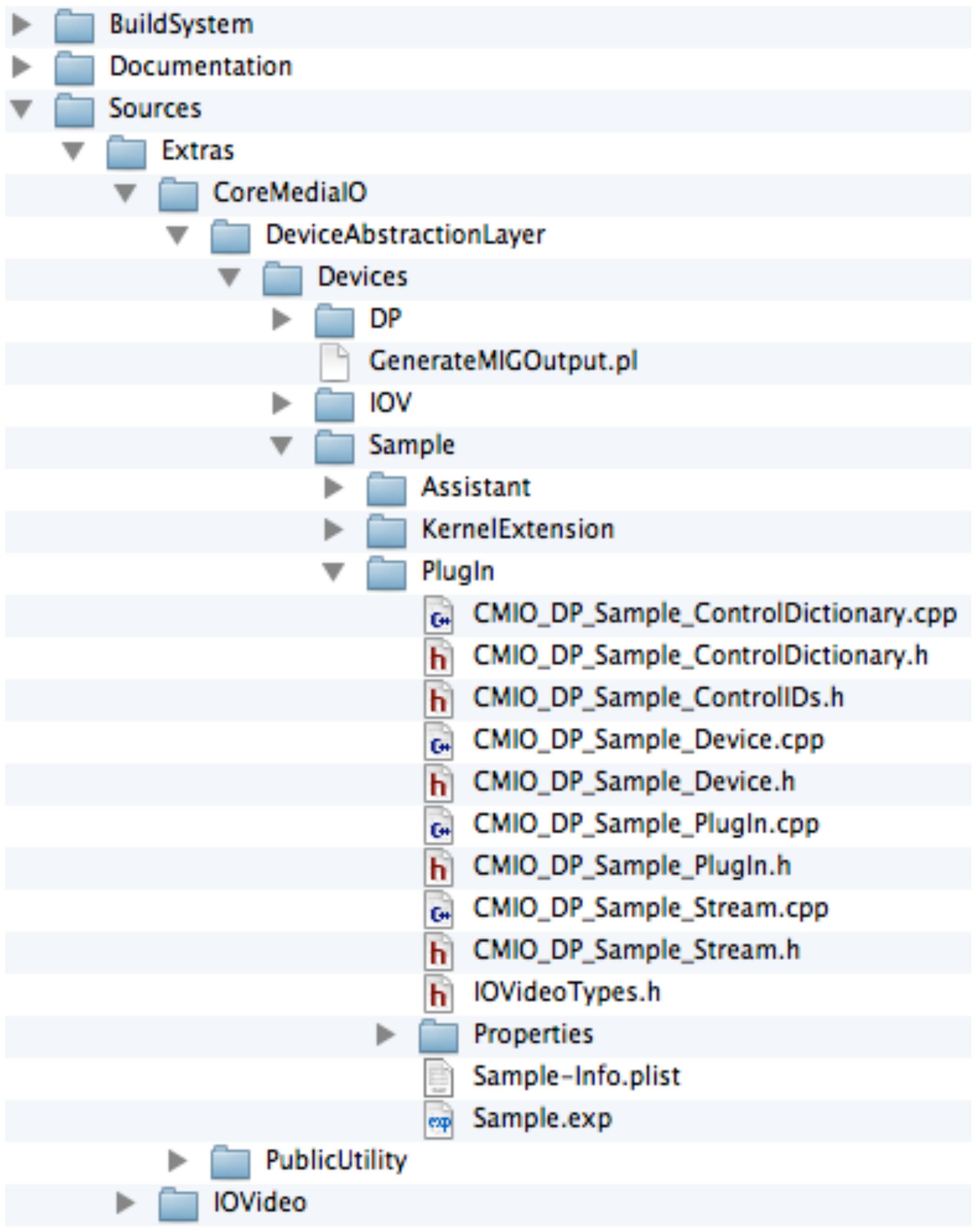
Each type of component has different rules that control its instantiation. There is an instantiation of the user-level plug-in for every process that wants to access the device. There will be an instantiation of the assistant for every user that has a process that instantiates the plug-in. Finally, there is one system-wide instantiation of the KEXT. This hierarchy can be viewed as follows:



Each of these components will be discussed in turn.

The DAL Sample Plug-In Overview

At a minimum, a DAL device must have a user-level DAL plug-in. The following files make up the plug-in portion of the DAL sample:



```
▶ BuildSystem
▶ Documentation
▼ Sources
    ▼ Extras
        ▼ CoreMediaIO
            ▼ DeviceAbstractionLayer
                ▼ Devices
                    ▶ DP
                      GenerateMIGOutput.pl
                    ▶ IOV
                    ▼ Sample
                        ▶ Assistant
                        ▶ KernelExtension
                        ▼ PlugIn
                              CMIO_DP_Sample_ControlDictionary.cpp
                              CMIO_DP_Sample_ControlDictionary.h
                              CMIO_DP_Sample_ControlIDs.h
                              CMIO_DP_Sample_Device.cpp
                              CMIO_DP_Sample_Device.h
                              CMIO_DP_Sample_PlugIn.cpp
                              CMIO_DP_Sample_PlugIn.h
                              CMIO_DP_Sample_Stream.cpp
                              CMIO_DP_Sample_Stream.h
                              IOVideoTypes.h
                            ▶ Properties
                              Sample-Info.plist
                              Sample.exp
            ▶ PublicUtility
        ▶ IOVideo
```

The DAL uses the `CFPlugIn` mechanism for loading plug-ins from `/Library/CoreMediaIO/Plug-Ins/DAL/`. As such, the plug-in only needs to export one symbol:

```
$ cat Sample.exp
# Only the PlugIn "Factory Function" needs to be exported
_AppleCMIODPSampleNewPlugIn
$
```

The factory method, `AppleCMIODPSampleNewPlugIn()`, is implemented in `CMIO_DP_Sample_PlugIn.cpp`. The `Sample-Info.plist` file specifies this function with the following key:

```
<key>CFPlugInFactories</key>
<dict>
    <key>E1E3EB8E-3D0F-49C9-8924-A8A40E1E1CAE</key>
    <string>AppleCMIODPSampleNewPlugIn</string>
</dict>
```

DAL plug-ins can be loaded during registration time or when their hardware is detected. To load when hardware is detected, the following is specified in the `Sample-Info.plist` file:

```
<key>CMIOHardwarePlugInLazyLoadingInfo</key>
<array>
      <dict>
            <key>IOProviderClass</key>
            <string>IOVideoSampleDevice</string>
      </dict>
</array>
```

This causes the DAL to invoke `IOServiceGetMatchingServices()` using the associated dictionaries leads to the the plug-in to be loaded when its associated hardware becomes known to the system. The IOServiceRegistry gets populated when the Sample KEXT's `start()` method override calls `registerService()` (in `IOVideoSampleDevice.cpp`).

The contents of the dictionary are specific to the type of device, and can specify more than just the provider class. This additional content are defined and populated by the device's IOKit Family. Here is an example of the values used for the Apple-provided VDC camera plug-in:

```
<key>CMIOHardwarePlugInLazyLoadingInfo</key>
<array>
    <dict>
        <key>IOProviderClass</key>
        <string>IOUSBDevice</string>
        <key>idProduct</key>
        <integer>34049</integer>
        <key>idVendor</key>
        <integer>1452</integer>
    </dict>
```

```
        <dict>
            <key>IOProviderClass</key>
            <string>IOUSBDevice</string>
            <key>bDeviceClass</key>
            <integer>239</integer>
            <key>bDeviceSubClass</key>
            <integer>2</integer>
            <key>bDeviceProtocol</key>
            <integer>1</integer>
        </dict>
    </array>
```

If for some reason the developer wants their plug-in to always load, `Sample-Info.plist` should either have the `CMIOHardwarePlugInLazyLoadingInfo` key/value pair removed, or its name changed to `NoCMIOHardwarePlugInLazyLoadingInfo`. Here is an example:

```
    <key>NoCMIOHardwarePlugInLazyLoadingInfo</key>
    <array>
        <dict/>
    </array>
```

In order to comply with Lion sandboxing policies dealing with server processes, the `Sample-Info.plist` must supply the service names of any assistant processes used. This is done by specifying the `CMIOHardwareAssistantServiceNames` key:

```
      <key>CMIOHardwareAssistantServiceNames</key>
      <array>
            <string>com.apple.cmio.DPA.Sample</string>
      </array>
```

If it did not specify `CMIOHardwareAssistantServiceNames`, the Sample would not be able to create its assistant server process.

In order for the DAL to interface with the C++ plug-in, `AppleCMIODPSampleNewPlugIn()` returns a reference to its `CMIOHardwarePlugInInterface`. This data structure provides callbacks that are used to call the plug-in's instance methods that implement required DAL plug-in functionality (such as initialize, teardown, etc.). In order to keep this list of callbacks manageable while still providing a measure of extensibility for 3rd parties, the interface includes a property mechanism.

The role of the plug-in object is to manage the devices that it abstracts. It is responsible for detecting the coming and going of its devices, starting and stopping device streams, and managing the buffer queue used by devices to source or sync data from DAL clients.

Outside of the plug-in, the devices are referred to by either a DAL Object ID or a Device GUID. The DAL Object ID is a non-persistent reference, meaning that the ID associated with a device only applies to the process in which the ID was created. The Device GUID on the other hand, can be used to identify a device in any process that has a reference to the device. For example, our Sample Device may be assigned device ID 258 in one process, while it could have ID 260 in another process. However, the GUID associated with the device will be the same in both processes.

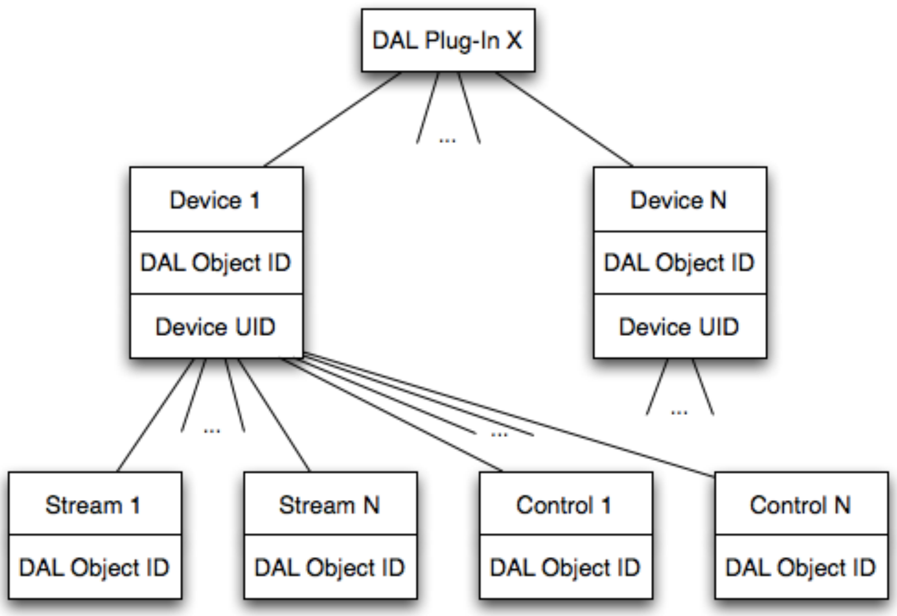
When the plug-in detects a new device, it creates a DAL object for it using `CMIOObjectCreate()` and adds it to its list of known devices. When it detects a device that is going away, it removes the device from the list and calls `CMIOObjectsPublishedAndDied()`, using the ID returned from the create call.

Internally, the plug-in can represent a device in any way it sees fit. The example has included a C++ base class that is designed to be extended by a plug-in to represent its devices, namely `DP::Device`. After the plug-in creates the object ID for the device, it creates the internal representation. The object ID and a Device GUID are provided to the internal representation, allowing for reverse lookups to be possible.

Devices present streams to clients. Just as the plug-in creates an ID for every device that it manages, every device creates an ID for every stream that it manages. The example provides the `DP::Stream` C++ base class to facilitate the internal representation of streams.

Devices have controls which modify their behavior. For example, a camera device might have a focus control. Like the other objects discussed, each control gets a DAL Object ID. Controls are self-describing. The device representation creates a list of control objects that can be queried and manipulated by clients. The example provides the `DP::Control` C++ base class to facilitate the internal representation of controls.

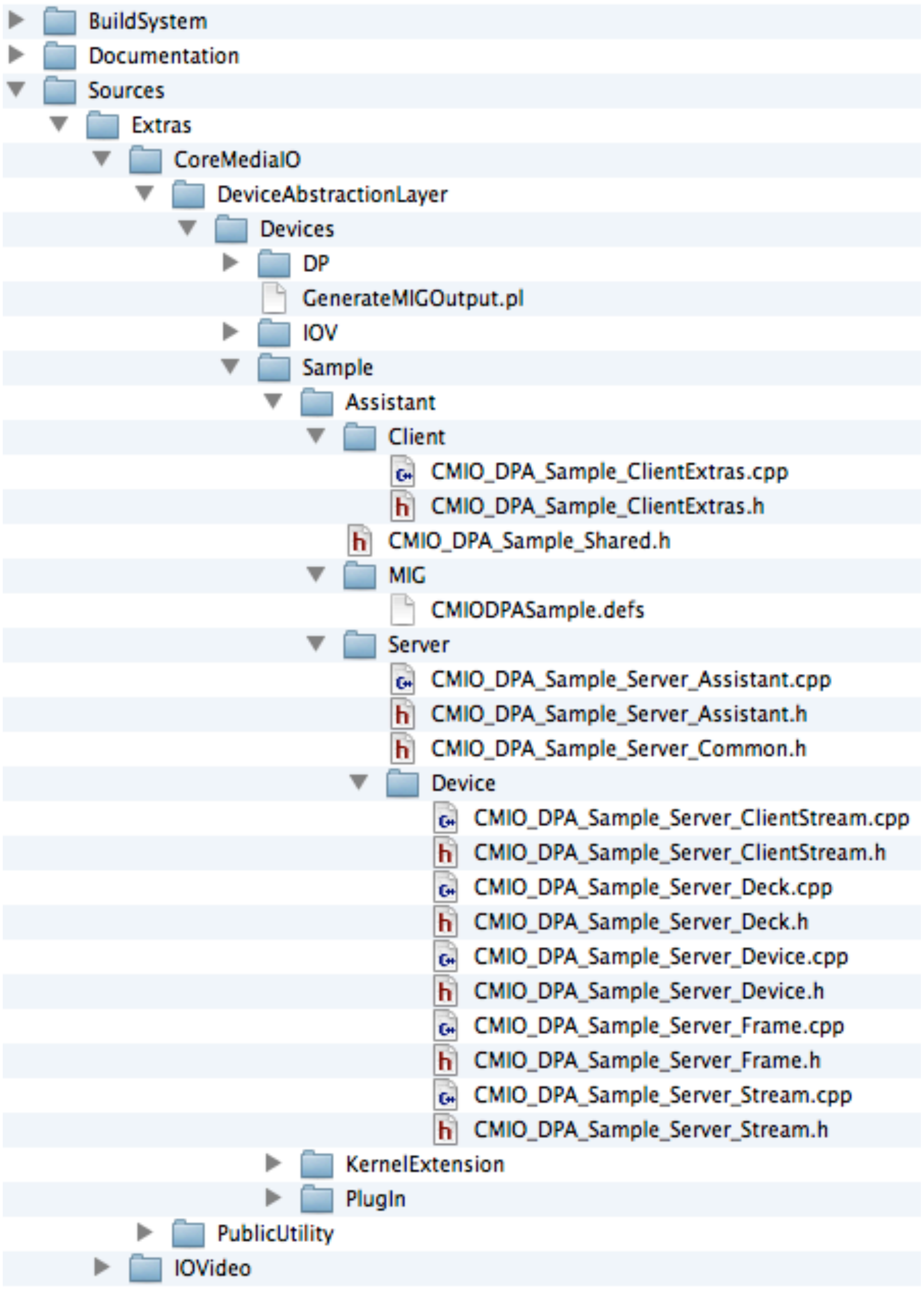
The plug-in/device/stream/control hierarchy can be viewed as follows:



A lot of the device functionality for the Sample is implemented in the assistant. The Sample's extension of the `DP::Device` class routes appropriate requests to the assistant. Communication is done using mach messaging.

The DAL Sample Assistant

If the developer would like their device to be used by several processes at once, then an "assistant" server process will be needed; if this piece is absent, then only one process at a time will be able to use the device. For example, PhotoBooth and FaceTime could both be getting input from the same camera at the same time. The following files make up the sample assistant:



```
▶ BuildSystem
▶ Documentation
▼ Sources
    ▼ Extras
        ▼ CoreMediaIO
            ▼ DeviceAbstractionLayer
                ▼ Devices
                    ▶ DP
                      GenerateMIGOutput.pl
                    ▶ IOV
                    ▼ Sample
                        ▼ Assistant
                            ▼ Client
                                CMIO_DPA_Sample_ClientExtras.cpp
                                CMIO_DPA_Sample_ClientExtras.h
                              CMIO_DPA_Sample_Shared.h
                            ▼ MIG
                                CMIODPASample.defs
                            ▼ Server
                                CMIO_DPA_Sample_Server_Assistant.cpp
                                CMIO_DPA_Sample_Server_Assistant.h
                                CMIO_DPA_Sample_Server_Common.h
                                ▼ Device
                                    CMIO_DPA_Sample_Server_ClientStream.cpp
                                    CMIO_DPA_Sample_Server_ClientStream.h
                                    CMIO_DPA_Sample_Server_Deck.cpp
                                    CMIO_DPA_Sample_Server_Deck.h
                                    CMIO_DPA_Sample_Server_Device.cpp
                                    CMIO_DPA_Sample_Server_Device.h
                                    CMIO_DPA_Sample_Server_Frame.cpp
                                    CMIO_DPA_Sample_Server_Frame.h
                                    CMIO_DPA_Sample_Server_Stream.cpp
                                    CMIO_DPA_Sample_Server_Stream.h
                        ▶ KernelExtension
                        ▶ PlugIn
            ▶ PublicUtility
        ▶ IOVideo
```

As noted above figure, there can be several processes that are using the Sample Device, and the object ID for the device may be different for each process. Because of this, the device's GUID is used to tell the assistant which device is an action's target. This allows an assistant to manage many devices.

Just as the plug-in has a set of classes to implement its various objects (`DP::Sample::Device` and `DP::Sample::Stream`), so does the assistant: `DPA::Sample::Server::Device` and `DPA::Sample::Server::Stream`.

The file `CMIODPASample.defs` defines the mach messages that are sent to the assistant. Here is an example:

```
/*
 SampleGetDeviceStates() - 60 second timeout
*/
waittime 60000;
routine SampleGetDeviceStates (
        clientSendPort      : mach_port_t;
    in  messagePort         : mach_port_t = MACH_MSG_TYPE_MAKE_SEND_ONCE;
    out states              : CMIOSampleUnboundedDeviceStateArray, dealloc
);
```

When the plug-in is built, a script runs that turns the definition into the following C++ function declaration:

```
/* Routine SampleGetDeviceStates */
#ifdef  mig_external
mig_external
#else
extern
#endif  /* mig_external */
kern_return_t CMIODPASampleGetDeviceStates
(
    mach_port_t clientSendPort,
    mach_port_t messagePort,
    CMIO::DPA::Sample::DeviceStatePtr *states,
    mach_msg_type_number_t *statesCnt
);
```

All of the routines defined in the `defs` file end up inside the build results folder in `CMIODPASampleClient.h` and `CMIODPASampleClient.cpp`, and are automatically included in the build of the plug-in. The file `CMIO_DPA_Sample_ClientExtras.h` includes `CMIODPASampleClient.h`, and provides convenience functions couched in the vernacular of the DAL object hierarchy. For example, `CMIO_DPA_Sample_ClientExtras.cpp` has the following:

```
void GetDeviceStates(mach_port_t port, mach_port_t messagePort,
              AutoFreeUnboundedArray<DeviceState>& deviceStates)
{
    IOReturn ioReturn =
              CMIODPASampleGetDeviceStates(port, messagePort,
                  deviceStates.GetAddress(), &deviceStates.GetLength());
    ThrowIfKernelError(ioReturn, CAException(ioReturn),
              "CMIO::DPA::Sample::ClientExtras::GetDeviceStates:");
}
```

In this fashion, if the DAL plug-in wants to get the devices states from the assistant, it need only call `DPA::Sample::GetDeviceStates()`.

For the assistant, again a script is run to turn the `defs` file into code. This time, the files are `CMIODPASampleServer.h` and `CMIODPASampleServer.cpp`. And again, `CMIODPASampleGetDeviceStates()` is defined. However, instead of implementing the receiver version of the function, the internal shell receiver function `_XSampleGetDeviceStates()` is created in `CMIODPASampleServer.cpp`, which calls `CMIODPASampleGetDeviceStates()`. The file `CMIO_DPA_Sample_Server_Assistant.cpp` implements `CMIODPASampleGetDeviceStates()`. This version of `CMIODPASampleGetDeviceStates()` is a simple wrapper for calling into the assistant proper:

```
kern_return_t CMIODPASampleGetDeviceStates(mach_port_t client,
                  mach_port_t messagePort, DeviceState** deviceStates,
                  mach_msg_type_number_t* length)
{
    return Assistant::Instance()->GetDeviceStates(client, messagePort,
                  deviceStates, length);
}
```

The assistant's `GetDeviceStates()` method is also defined in `CMIO_DPA_Sample_Server_Assistant.cpp`.

To recap our example function call:

- The plug-in needs to get the device states.
- This information is known by the assistant.
- The plug-in calls `DPA::Sample::GetDeviceStates()` from `CMIO_DPA_Sample_ClientExtras.cpp`.
- This calls `CMIODPASampleGetDeviceStates()` from `CMIODPASampleClient.cpp`, which was generated from `CMIODAPSample.defs`.
- This version of `CMIODPASampleGetDeviceStates()` initiates a mach remote procedure call.
- The remote procedure call is received by the assistant and routed into `_XSampleGetDeviceStates()` from `CMIODPASampleServer.cpp`, which was generated from `CMIODAPSample.defs`.
- `_XSampleGetDeviceStates()` calls the version of `CMIODPASampleGetDeviceStates()` from `CMIO_DPA_Sample_Server_Assistant.cpp`.

- This calls into the assistant's version of `GetDeviceStates()`, also from `CMIO_DPA_Sample_Server_Assistant.cpp`, which provides the result.
- The result is returned back through the mach remote procedure call machinery to the plug-in.

Usually, its the plug-in that is calling into the assistant for something. But when it comes to providing buffers that convey data, the assistant needs to notify the plug-ins that something is ready to be processed. For input streams, these buffer are filled with data from the device. For output streams, these buffers are empty and need to be filled with data for the device.

The abstraction for a Sample device's data stream is provided in `CMIO_DPA_Sample_Server_Stream.cpp`. One of the first things that it does is to register an output callback with the KEXT abstraction (`StreamOutputCallback`). For input streams, this callback will be called every time a frame is available from the KEXT. For output streams, it would be called every time the KEXT has a free buffer to be filled with output data.

Every plug-in that has connected to the assistant has a set of device streams that are active. In the plug-in, this abstraction is provided by `CMIO_DP_Sample_Stream.cpp`. Its important to note an intentional limitation of the Core Media IO architecture. Specifically, more than one plug-in can be supplied by an assistant's input stream (each plug-in gets an identical copy of the data), but only one plug-in can supply data for an output stream. It would be very confusing if several processes could send data to an output stream at the same time. Therefore, for the input case, there can be many `CMIO_DP_Sample_Stream` instantiations for each instantiation of `CMIO_DPA_Sample_Server_Stream`.

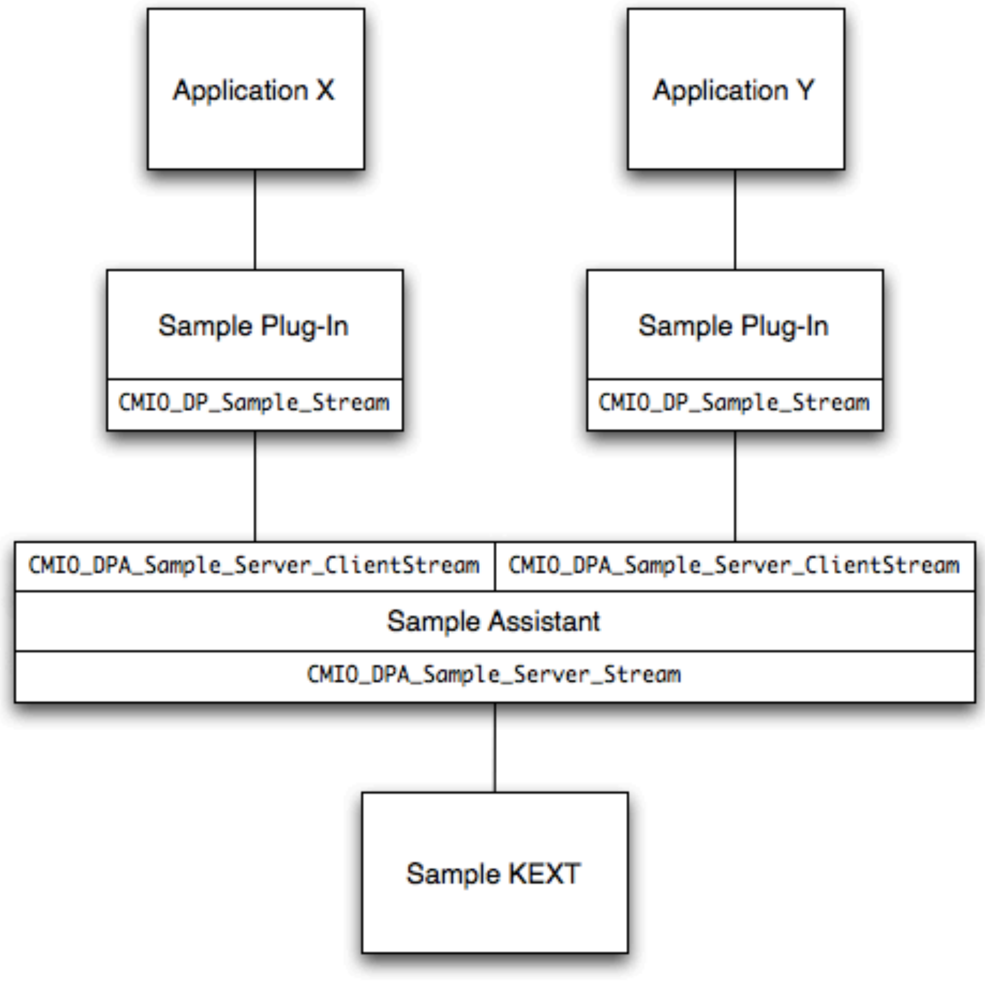
The plug-in's abstraction in `CMIO_DP_Sample_Stream.cpp` also manages moving buffers between it and its client application. Because the client application runs on its own set of threads, there is a queue that is used to store buffers until such time as they can be processed. The client application calls `CopyBufferQueue()` to get a reference to the queue, and to provide a callback for notification that a new buffer has been put on the queue (or in the case of output, that a has been taken off of the queue).

Inside the assistant, each active stream for each plug-in has its own queue. The queue is a place to store buffers as they are moved between assistant and plug-in. Each queue has a messaging thread that is used for notifications about the queue. `CMIO_DPA_Sample_Server_ClientStream.cpp` encapsulates everything to connect a stream to a plug-in. There is a one-to-one correspondence between plug-in based `CMIO_DPA_Sample_Server_ClientStream` instantiations for each `CMIO_DP_Sample_Stream` instantiation.

To recap the difference between what is in `CMIO_DPA_Sample_Server_Stream.cpp` versus `CMIO_DPA_Sample_Server_ClientStream.cpp` versus `CMIO_DP_Sample_Stream.cpp`:

- The assistant's abstraction for every stream of data between the KEXT and the assistant is implemented by `CMIO_DPA_Sample_Server_Stream.cpp`.
- The assistant's abstraction for every stream of data between the assistant and the plug-in is implemented by `CMIO_DPA_Sample_Server_ClientStream.cpp`.
- The plug-in's abstraction for every stream of data between the assistant and the plug-in is implemented by `CMIO_DP_Sample_Stream.cpp`.

Here is a picture of their relative positions in the object hierarchy, assuming that input is being performed:



Focusing on the input scenario, when the KEXT generates a new frame that needs to be processed, it must get this frame to the plug-in. The frame's details are sent using a mach message to each plug-in that is currently sourcing from the frame's stream.

Delivery starts out when the KEXT calls the registered handler (`CMIO_DPA_Sample_Server_Stream.cpp`'s `StreamOutputCallback()`) with each new frame. The handler calls `FrameArrived()`, which converts the arrived frame's token and control information into an internal `Frame` data structure, and puts the construct on each of the related plug-in queues managed by `ClientStream` instantiations.

Each `ClientStream` message thread is signaled, causing `MessageThreadEntry()` in `CMIO_DPA_Sample_Server_ClientStream.cpp` to call `SendFrameArrivedMessage()`, which sends a mach message indicating the frame's particulars to the plug-in.

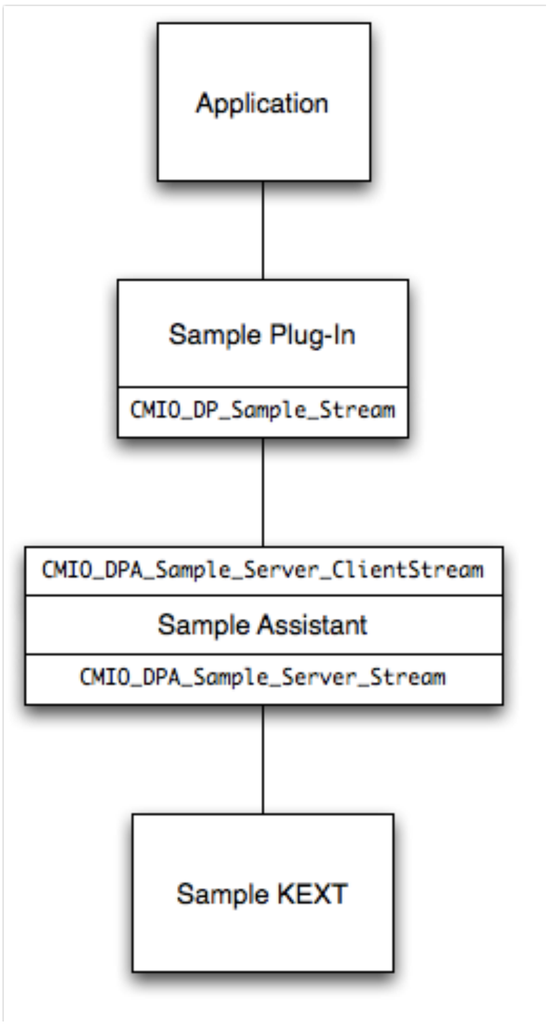
The plug-in's mach message handler, `Messages()` in `CMIO_DP_Sample_Stream.cpp`, calls `FrameArrived()`. Ideally, at this point a Core Media Sample Buffer is created so that the frame can be further processed by the client application. Since the application runs on its own set of threads, there is a queue to store the Sample Buffer until such time as it can be processed. When a buffer is placed on this queue the client application's notification callback is invoked.

However, if the delivery queue has filled to capacity (typically due to excessive machine load preventing the client application from processing buffers in an expedient fashion), then the buffer must be dropped. Because the Sample Device discussed here only delivers I-Frame buffers, a dropped buffer does not necessarily have to create a disruption in the stream; the dropped buffer's timing information is saved off and is used to modify the next saved buffers information (in other words, the hole in the stream timing caused by the dropped buffer is filled in). A flag is set on the buffer that can be examined by client applications interested in getting every frame, so that they can react accordingly to a purely non-contiguous stream of buffers. For example, a high-end capture application will want to capture every buffer from a device, while a video conferencing application should not care if a buffer is dropped here or there.

To recap how a frame is delivered:

- The KEXT captures a frame (or in the case of the Sample Device, a frame is generated at a prescribed interval).
- The KEXT calls `IOStream`'s `sendOutputNotification()`, which calls the registered handler, the assistant's `CMIO_DPA_Sample_Server_Stream.cpp`'s `StreamOutputCallback()`.
- Because the stream is an input stream, `StreamOutputCallback()` calls `FrameArrived().`
- The frame token and information is stored in an internal `Frame` data structure, which is put on relevant queues controlled by `CMIO_DPA_Sample_Server_ClientStream` objects (one object per plug-in that is currently sourcing the stream). A signal is sent that new data is on the queue(s).
- The signal is caught and control works it way to `CMIO_DPA_Sample_Server_ClientStream.cpp`'s `SendFrameArrivedMessage()`, which sends the data in a mach message to the plug-in.
- The message is received by the plug-in's mach message handler, `Messages()` in `CMIO_DP_Sample_Stream.cpp`
- `Messages()` calls `FrameArrived()`.
- `FrameArrived()` builds a Core Media Sample Buffer that is put on a queue for processing by the client application, and the client's `CMIODeviceStreamQueueAlteredProc` callback is invoked so that it may pull the data off of the queue.

With respect to the output scenario, as mentioned above, only one client can supply the data for a given stream:



The plug-in's stream object supplies a queue for the client application to fill, using the same API as for capture: `CMIO_DPA_Sample_Server_Stream.cpp`'s `CopyBufferQueue()`. However, for the output case, the supplied `CMIODeviceStreamQueueAlteredProc` callback is invoked when the stream has removed a buffer from the queue for sending to the device (the Sample KEXT in this case); the client application uses this invocation to trigger the process it uses to fill and add another buffer to the queue.

But there is an important first step that the client application needs to undertake before submitting its first buffer to the queue. In order to prevent "start-up" glitches, the client needs to call the stream's `GetOutputBuffersRequiredForStartup()`. Once the initial buffers are loaded, it adds them to the queue all at once. It can then can go about filling up the queue to its limit, which is given by `GetOutputBufferQueueSize()`. As mentioned above, the client uses the `CMIODeviceStreamQueueAlteredProc` callback to know when a new slot is available in the queue.

When the KEXT is running an output stream, whenever it needs a new buffer to output, it will call `IOStream`'s `sendOutputNotification()`, which calls the registered handler to supply a buffer. In this case, the assistant's stream object, `CMIO_DPA_Sample_Server_Stream.cpp`, has registered a handler, which causes `GetOutputBuffer()` to be called. This method sends a mach message to the plug-in (`kOutputBufferRequested`). The `Messages()` listener in `CMIO_DP_Sample_Stream.cpp` gets the message and invokes its version of `GetOutputBuffer()`, which pulls a buffer off of the queue that is fed by the client, calls the `CMIODeviceStreamQueueAlteredProc` callback (to trigger it to generate a new buffer for the queue), and messages the buffer to the assistant.

Here is a recap of how output is performed:

- The client application gets the queue for buffers by calling `CMIO_DP_Sample_Stream.cpp`'s `CopyBufferQueue()`, and supplies a `CMIODeviceStreamQueueAlteredProc`.
- The client application calls `CMIO_DP_Sample_Stream.cpp`'s `Start()`, and also starts its own thread to fill the queue.
- The client application's thread fills the initial set of buffers, the number of which is provided by `GetOutputBuffersRequiredForStartup()`. Once these initial buffers are available, it places them onto the buffer queue.
- The client application's thread keeps supplying buffers to the queue until it is full, as according to the number returned by `GetOutputBufferQueueSize()`.
- Meanwhile, whenever the KEXT needs a buffer, it calls `IOStream`'s `sendOutputNotification()`, which calls the registered handler, the assistant's `CMIO_DPA_Sample_Server_Stream.cpp`'s `StreamOutputCallback()`
- Because the stream is an output stream, `StreamOutputCallback()` calls `GetOutputBuffer().`
- `GetOutputBuffer()` sends a `kOutputBufferRequested` message to the plug-in.
- The `kOutputBufferRequested` is received by `CMIO_DP_Sample_Server_Stream`'s `Messages()` listener.
- `Messages()` calls `GetOutputBuffer()`.
- `GetOutputBuffer()` pulls a buffer off of the queue that is fed by the client application, calls the `CMIODeviceStreamQueueAlteredProc` callback, and messages the buffer to the assistant.

A Brief Note About Clocks

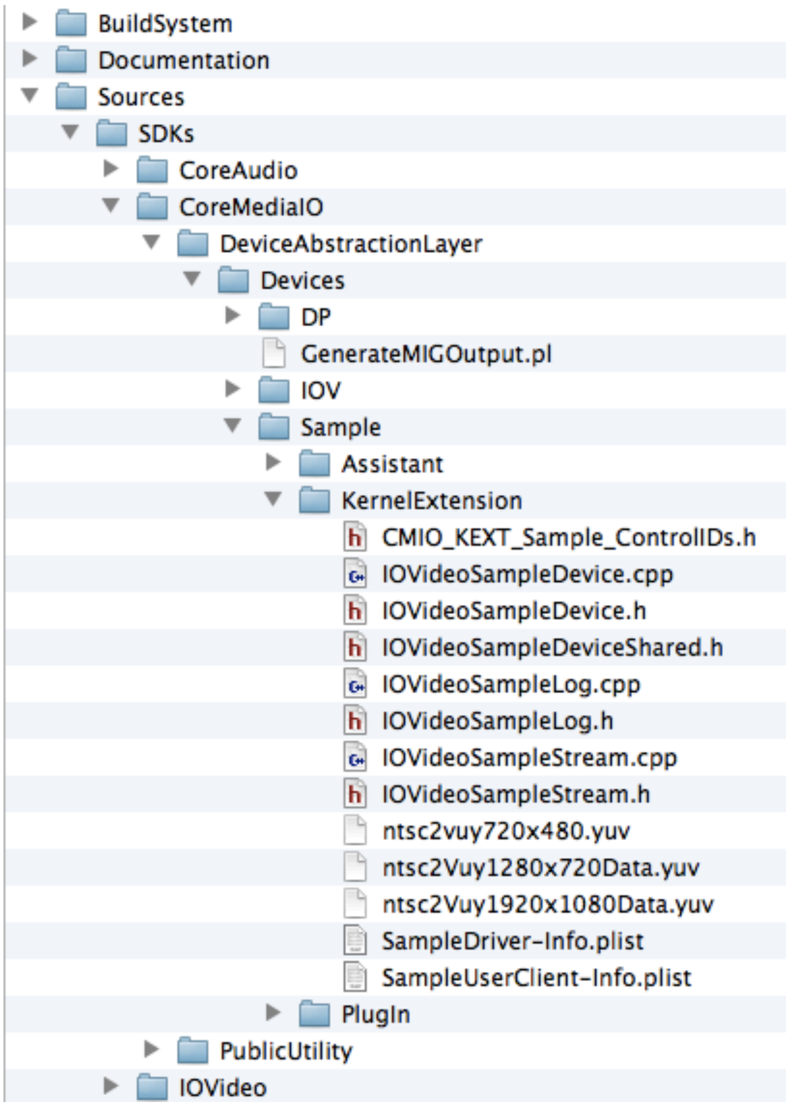
In order for proper A/V sync to be maintained when using multiple devices for a particular purpose, CoreMediaIO requires that all DAL devices publish a clock object. Clocks perform two important functions to the system: they are able to convert between a device's concept of time to hosttime (and back), and they are able to calculate a rate at which they are running with respect to hosttime. By having hosttime be the "common denominator" for all clocks, media from different devices can be coordinated.

Currently, the clock object implementation is fairly opaque to device writers. But the clocking requirements of device drivers is pretty basic: they need to create the clock, drive the clock, and occasionally convert a time on their clock time to hosttime. All of the synchronization is handled above the plug-in. Therefore, there are only four routines needed for device driver writers: `CMIOStreamClockCreate()`, `CMIOStreamClockInvalidate()`, `CMIOStreamClockPostTimingEvent()`, and `CMIOStreamClockConvertHostTimeToDeviceTime()`. These routines are described in `CMIOHardwareStream.h`, and used in `CMIO_DP_Sample_Stream.cpp`.

The DAL Sample Kernel Extension

If the device has hardware that needs to be manipulated at the kernel level, then a kernel extension (KEXT) is required. If a KEXT is not required, then device manipulation will be either in the assistant (if present) or in the plug-in itself.

The following files make up the Sample KEXT:


```
▶ BuildSystem
▶ Documentation
▼ Sources
    ▼ SDKs
        ▶ CoreAudio
        ▼ CoreMediaIO
            ▼ DeviceAbstractionLayer
                ▼ Devices
                    ▶ DP
                      GenerateMIGOutput.pl
                    ▶ IOV
                    ▼ Sample
                        ▶ Assistant
                        ▼ KernelExtension
                              CMIO_KEXT_Sample_ControlIDs.h
                              IOVideoSampleDevice.cpp
                              IOVideoSampleDevice.h
                              IOVideoSampleDeviceShared.h
                              IOVideoSampleLog.cpp
                              IOVideoSampleLog.h
                              IOVideoSampleStream.cpp
                              IOVideoSampleStream.h
                              ntsc2vuy720x480.yuv
                              ntsc2Vuy1280x720Data.yuv
                              ntsc2Vuy1920x1080Data.yuv
                              SampleDriver-Info.plist
                              SampleUserClient-Info.plist
                        ▶ PlugIn
            ▶ PublicUtility
        ▶ IOVideo
```

Since the Sample KEXT isn't backed by any real hardware, it instead reads pre-generated video frames from an internal data section that is generated from the `*.yuv` files listed above. This is why the KEXT weighs in a 1.2GB; its not really that complicated! There are three frame sizes represented, and each data section contains 60 frames of data.

The Sample KEXT is based upon the `IOVideo` IOKit classes, which were designed for CoreMediaIO devices. The Sample Device is represented as a subclass of `IOVideoDevice`.

An `IOVideoDevice` hosts any number of `IOVideoStreams`; `IOVideoStream` is a subclass of `IOStream`. The `IOStream` class defines a mechanism for moving buffers of data from kernel space to user space and vice-versa. The policy for which direction the data flows and the nature of the data is left up the the implementer of the driver.

The data that flows in an `IOStream` is contained in `IOStreamBuffers`. Managing the lifecycle of an `IOStreamBuffer` can be tricky, as the buffer can be given to multiple clients, and the contents of the buffer can be retained by a client for an arbitrary time. The driver should not expect buffers to be returned in the order in which they are handed out, or within any given timeframe.

`IOStreamBuffers` are referenced by an opaque `IOStreamBufferID`, as any type of memory reference would not be valid in both kernel and user address spaces.

An `IOStreamBuffer` wraps two memory references, one for data and one for control information. The control information is defined by the driver. One use of the control information is to return basic metadata about the contents of the data buffer. For example, the Sample Device uses the following structure to define its control information:

```
struct SampleVideoDeviceControlBuffer
{
    UInt64  vbiTime;
    UInt64  outputTime;
    UInt64  totalFrameCount;
    UInt32  droppedFrameCount;
};
```

The values in a control buffer must be types that can cross the kernel boundary (objects based on scalar types, or on `OSObject`, such as `OSArray`).

An `IOStream` has two `IOStreamBufferQueues`, one for sending `IOStreamBuffers` from kernel space to user space, and one for sending `IOStreamBuffers` from user space to kernel space. Depending on whether or not a stream is used for input or output determines which of the queues will have buffers waiting to be filled versus filled buffers waiting to be processed.

It is assumed that all buffers in a given stream have the same format. The `IOVideoStreamDescription` structure is used for this specification. It is serialized into an `OSDictionary` by the `IOVideoStreamDictionary` API. If a device changes format for some reason, the existing stream needs to be stopped, and the client will need to start a new stream.

In addition to specifying streams for the device, `IOVideoDevice` is used to specify any controls that the device has. The Sample Device has a boolean control that is used to specify if it is configured for input or output, as well as a selector control for input (“Tuner”, “Composite”, or “S-Video”).

Most of the code is contained in `IOVideoSampleDevice.cpp`, with `IOStream` overrides and support provided by `IOVideoSampleStream.cpp`. But before going into details, its important to note two things. First, the terms “input” and “output” when used with the KEXT are interpreted from the position of the KEXT. “Input” means “input to the KEXT”, which for CoreMediaIO purposes, means that the device is being used for display (and from the vantage point of CoreMediaIO, output). Likewise, “output” means “output from the KEXT”, which means the device is being used for capture (vantage point CoreMediaIO: input). The second thing to note is that the KEXT calls the same routine to notify its client about frames that have been captured or buffers that are available to be filled: `sendOutputNotification()`; this routine probably could have been named better.

Initialization happens in the `IOVideoSampleDevice::start()` method, where references to the embedded sample frames are obtained. Along with creating the control buffers, IOVideoSampleDevice::AllocateFrameBuffers() is called to create space in kernel memory to hold the sample frames, and then IOVideoSampleDevice::LoadFrameBuffer() is called repeatedly to copy the frames from the data section. Once the frames are loaded, the device's controls are created, and then depending on the current direction of the device, either input or output streams are created using AddInputStreams() and AddOutputStreams(). A timer event is added to the device's workloop for generating frames (when used as an input device) or requesting frames (when used as an output device). In a real device, an interrupt source would be used. Finally, registerService() is called to allow others to discover the device.

The Sample Device supports only one input stream. It is created by the IOVideoSampleDevice::AddInputStreams() method, and stored in the _outputStream instance variable (for historical reasons based on the presentation of Core Audio drivers, this nomenclature is chosen to reflect the "name mirroring" discussed above). An array is created to hold references to the buffers that are going to be sent to the clients. Once the array is created, it is given to the _outputStream, and the stream is added to the IOKit registry.

The IOVideoSampleDevice::timerFired() method is called at the frame rate of the simulated device, and for the case where the device is providing frames for capture, uses the IOVideoSampleDevice::sendOutputFrame() method to send a frame to the assistant.

The IOVideoSampleDevice::sendOutputFrame() method does several things. First, it gets the current hosttime, which will be used by the Sample plug-in to drive the stream's clock object. Next a buffer is needed to send to the client. Since we are sending the same data over and over, the Sample KEXT simply reuses the same pre-loaded buffers. It gets them by calling IOStream::getBufferWithID() to convert an ID to a buffer. It then writes the hosttime at the start of the buffer's associated control buffer (the IOStreamBuffer::getControlBuffer() method is used). The data and the control buffer are made available to the Sample assistant with the IOStream::enqueueOutputBuffer() method, and the assistant is notified by the KEXT calling IOStream::sendOutputNotification(). Finally, the KEXT updates the index (buffer ID) used to cycle through the pre-loaded buffers.

For the output scenario, the Sample KEXT doesn't do very much, as there is no real hardware to play the buffer on. However, it does demonstrate using two OSArrays to manage buffer lifecycles: member variables IOVideoSampleStream::_freeBuffers and IOVideoSampleStream::_filledBuffers. When the stream is created, IOVideoSampleStream::initWithBuffers() fills up _freeBuffers.

When the timer fires, the IOVideoSampleDevice::consumeInputFrame() method is called. Like its peer, the first thing done is to get the current hosttime. It then tries to get a frame filled by the assistant by calling the IOVideoSampleStream::getFilledBuffer() method, which gets a buffer from the _filledBuffers member variable. Since no output is actually being done, the KEXT immediately puts the buffer onto the free queue via the IOVideoSampleStream::returnBufferToFreeQueue(), and calls the IOVideoSampleStream::postFreeInputBuffer() method to make the buffer available to the client.

One thing that IOVideoSampleStream::postFreeInputBuffer() does is to fill in the control information so that the Sample plug-in can run its device clock and monitor the status of the output.

It makes the buffer available to the assistant via the `enqueueOutputBuffer()` method, and notifies the assistant by calling the `sendOutputNotification()` method.

The way that the buffers get back onto the `_filledBuffers` queue is via `IOVideoSampleStream::inputCallback()` override of `IOStream::inputCallback()`. The callback gets triggered when the Sample assistant calls `EnqueueInputBuffer()` followed by `SendInputNotification()`. The callback calls `IOStream::dequeueInputEntry()` to get the next filled buffer from the plug-in, and places it on `_filledBuffers`.