Using IOMX in Android

1. Introduction

Android is a software stack that includes an operating system, middleware and key applications. Currently, Android is the top free open source operating system used in various products like Smart phones, Feature phones, Tablets, Set top boxes etc. It has very rich support for multimedia related functionalities.

Audio/Video/Image codecs, Audio/Video renderer are the most important building blocks of any multimedia system. Again video related modules have the highest importance during system design, as they are the most computation heavy elements in a multimedia system. Video components are always hardware accelerated and hence very much platform dependent on handheld devices.

Khronos Group developed OpenMAX IL standard for standardizing the interfaces for A/V/I codecs & renderer software components. Almost all handheld devices use OpenMAX IL compliant multimedia components in their system. We referred only the OpenMAX Integration Layer in this entire document.

Android's multimedia functionality uses a client server architecture where the OMXIL components are owned by mediaserver process. In order to access the OMXIL functionality from a client process, such as a multimedia application, Android provides IOMX interface. The present document briefs on how to create an IOMX based media player for a released non routed android phone. We will consider only H264 decoding in this document.

This document is targeted for people who already have basic knowledge on OMXIL.

2. Application

Android application is written in JAVA and it has the necessary graphical user interface. It creates the display window and manages user inputs. The main multimedia functionality runs on the middleware layer, which is written in C++. Application layer controls the middleware based on user inputs though. IOMX player application will essentially be a normal media player application having common play, pause, stop, seek kind of functionalities.

3.1 Creation and usage of surface

Surface object is created in the Application and accessed by native. The following code snippet shows how we are referencing the surface object.

3.1.1 JAVA side code

```
public Surface mSurface; // accessed by native methods
public void setDisplay(SurfaceHolder sh) {
            mSurfaceHolder = sh;
            if (sh != null) {
                  mSurface = sh.getSurface();
}
```

3.1.2 JNI code

We get the id of the surface object which is created by the App.

```
static void
com_vedio_player_MediaPlayerTest_native_init(JNIEnv *env)
{
      ....

jfieldID fields.surface = env->GetFieldID(jclass clazz, "mSurface",
Landroid/view/Surface;");

}
```

We refer the surface object using the surface id and pass it to the native MediaPlayer.

```
static void
com_vedio_player_MediaPlayerTest_start(JNIEnv *env, jobject thiz)
{
    LOGV("start");
    //mediaplayertest *mp = new mediaplayertest();

    ...

    jobject surface = env->GetObjectField(thiz, fields.surface);
    mp->start(env, surface);
}
```

3.2 Passing the file path

Typical way to pass the file to be played to the middleware layer is to implement something like setDataSource(string filePath) API. Since the file to be played is chosen by user, the path is known to the UI application and it's passed to the middleware layer using JNI.

3. Middleware

Interfacing of application with middleware layer written in C++ is done through Java Native Interface (JNI). The display window is created by application and is denoted by a surface object. The surface object is passed down to the native layer through JNI.

3.1 Video file parsing

The video file under playback has to be parsed to extract metadata and encoded video frames from it. This is done through software file parser modules. Several file parsers are openly available to parse various video file formats. Those can be integrated in the middleware layer easily.

3.2 Using IOMX interface

Video playback use case requires accessing video decoder and video renderer to be accessed through IOMX interface. This section describes how to use the IOMX interface through native code.

3.2.1 Getting the IOMX interface

IMediaPlayerService in mediaserver implements getOMX() function. This can be used to get the IOMX interface as shown in the following code snippet.

```
sp<IServiceManager> sm = defaultServiceManager();
sp<IBinder> binder = sm->getService(String16("media.player"));
sp<IMediaPlayerService> service = interface_cast<IMediaPlayerService>(binder);
sp<IOMX> mOMX = service->getOMX();
```

3.2.2 Listing available OMX components

OMX components are identified by their name and role. Name of a component defined by the implementer of the component. It often contains short stings to indicate the functionality and implementer's name. As an example a component with name "omx.qcom.avcdec" signifies that the component is developed by Qualcomm and it does AVC (H264) video decoding. The following code snippet shows how to list all the available OMX components in a system.

```
List<IOMX::ComponentInfo> componentInfos;
status_t err = mOMX->listNodes(&componentInfos);

List<IOMX::ComponentInfo>::iterator it = componentInfos.begin();
const IOMX::ComponentInfo &info = *it;
const char *componentName = info.mName.string();

for (List<IOMX::ComponentInfo>::iterator it = componentInfos.begin();
     it != componentInfos.end(); ++it)
{

   const IOMX::ComponentInfo &info = *it;
   const char *componentName = info.mName.string();

   for (List<String8>::const_iterator role_it = info.mRoles.begin();
        role_it != info.mRoles.end(); ++role_it)
   {
      const char *componentRole = (*role_it).string();
      LOGD("componentName: %s, componentRole: %s\n", componentName, componentRole);
   }
}
```

3.2.3 Selecting the required OMX component

Name of an OMX component is of implementer's choice. It's difficult to select a component based on its name, whereas role of a component is a standard string provided by OMX standard. Hence selecting a component is done based on role. In our current example we have to select a component which has role "video_decoder.avc". Note that a single component can have multiple roles and multiple components have same role. Though these cases are rare, but the component selection mechanism should take care of those. OMX components are loaded dynamically as shown in the following code snippet.

```
IOMX::node_id mNode;

mOMX->allocateNode(compName, this, &mNode);
```

Here compName is a "C" like string having the name of the component to be loaded. The purpose of passing "this" pointer is to handle the call backs from the component which is described later.

3.2.4 Memory allocation

Input/Output buffers for data processing by an OMX component can be done in two ways. Either the client (the layer which calls OMX component) can allocate the buffers and provide to the component through useBuffer() or the client can instruct the component to allocate the buffers by calling allocateBuffer().

Video buffers are very large in general. Hence those are not copied between layers normally. Often hardware accelerated video components deal with buffers those can be accessed from Kernel space only. In such cases allocateBuffer() may create buffers with physical address which can't be accessed from user space (middleware layer) at all. Calling useBuffer() will be helpful if the buffer is needed to be accessed from user space.

```
sp<MemoryDealer> mDealer = new MemoryDealer(16 * 1024 * 1024, "OmxH264Dec");
sp<IMemory> mMemory = mDealer->allocate(inBufLen);
IOMX::buffer_id mID;
uint32_t mInPortIndex = 0;
uint32_t mOutPortIndex = 1;


for(i = 0; i < numInBuf; i++)
{
   mOMX->useBuffer(mNode, mInPortIndex, mMemory, &mID);
   mBufferHandler->registerInBuf(mMemory, mID)
}

for(i = 0; i < numOutBuf; i++)
{
   mOMX->useBuffer(mNode, mOutPortIndex, mMemory, &mID);
   mBufferHandler->registerOutBuf(mMemory, mID)
}
```

The purpose of registering the buffers to the buffer handler class is explained in the buffer handling section later.

3.2.5 Configuring the component

OMX components need to be configured before being used. Configuring the I/O ports is minimum requirement for any component. This can be done when the component is in loaded state.

The following code snippet shows how to configure the ports.

```
OMX_PARAM_PORTDEFINITIONTYPE portDefn;
portDefn.nPortIndex = mInPortIndex;

mOMX->getParameter(mNode, OMX_IndexParamPortDefinition, &portDefn, sizeof(portDefn));

portDefn.nBufferCountActual = mInBufCnt;        // set some suitable value here or don't update to
                                                   use default value
portDefn.format.video.nFrameWidth = vidWidth;   // width of the video to be played
portDefn.format.video.nFrameHeight = vidHeight; // height of video to be played
portDefn.format.video.nStride = vidWidth;
portDefn.format.video.nSliceHeight = vidHeight;

mOMX->setParameter(mNode, OMX_IndexParamPortDefinition, &portDefn, sizeof(portDefn));
```

```
portDefn.nPortIndex = mOutPortIndex;

mOMX->getParameter(mNode, OMX_IndexParamPortDefinition, &portDefn, sizeof(portDefn));

portDefn.nBufferCountActual = iOutBufCnt;  // set suitable value or leave to default.
portDefn.nBufferSize = (vidWidth * vidHeight * 3) / 2;
portDefn.format.video.nFrameWidth = vidWidth;
portDefn.format.video.nFrameHeight = vidHeight;
portDefn.format.video.nStride = vidWidth;
portDefn.format.video.nSliceHeight = vidHeight;

mOMX->setParameter(mNode, OMX_IndexParamPortDefinition, &portDefn, sizeof(portDefn));
```

Note that getParameter() is called to initialize the portDefn structure with default values. Only the necessary fields of the structure is modified before setting it back.

3.2.6 Handling call backs from OMX component

OMX component uses three call back functions EmptyThisBuffer(), FillThisBuffer() & EventHandler() to notify buffer processing completion and events. IOMX clubbed the three call backs into a single one onMessage(). The client class (which wants to receive the call backs) should be derived from BnOMXObserver base class implement the virtual method onMessage().

```
class OmxH264Dec : public BnOMXObserver
{
  . . .
}
```

Three types of call back functionalities are implemented in three local functions and called based on message type in the following code snippet.

```
void OmxH264Dec::onMessage(const omx_message &msg)
{
   switch(msg.type)
   {
      case omx_message::EVENT:
      OnEvent(msg);
      break;

      case omx_message::EMPTY_BUFFER_DONE:
      OnEmptyBufferDone(msg);
      break;

      case omx_message::FILL_BUFFER_DONE:
      OnFillBufferDone(msg);
      break;
   }
}
```

3.2.7 Frame decoding

Encoded video frames are read from the file through the file parser module. Codec configuration data is also fetched through parser and sent to the decoder at the beginning. Both frame data and codec configuration data are passed through emptyBuffer() on the input port as shown here.

Free buffers for decoded data are pushed to the component through fillBuffer() as shown here. OMX decoder component fills the buffer and provides back to the client through call back.

3.2.8 Video rendering

Video renderer object is created by referencing the surface object received from Application through JNI. The displayed video can be scaled through `displayWidth` & `displayHeight and rotated through rotationDegrees` parameters.

```
sp<IOMXRenderer> mOMXRenderer = mOMX->createRenderer(surface, compName,
                                                     OMX_COLOR_FormatYUV420Planar,
                                                     vidWidth, vidHeight,
                                                     displayWidth,  displayHeight,
                                                     rotationDegrees);
```

Filled decoded video buffer can be rendered through hardware video renderer as shown below in the code snippet.

```
mOMXRenderer->render(mVideoBuffer);
```

3.3 State management

States of OMX component and transition between states is as per OMX standard. The client layer can mimic the states of the component or use a different set of states. This decision is solely dependent on client functionality and design. It is not advisable to mimic the OMX component states without having valid reason.

All state change notifications are given through call back from the OMX component.

3.4 Buffer management

OMX components use a number of buffers for both input and output ports. Typically a H264 component uses 2 or more buffers for input port and 4 or more buffers for output ports. It's the client's responsibility to track the usage of all the buffers. It is advisable to design a separate buffer handler class to distribute the complexities involved in buffer handling.

3.4.1 Registering a buffer to the handler

Registering a buffer to the buffer handler is done once per buffer during its allocation. This is shown earlier in this document.

3.4.2 Status of a buffer

A buffer can be used by multiple modules in the data path. Buffer manager should know who is using a particular buffer at a given point of time. Maintaining status of all buffers is absoultely necessary. If there are n modules which can use a buffer, then status of the buffer can be an enumeration with (n+ 1) values as shown in the example below.

BUFFER_STATUS_FREE,
BUFFER_STATUS_USED_BY_COMP1
BUFFER_STATUS_USED_BY_COMP2
.
.
.
BUFFER_STATUS_USED_BY_COMPn

3.4.3 Acquiring a buffer

Status of a buffer is BUFFER_STATUS_FREE when it's not being used by any module. When some module wants to use it, it should notify the buffer handler so that the status of the buffer is modified accordingly. This is called acquiring of a buffer.

OMX components maintain buffer queues on input and output ports. It is client's responsibility to keep the queues always full by calling emptyBuffer() for input port and fillBuffer() for output port to achieve best performance from the OMX component.

```
buffer_id in, out;
uint8_t *pData;
uint32_t flags = OMX_BUFFERFLAG_ENDOFFRAME;
uint32_t dataLen;

in = iBufHndlr->AccuireInBuf(&pData);

readEncFrame(pData, &dataLen);   // Fetch H264 encoded frame data here

if(mFirstFrame)  // First frame should contain only codec config data.
{
   flags |= OMX_BUFFERFLAG_CODECCONFIG;
```

```
   mFirstFrame = 0;
}

mOMX->emptyBuffer(mNode, in, 0, dataLen, flags, 0);

out = iBufHndlr->AccuireOutBuf();

mOMX->fillBuffer(mNode, out);
```

Note: Codec configuration data is SPS PPS information in case of H264. Some implementation can expect SPS & PPS together in the first frame. Some H264 decoder implementation expects SPS & PPS data to be sent through first two frames. It's advisable to go for the two frame approach to make the implementation more generic.

3.4.4 Releasing a buffer

Buffers are returned back through call backs after being used by OMX component. Client's call back handler functions should return the buffers to the buffer handler then.

```
void OmxH264Dec::OnEmptyBufferDone(const omx_message &msg)
{
   mBufHndlr->ReleaseInBuf(msg.u.buffer_data.buffer);
}
```

Releasing the output buffer can't be done from the corresponding call back though. Those have to be sent to video renderer for display. Those can be returned to the buffer handler once the display is over.

3.5 Error handling

All IOMX APIs returns error codes. Also there can be error event call back from OMX component. These errors have to be handled in the client.

4. References

- OMX IL standard

5. Authors

L&T Infotech Limited
Susanta Bhattacharjee - Senior Technical Architect
Venkateswaran Raghavan – Head Android Practice