Tech

Android 多媒体架构分析

Revision History

Date Issue		Description	Author
<2/05/2010> <0.5>			wylhistory

目录

1. Abstract

主要是分析一下android的多媒体架构，以及和集成相关的东西 .

2. Introduction

Android的多媒体架构及其的复杂，代码量也是非常的大，甚至我认为是所有模块里面最复杂的一块，因为里面包含了音频，视频，时钟，同步等等；

说实话，文档很长，我都很难有勇气看两遍，所以错误在所难免，请多包涵！

3. Android 多媒体架构

Android 的多媒体架构简单划分可以分层两部分， opencore以及 opencore之上 ,如图：

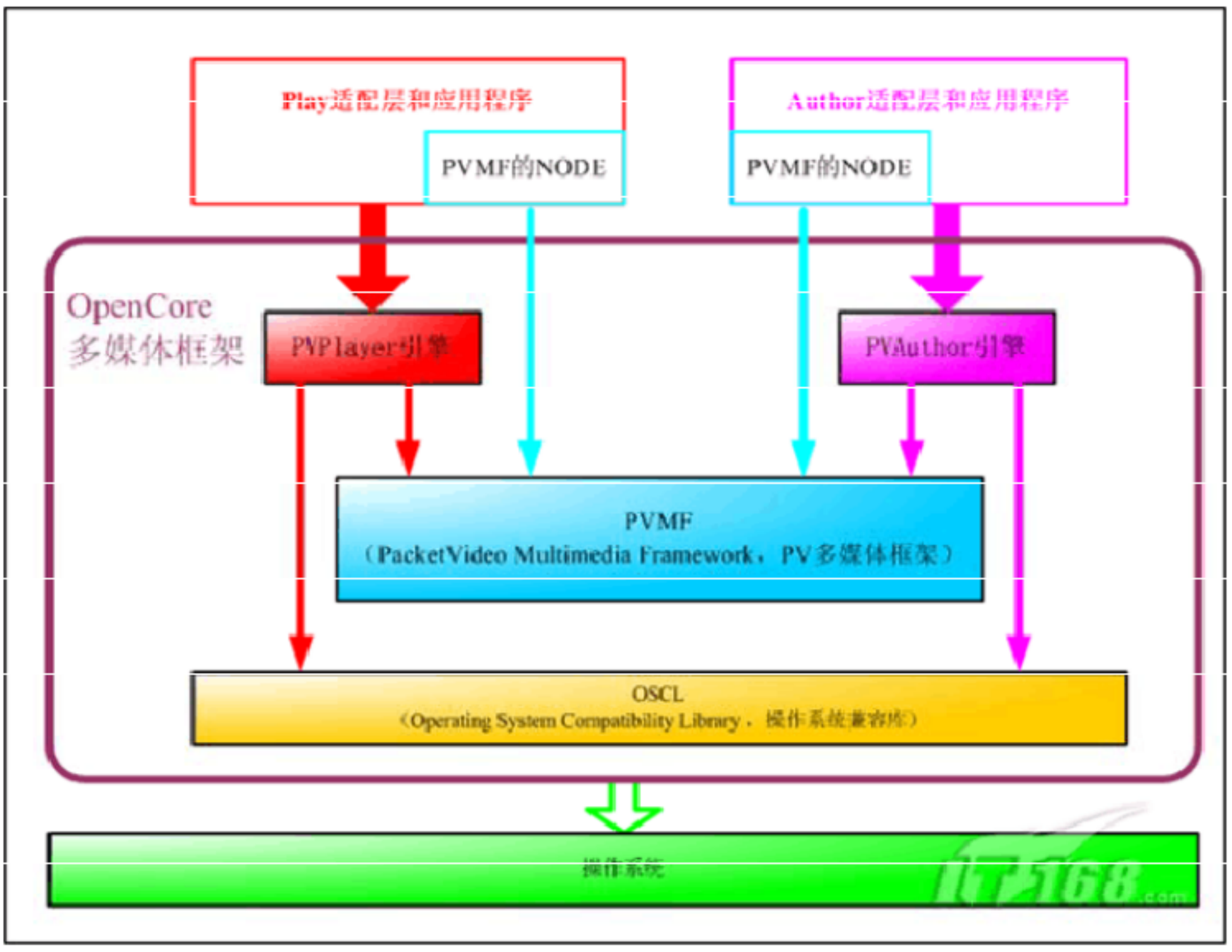


图3.0.0 android多媒体架构

Opencore本身的架构又分成如下三个部分：

AL ：application layer，这层，由 android自己实现；

IL:integration layer，这层， marvel自己实现了一套编解码 component；

DL:development layer;

如下图所示：

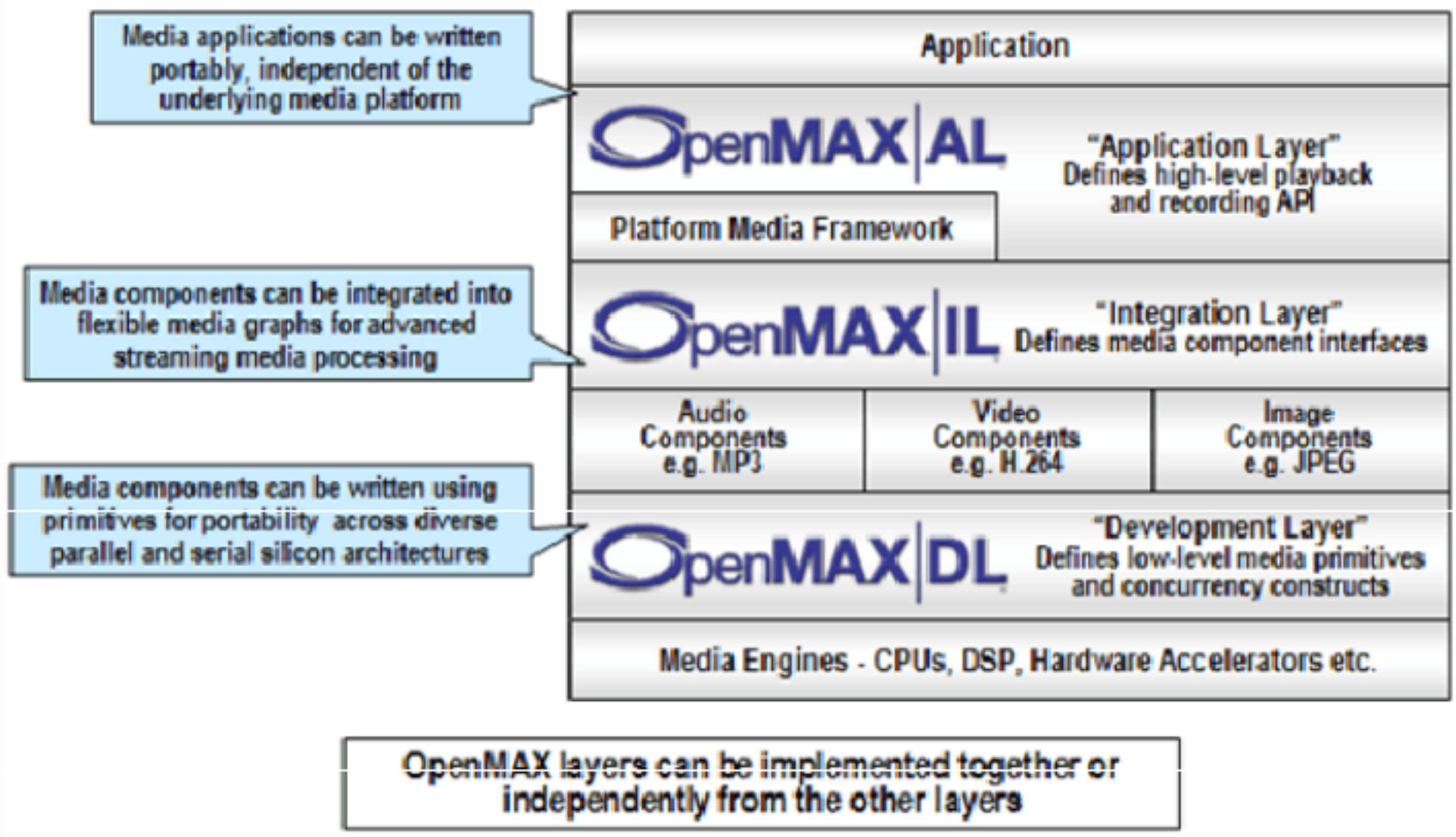


图3.0.1 openmax框架

总的来说，包括以下几个部分：

UI层：也就是用户操作的界面，比如播放暂停等，主要是 mediaplayer.java；

JNI层：也就是连接 java和c代码的地方 ,主要是 android_media_mediaplayer.cpp;

Mediaplayer客户端：也就通过 binder转发 jni的命令到服务器端的地方，主要在 mediaplayer.cpp里面；

服务器端： 也就是接受客户端请求的地方， 当然它依然只是一个管理的地方， 不是真正干活的地方；

Framework和openmax的适配层：这里是连接 android框架和 openmax引擎的地方；

Openmax引擎：这里是引擎层，负责管理各个 component的地方，并控制状态切换；

Component层：这里就是各个 component了；

DL 层：这里就是提供一些基本的操作原语的地方；

下面分别介绍一下这几层，包括它们之间的交互，当然需要提一下相关的代码，最后以一个实际的例子来分析。

3.1 代码位置

以开源的 Android为例MediaPlayer的代码主要在以下的目录中：

JAVA 程序的路径：

packages/apps/Music/src/com/android/music/

JAVA 本地调用部分（ JNI）：

frameworks/base/media/jni/android_media_MediaPlayer.cpp

这部分内容编译成为目标是 libmedia_jni.so。

主要的头文件在以下的目录中：
frameworks/base/include/media/
这个目录是和 libmedia.so库源文件的目录 frameworks/base/media/libmedia相对应的。主要的头文件有以下几个：
IMediaPlayerClient.h：定义了类 class IMediaPlayerClient 和类 BnMediaPlayerClient ，后者主要用来和服务通讯；
mediaplayer.h：定义了类 class MediaPlayer，继承于 BnMediaPlayerClient ，供 jni 层使用；
IMediaPlayer.h：这个是接口类，定义了 class IMediaPlayer: public IInterface 和 class BnMediaPlayer: public BnInterface<IMediaPlayer> ；
IMediaPlayerService.h ：定义了接口 class IMediaPlayerService: public IInterface ，和 class BnMediaPlayerService: public BnInterface<IMediaPlayerService> 。
MediaPlayerInterface.h：

多媒体底层库在以下的目录中：
frameworks/base/media/libmedia/
这部分的内容被编译成库 libmedia.so。

多媒体服务部分：
frameworks/base/media/libmediaplayerservice/
文件为 mediaplayerservice.h 和 mediaplayerservice.cpp
这部分内容被编译成库 libmediaplayerservice.so。

基于 OpenCore 的多媒体播放器部分
external/opencore/
这部分内容被编译成库 libopencoreplayer.so。

从程序规模上来看， libopencoreplayer.so 是主要的实现部分，而其他的库基本上都是在其上建立的封装和为建立进程间通讯的机制，当然复杂的部分是编解码器，我们一般看不到。

3.2 MediaPlayer

这是一个比较上层的类，是给更上层或者说 UI 程序使用的 java 类；它的存在就是为了完全独立底层的实现，比如它的一个典型的用法就是这样：

```
MediaPlayer mp=new MediaPlayer();
mp.setDataSource(PATH_TO_FILE);
mp.prepare();
mp.start();
```

它所交互的层就是 JNI 层，也就是说它真正做的事都是通过 JNI 来做的，这几句话做的事情后面会详细介绍，这里就是有一个直观的了解就行了。

3.3 jni 层

这里有一个表，如下：

```
static JNINativeMethod gMethods[] = {
    {"setDataSource",       "(Ljava/lang/String;)V",            (void
*) android_media_MediaPlayer_setDataSource   },
    {"setDataSource",       "(Ljava/io/FileDescriptor;JJ)V",    (void
*)android_media_MediaPlayer_setDataSourceFD},
    {"_setVideoSurface",    "()V",                              (void *)android_media_MediaPlayer_setVideoSurface},
```

```
    {"prepare",             "()V",                      (void *)    android_media_MediaPlayer_prepare   },
    {"prepareAsync",        "()V",                      (void *)android_media_MediaPlayer_prepareAsync},
    {"_start",              "()V",                     (void *)    android_media_MediaPlayer_start   },
    {"_stop",               "()V",                     (void *)android_media_MediaPlayer_stop},
    {"getVideoWidth",       "()I",                      (void *)android_media_MediaPlayer_getVideoWidth},
    {"getVideoHeight",      "()I",                      (void *)android_media_MediaPlayer_getVideoHeight},
    {"seekTo",              "(I)V",                     (void *)android_media_MediaPlayer_seekTo},
    {"_pause",              "()V",                      (void *)android_media_MediaPlayer_pause},
    {"isPlaying",           "()Z",                     (void *)android_media_MediaPlayer_isPlaying},
    {"getCurrentPosition",  "()I",                      (void *)android_media_MediaPlayer_getCurrentPosition},
    {"getDuration",         "()I",                     (void *)android_media_MediaPlayer_getDuration},
    {"_release",            "()V",                      (void *)android_media_MediaPlayer_release},
    {"_reset",              "()V",                     (void *)android_media_MediaPlayer_reset},
    {"setAudioStreamType",  "(I)V",                          (void
*)android_media_MediaPlayer_setAudioStreamType},
    {"setLooping",          "(Z)V",                     (void *)android_media_MediaPlayer_setLooping},
    {"isLooping",           "()Z",                      (void *)android_media_MediaPlayer_isLooping},
    {"setVolume",           "(FF)V",                        (void *)android_media_MediaPlayer_setVolume},
    {"getFrameAt",          "(I)Landroid/graphics/Bitmap;",     (void
*)android_media_MediaPlayer_getFrameAt},
    {"native_invoke",       "(Landroid/os/Parcel;Landroid/os/Parcel;)I",(void
*)android_media_MediaPlayer_invoke},
    {"native_setMetadataFilter", "(Landroid/os/Parcel;)I",      (void
*)android_media_MediaPlayer_setMetadataFilter},
    {"native_getMetadata", "(ZZLandroid/os/Parcel;)Z",          (void
*)android_media_MediaPlayer_getMetadata},
    {"native_init",         "()V",                      (void *)android_media_MediaPlayer_native_init},
    {"native_setup",        "(Ljava/lang/Object;)V",            (void *)android_media_MediaPlayer_native_setup},
    {"native_finalize",     "()V",                       (void *)android_media_MediaPlayer_native_finalize},
    {"snoop",               "([SI)I",                   (void *)android_media_MediaPlayer_snoop},
};
```

可以看见，这里面有对先前调用的 setDataSource,prepare,start 的本地实现，比如：

```
static void
android_media_MediaPlayer_prepare(JNIEnv *env, jobject thiz)
{
    sp<MediaPlayer>mp = getMediaPlayer  (env, thiz);
    if (mp == NULL ) {
        jniThrowException(env, "java/lang/IllegalStateException", NULL);
        return;
    }
    setVideoSurface(mp, env, thiz);
    process_media_player_call ( env, thiz, mp->prepare() , "java/io/IOException", "Prepare failed." );
}
```

这里也只是跑马观花的走一圈，大约就是先取得已经在 new的时候建立的 MediaPlayer，然后 mp->setVideoSurface，然后开始调用 mp->prepare ，prepare里面做的工作是最多的，非常的繁琐，后面会讲，从应用程序的角度来说， prepare以后，就可以通过 start来启动了；

3.4 MediaPlayer 客户端

如果理解 binder的机制， 这个就很好理解了， 这个就是 MediaplayerService在客户端的代理了， 它的创建如下所示：

```
const sp<IMediaPlayerService>& MediaPlayer::getMediaPlayerService()
{
```

```
Mutex::Autolock _l(sServiceLock);
if (sMediaPlayerService.get() == 0) {
    sp<IServiceManager> sm = defaultServiceManager();
    sp<IBinder> binder;
    do {
        binder = sm->getService(String16("media.player"));
        if (binder != 0)
            break;
        LOGW("MediaPlayerService not published, waiting...");
        usleep(500000); // 0.5 s
    } while(true);
    if (sDeathNotifier == NULL) {
        sDeathNotifier = new DeathNotifier();
    }
    binder->linkToDeath(sDeathNotifier);
    sMediaPlayerService = interface_cast<IMediaPlayerService>(binder);
    }
    LOGE_IF(sMediaPlayerService==0, "no MediaPlayerService!?");
    return sMediaPlayerService;
}
```

通过 servicemanager 利用 getService 就可以返回一个 BpMediaPlayerS ervice,然后通过它的 create 函数也就是 service->create(getpid(), this, url) 就创建了一个 BpMediaPlayer，这两个结构本质上就是 BnMediaPlayerService 和 BnMediaPlayer 的代理，通过它就可以访问后面这两个服务器端的方法了，比如：

mp->prepare()

最后就会调用 mPlayer->prepareAsync()

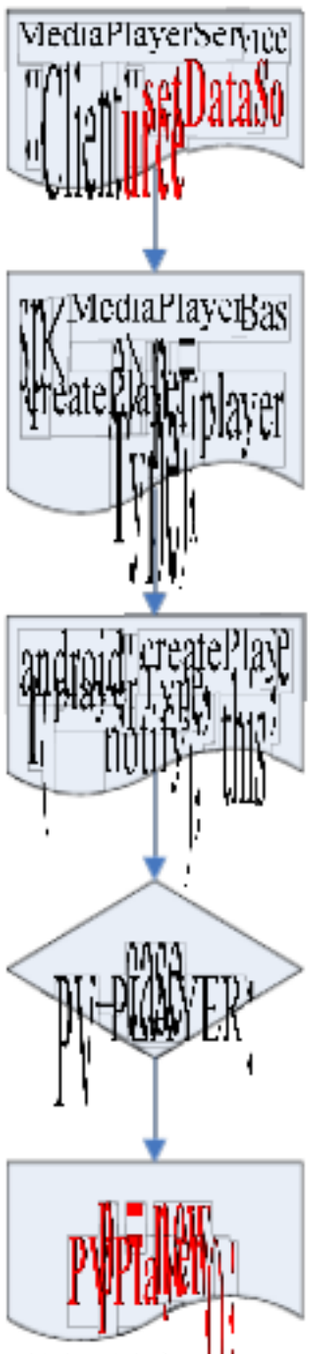
它最后会调用到 BnMediaPlayer里面的 prepareAsync了；

3.5 BnMediaPlayer

mPlayer->prepareAsync的实现如下（注意这里的 Client原型为：class Client : public BnMediaPlayer ）：

```
status_t MediaPlayerService::Client::prepareAsync()
{
    LOGV("[%d] prepareAsync", mConnId);
    sp<MediaPlayerBase> p = getPlayer();
    if (p == 0) return UNKNOWN_ERROR;
    status_t ret =p->prepareAsync();
#if CALLBACK_ANTAGONIZER
    LOGD("start Antagonizer");
    if (ret == NO_ERROR) mAntagonizer->start();
#endif
    return ret;
}
```

可以看到中里面的 prepareAsync还是通过一个 p->prepareAsync 来实现的，而 p是来自于哪里呢？如下：



可见 p是在 setDataSource的时候，通过 new创建出来的一个 PVPlayer 实例；
这样控制就到了 PVPlayer层了，继续往下看：

3.6 PVPlayer 层

我们还是先不看它的初始化等了，因为内容是在是太多了，还是看看上面的 prepareAsync

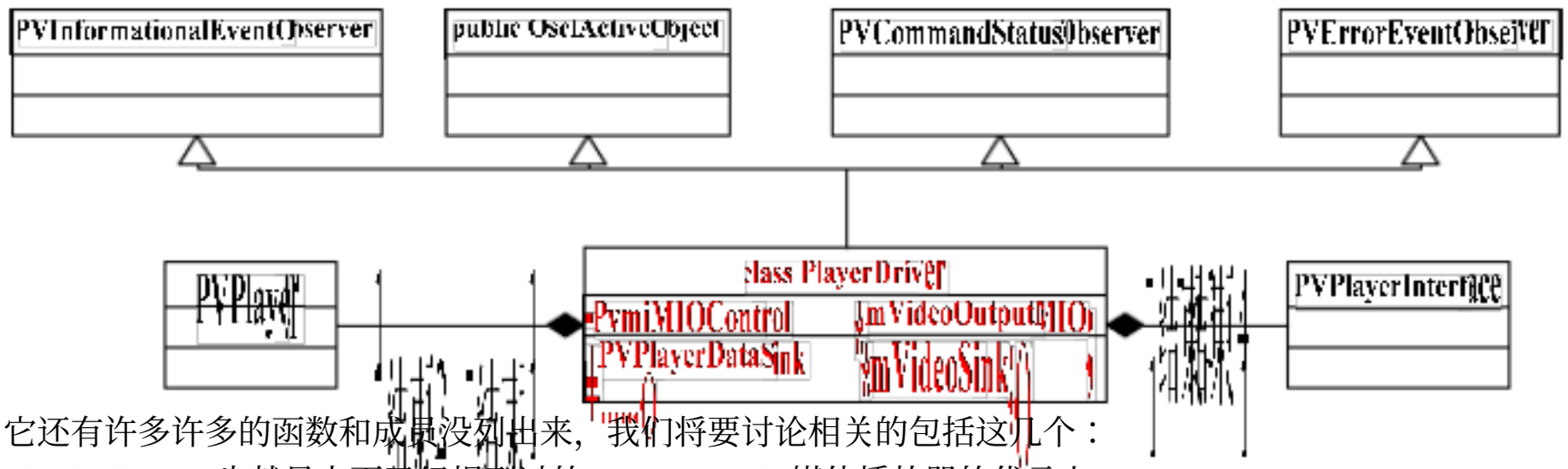
```
status_t PVPlayer::prepareAsync()
{
    LOGV("prepareAsync");
    status_t ret = OK;

    if (!mIsDataSourceSet) {  // If data source has NOT been set.
        // Set our data source as cached in setDataSource() above.
        LOGV("  data source = %s", mDataSourcePath);
        ret = mPlayerDriver->enqueueCommand   (new
PlayerSetDataSource (mDataSourcePath,run_init  ,this));
        mIsDataSourceSet = true;
    } else {  // If data source has been already set.
        // No need to run a sequence of commands.
        // The only code needed to run is PLAYER_PREPARE.
        ret = mPlayerDriver->enqueueCommand(new PlayerPrepare(do_nothing, NULL));
    }

    return ret;
}
```

这个 PVPlayer::prepareAsync基本上没干啥有用的， 只是把一条命令加入到了一个队列， 设置标志位，就返回了，这也就是为什么名字上有 asynd的原因了；
OK ， 前面的所有的逻辑基本上都是二传手，下面着重分析的是 PVPlayer 以及下面几层 ；

3.7 PlayerDriver

PVPlayer加这层基本上可以看做是 android的多媒体架构和 opencore之间的桥梁；

它的类结构如下：



它还有许多许多的函数和成员没列出来，我们将要讨论相关的包括这几个：

A) PVPlayer 也就是上面已经提到过的， Android 媒体播放器的代言人；
B) 而 PVPlayerInterface 则是 opencore 的引擎层的代理人；
C) 红色的两个成员是和我们将要讨论的视频的输出有关的，后面会讲到；
D) 至于其它的它的父类，为了不打断我们的思路，就留到后面讲，简单的说：
PVInformationalEventObserver 的作用就是给引擎层访问的 PlayerDriver 提供接口，用来处理引擎层或以下上传的事件；而 OsclActiveObject 简单说就是一个执行上下文，它类似一个 Timer，到期就会执行它的 Run 函数，如果没人触发它将一直不执行，比如 PVPlayer 发送到队列里面的处理就是在 Run 函数里面处理的；而 PVCommandStatusObserver 也和 PVInformationalEventObserver 类似，是由引擎层来调用的接口， 以方便 PlayerDriver 传递的命令的执行状态的跟踪； PVErrorEventObserver 类似；

PLayerDriver的逻辑架构很简单，就是将 PVPlayer的命令放入队列，然后在 run函数里面一个个取出来处理，凡是以 ”handle开头的函数都是实际的处理函数，而不带 handle的都是加入到队列的函数，而带 handle的函数的处理一般都是这样的，比如先前的 prepareAsync命令的处理函数：

```
void PlayerDriver::handlePrepare(PlayerPrepare* command)
{
    //Keep alive is sent during the play to prevent the firewall from closing ports while
    //streaming long clip
    PvmiKvp iKVPSetAsync;
    OSCL_StackString<64> iKeyStringSetAsync;
    PvmiKvp *iErrorKVP = NULL;
    int error=0;
    iKeyStringSetAsync=_STRLIT_CHAR("x-pvmf/net/keep-alive-during-play;valtype=bool");
    iKVPSetAsync.key=iKeyStringSetAsync.get_str();
    iKVPSetAsync.value.bool_value=true;
    iErrorKVP=NULL;
    OSCL_TRY(error, mPlayerCapConfig->setParametersSync (NULL, &iKVPSetAsync, 1,
iErrorKVP));
    OSCL_TRY(error, mPlayer->Prepare(command) );
    OSCL_FIRST_CATCH_ANY(error, commandFailed(command));

    char value[PROPERTY_VALUE_MAX] = {"0"};
```

```
    property_get("ro.com.android.disable_rtsp_nat", value, "0");
    LOGV("disable natpkt - %s",value);
    if (1 == atoi(value))
    {
        //disable firewall packet
        iKeyStringSetAsync=_STRLIT_CHAR("x-pvmf/net/disable-firewall-packets;valtype=bool");
        iKVPSetAsync.key=iKeyStringSetAsync.get_str();
        iKVPSetAsync.value.bool_value = 1; //1 - disable
        iErrorKVP=NULL;
        OSCL_TRY(error,mPlayerCapConfig->setParametersSync(NULL,&iKVPSetAsync,1,iErrorKVP));
    }
}
```

看到了吗，实际的处理是通过 mPlayerCapConfig->setParametersSync 和mPlayer->Prepare 来完成的，而实际上对于 opencore这个mPlayerCapConfig 和mPlayer 是同一个指针；

3.8 引擎层 PVPlayerEngine

对于命令的实际处理， 比如prepare等，最后都落在了 PVPlayerEngine这里了，这里是 opencore的核心，它是基于状态机的， 逻辑处理有点像 PlayerDriver，也是先把命令放到队列里面， 然后在 run函数里面一个个取出来执行的，不过非常繁琐，就这一个文件就有 一万七千行 ！！！还是先看看它的组成结构吧：

```
class PVPlayerEngine
        : public OsclTimerObject
        , public PVPlayerInterface
        , public PvmiCapabilityAndConfigBase
        , public PVMFNodeCmdStatusObserver
        , public PVMFNodeInfoEventObserver
        , public PVMFNodeErrorEventObserver
        , public PVPlayerDatapathObserver
        , public OsclTimerObserver
        , public PVPlayerLicenseAcquisitionInterface
        , public PVPlayerRecognizerRegistryObserver
        , public PVPlayerWatchdogTimerObserver
        , public PVPlayerTrackSelectionInterface
        , public PVMFMediaClockNotificationsObs
        , public ThreadSafeQueueObserver
        , public PVMFCPMStatusObserver
```

估计看到这一堆东西，你就再没有胃口看下去了，我也是；

或许另外一个简单一点的图可以让你喘口气：

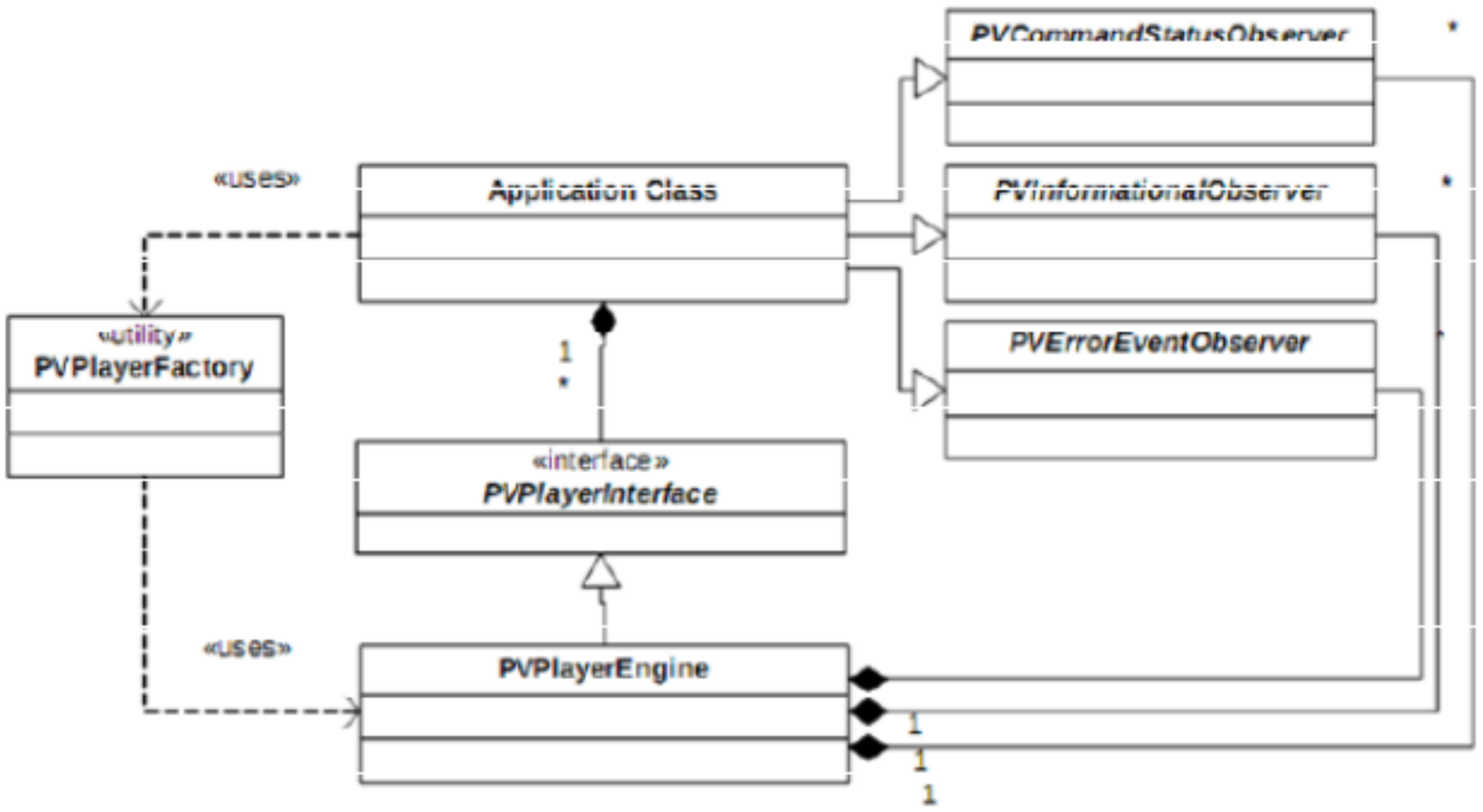


Figure 2: Class Diagram

它的创建流程如下：

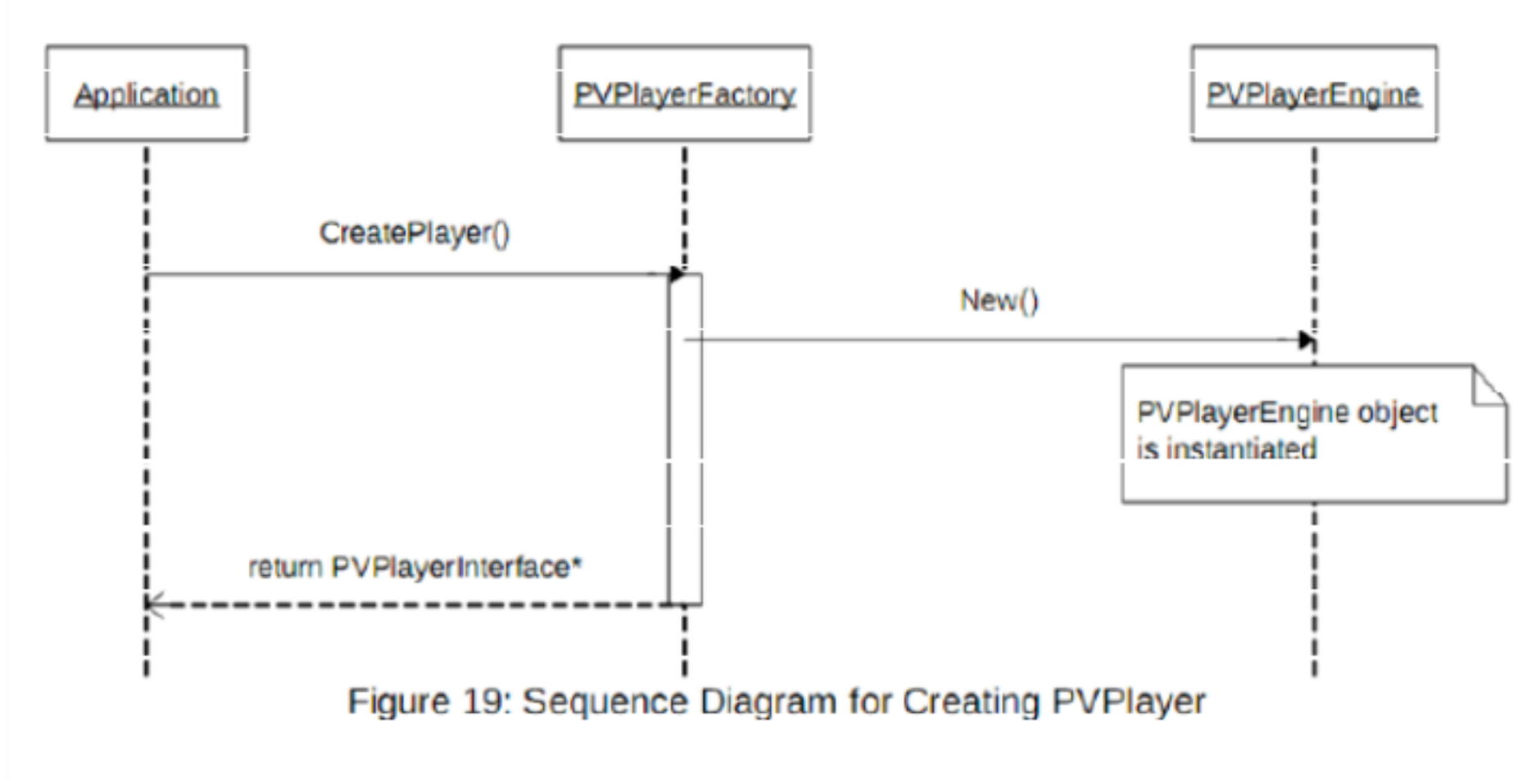


Figure 19: Sequence Diagram for Creating PVPlayer

我最多能够简单介绍一下： 它的特点就是凡是以 ”Do”开始的就是实际的处理命令的地方， 而不带 Do 的就只是把命令加入到队列的地方，比如：

```
PVCommandId PVPlayerEngine::Prepare(const OsclAny* aContextData)
{
   PVLOGGER_LOGMSG(PVLOGMSG_INST_LLDBG, iLogger, PVLOGMSG_STACK_TRACE, (0,
"PVPlayerEngine::Prepare()"));
   return AddCommandToQueue  (PVP_ENGINE_COMMAND_PREPARE     ,
(OsclAny*)aContextData);
}
```

这个不带“ Do”的就只是通过 AddCommandToQueue加入到队列，而

```
PVMFStatus PVPlayerEngine::DoPrepare(PVPlayerEngineCommand& aCmd)
```

{
…
}
就是实际的处理的地方，这个函数很大，实际上绝大多数工作都是在这里做或者说发起的。
其中 Oscl_Vector<PVPlayerEngineCommand, OsclMemAllocator> iCurrentCmd ;是描述当前正在处理的命令，而 OsclPriorityQueue<PVPlayerEngineCommand, OsclMemAllocator, Oscl_Vector<PVPlayerEngineCommand, OsclMemAllocator>, PVPlayerEngineCommandCompareLess> iPendingCmds ; // Vector to hold the command that has been requested是描述放入到队列里面的命令，
引擎的核心就是 run函数，在那里会根据状态以及命令的类型一个个取出来然后处理；
也许可以看看它的状态转换图，就大约明白它都干了些什么：

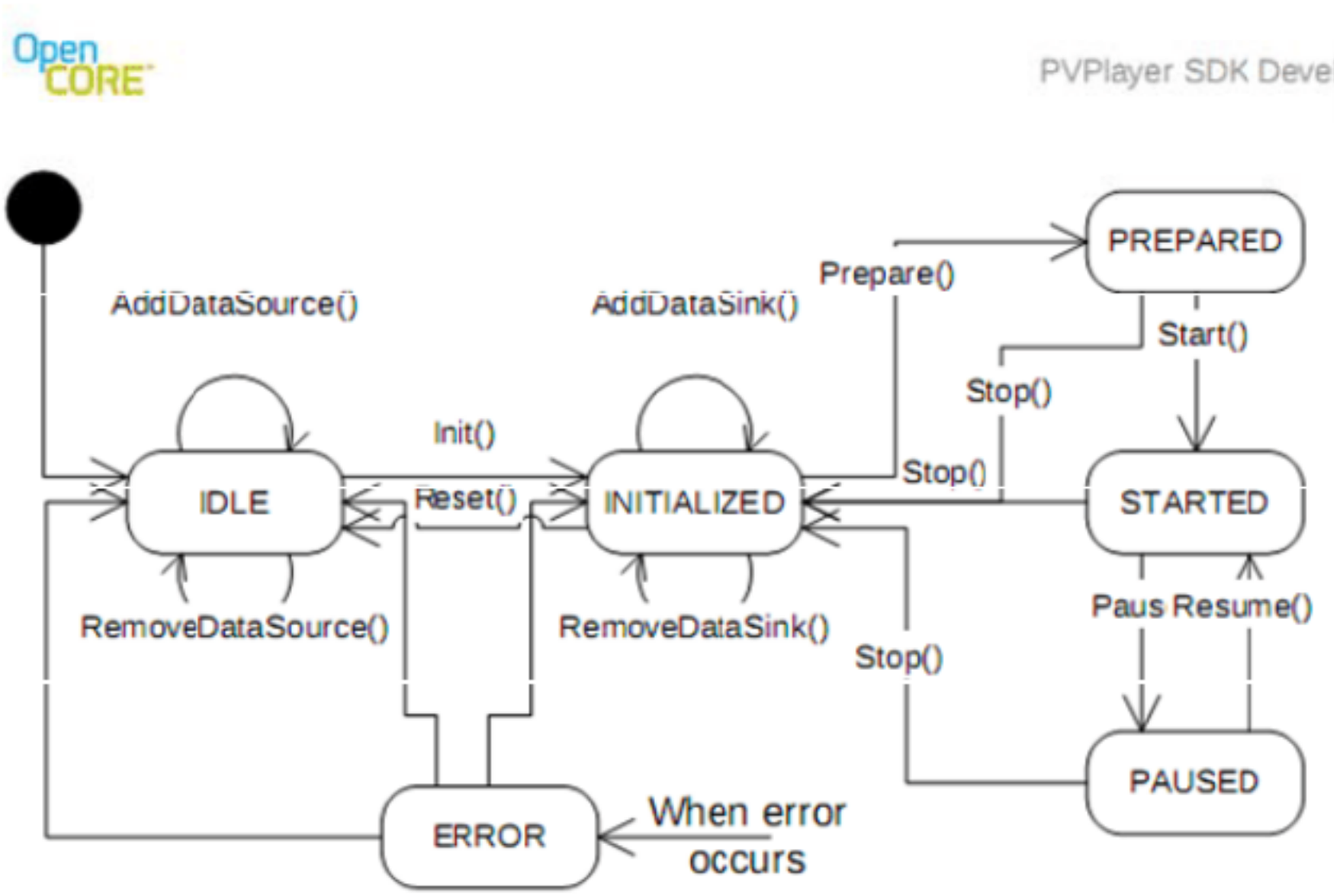
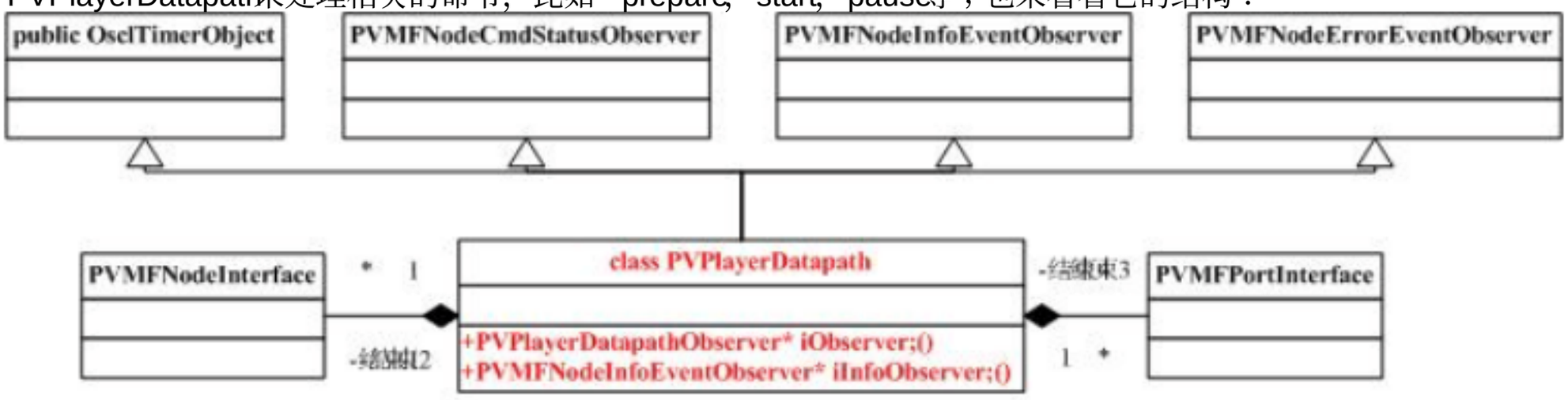


Figure 3: State Transition Diagram

画得很清楚了， 也就是正常情况下从 IDLE ——>INITIALIZED ——>PREPARED——>STARTED，当然实际的情况会很复杂，比如对于 prepared 里面又细分为好几个状态，后面会讲到；

3.9 PVPlayerDatapath 层

什么时候会走到这里？
在引擎层的命令处理中，有一部分就是构造这个 PVPlayerDatapath，而另外一部分就是调用这个 PVPlayerDatapath来处理相关的命令，比如 prepare，start，pause等；也来看看它的结构：

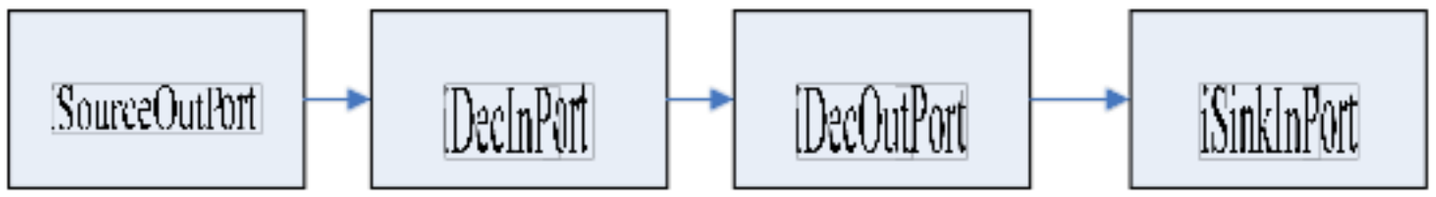


这里面需要说明的是 PVMFNodeInterface ,它其实就是存放的实际的 node，比如 iSourceNode，

iDecNode，iSinkNode，而 PVMFPortInterface 就是存放的这些 Node所拥有的这些 Port，而这些 port 才是实际负责传递数据的对象，比如：

PVMFPortInterface* iSourceOutPort;
PVMFPortInterface* iDecInPort;
PVMFPortInterface* iDecOutPort;
PVMFPortInterface* iSinkInPort;

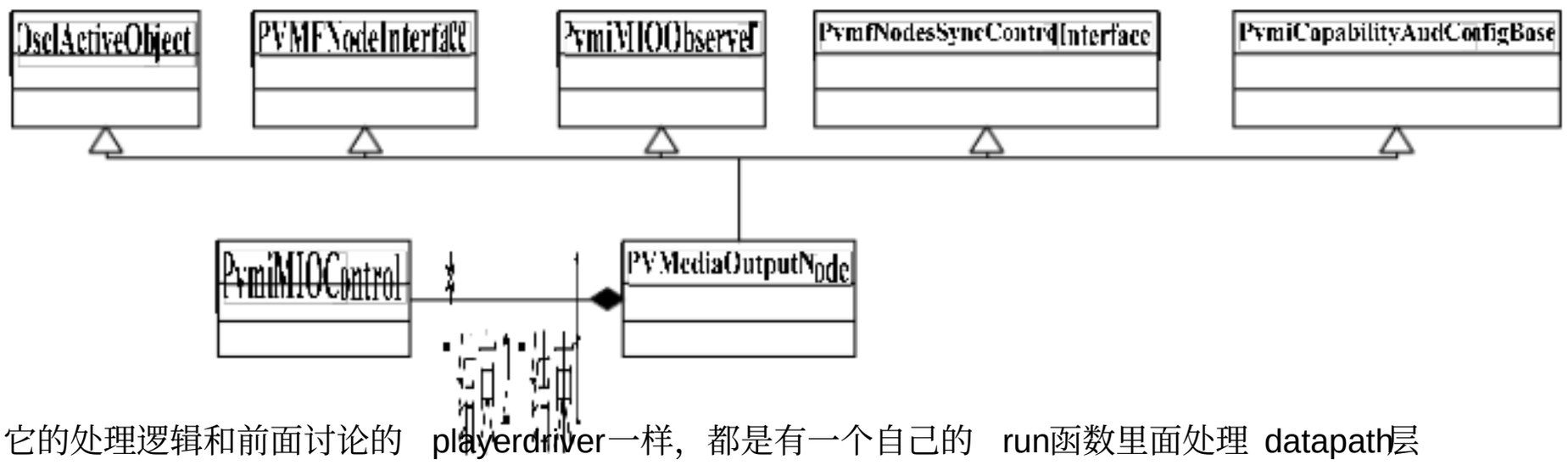
它们的连接方式像这样：



而ISourceOutPort就有可能是来自于一个文件构成的 node，中间两个 port是由一个解码器的 node来提供的，最后是一个 sink port它就有可能是视频输出的地方了；

3.10 节点层

这里讨论的是 PVMediaOutputNode，因为我们比较关注的是它和 MIO 的集成，先看看它的组成：



它的处理逻辑和前面讨论的 playerdriver一样，都是有一个自己的 run函数里面处理 datapath层（PVPlayerDatapath ）传下来的命令，比如：

```
OSCL_EXPORT_REF PVMFCommandId PVMediaOutputNode::Prepare (PVMFSessionId s, const
OsclAny* aContext)
{
   PVLOGGER_LOGMSG(PVLOGMSG_INST_LLDBG, iLogger, PVLOGMSG_STACK_TRACE,
                   (0, "PVMediaOutputNode::Prepare() called"));
   PVMediaOutputNodeCmd cmd;
   cmd.PVMediaOutputNodeCmdBase:Construct (s, PVMF_GENERIC_NODE_PREPARE , aContext);
   return QueueCommand L(cmd);
}
```

这个就是只放到队列里面，真正的处理都是在以 ”Do ”开头的函数里面，比如：

```
PVMFStatus PVMediaOutputNode:: Do Prepare (PVMediaOutputNodeCmd& aCmd)
{
   PVLOGGER_LOGMSG(PVLOGMSG_INST_LLDBG, iLogger, PVLOGMSG_STACK_TRACE,
                   (0, "PVMediaOutputNode::DoPrepare"));

   if (iInterfaceState != EPVMFNodeInitialized)
      return PVMFErrInvalidState;

   return SendMioRequest (aCmd, EInit );

}
```

可以看到它的实现又通过 SendMioRequest 由iMIOControl 来完成具体的工作：

```
PVMFStatus PVMediaOutputNode::SendMioRequest(PVMediaOutputNodeCmd& aCmd, EMioRequest
aRequest)
{
```

```
…
case EInit:
        {
            PvmiMediaTransfer* mediaTransfer = NULL;
            if (iInPortVector.size() > 0)
            {
                mediaTransfer =iInPortVector[0]->getMediaTransfer();
            }
            if (mediaTransfer != NULL)
            {
                OSCL_TRY(err, iMediaIOCmdId =  iMIOControl->Init();     );
            }
            if ((err != OsclErrNone))
            {
                PVLOGGER_LOGMSG(PVLOGMSG_INST_LLDBG, iLogger, PVLOGMSG_ERR,
                                (0, "PVMediaOutputNode::SendMioRequest: Error - iMIOControl->Init failed"));
                aCmd.iEventCode = PVMFMoutNodeErr_MediaIOInit;
                status = PVMFFailure;
            }
        }
        break;
…
}
```

总之这个输出节点的处理又是通过这个 iMIOControl 的init 来完成具体的准备工作；
它的整体工作逻辑，也就是数据传递的流程如下（后面会详细介绍）：

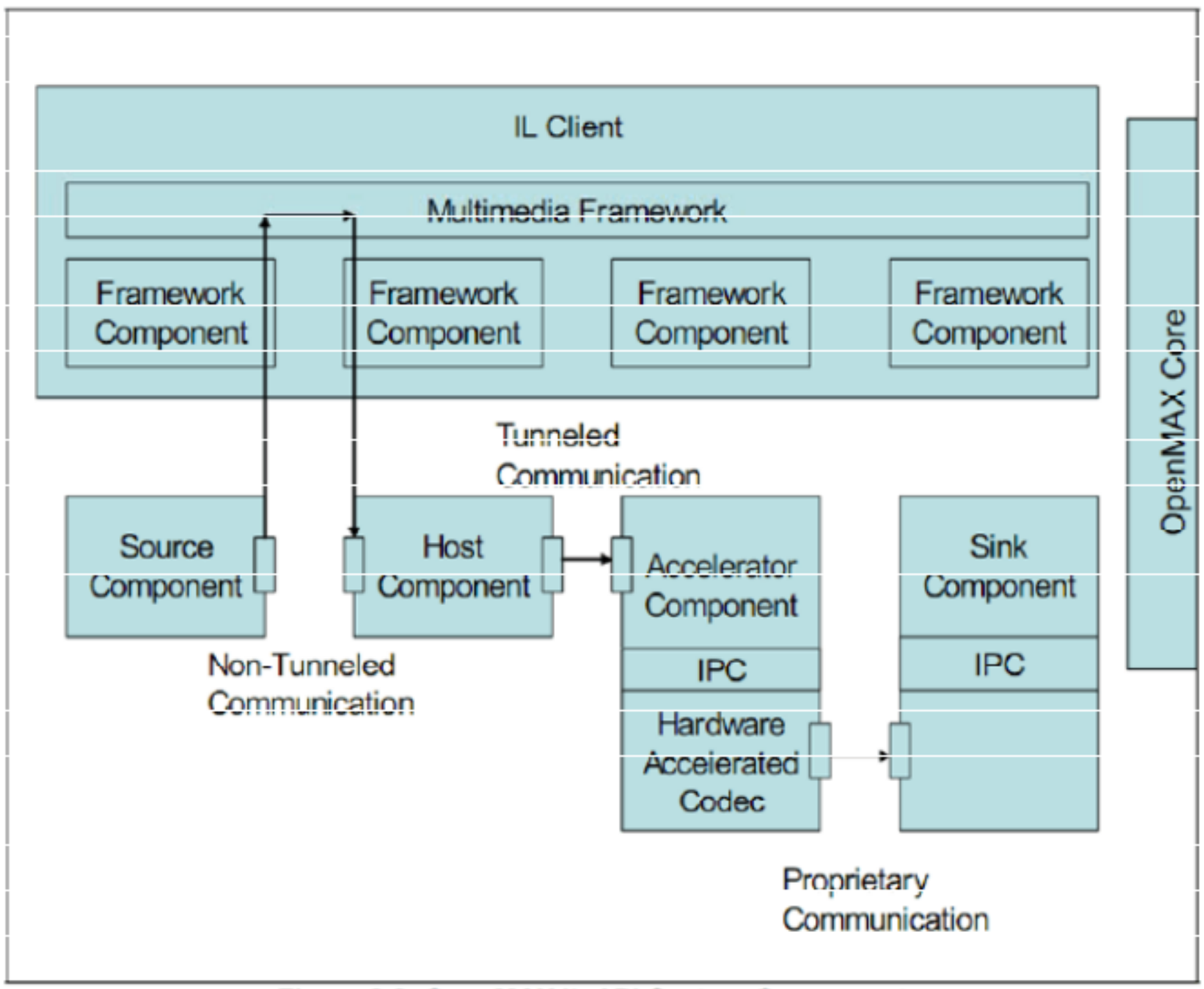
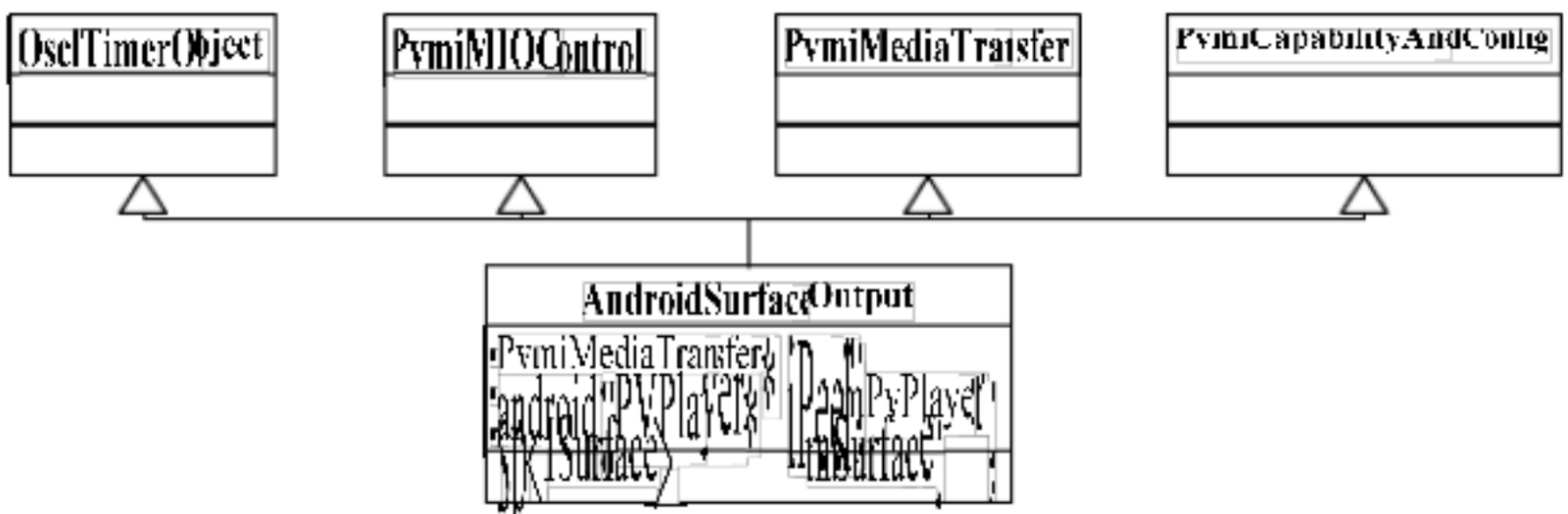


Figure 2-2. OpenMAX IL API System Components

上面的什么“ Framework Component ”也就是我说的 Node了，而下面的那些 ”Source Component”等就是我说说的解码的或者源的 component了，而对于所谓的 tunnel模式， Non-tunnel模式也留到后面再说吧；

3.11 MIO 层

按照marvel的处理逻辑，最后的数据输出都是经过 MIO 出来的，所以需要看看它的结构：



android默认实现了 PvmiMIOControl,PvmiMediaTransfer 和PvmiCapabilityAndConfig 以及 OsclTImerObject 接口，这样就可以在一个类里面做配置传输等工作了；其中的 iPeer也就是和它传递数据的对端， 而mSurface也就是代表物理的层， 当然，对于marvel的实现， 这个成员仅仅是用来判断是否在顶层，而数据的传递是直接通过打开 fb2来发送的，后面会讲；
现在从整体上看看它和上层应用的交互：

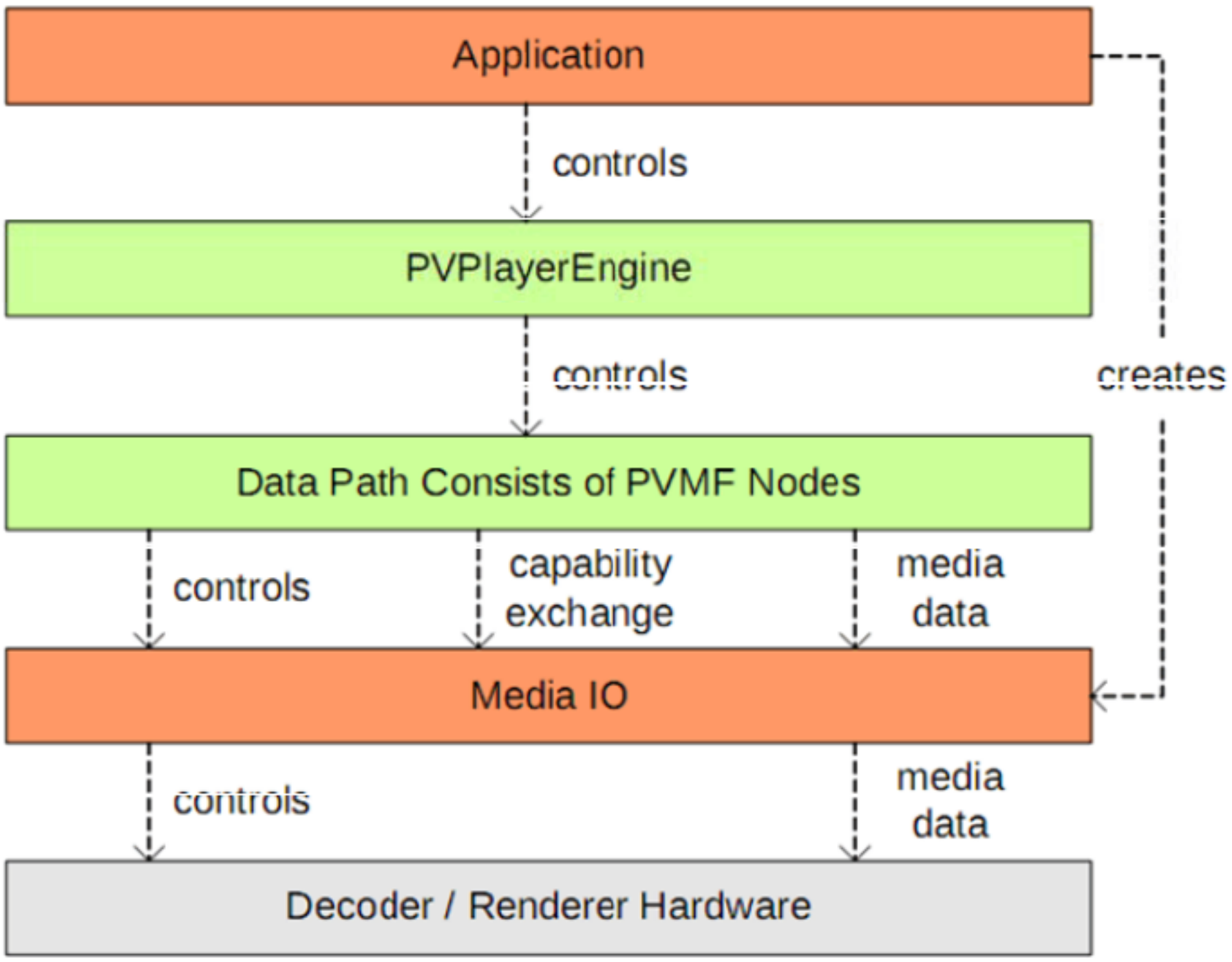


Figure 4: A diagram of the interaction between MIO components and player components.

基本上这些逻辑前面都已经有提到了，至于具体的函数流程图后面再讲，这里先看一个数据传输起始图：

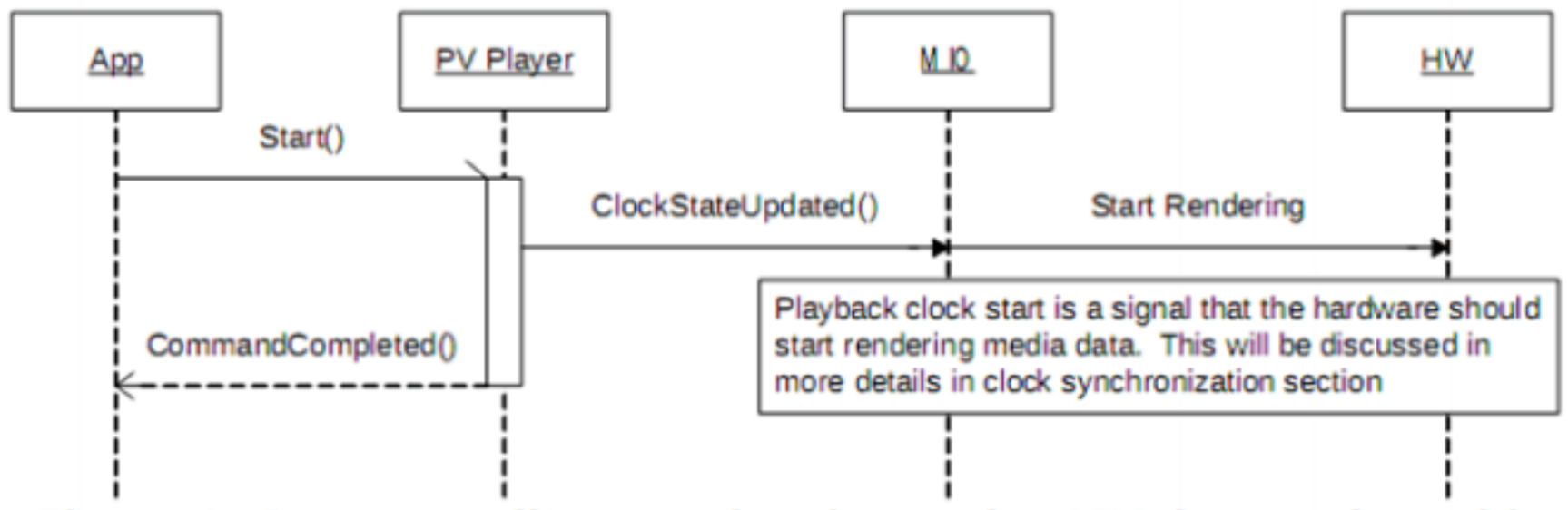
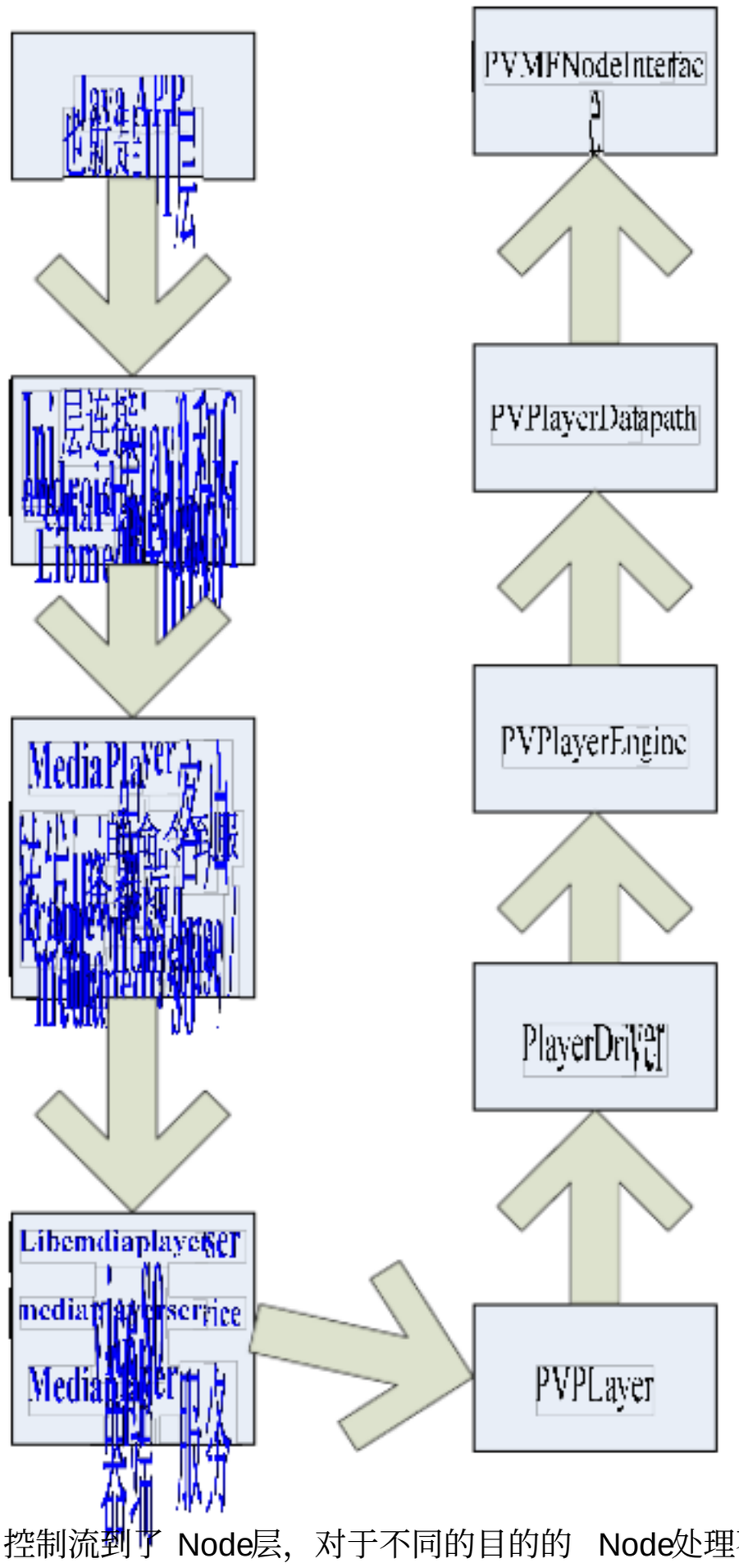


Figure 9: Sequence diagram showing active MIO interaction with clock at start of playback.

3.12 控制逻辑小结

到这里我们先整理一下控制逻辑的传递过程：



控制流到了 Node层，对于不同的目的的 Node处理不一样，比如上面已经讨论的 output 的node它就需要对 MIO 做初始化等，而对于 decode的node，就需要对里面所拥有的 decode的component进行初始化等；

4. 例子分析

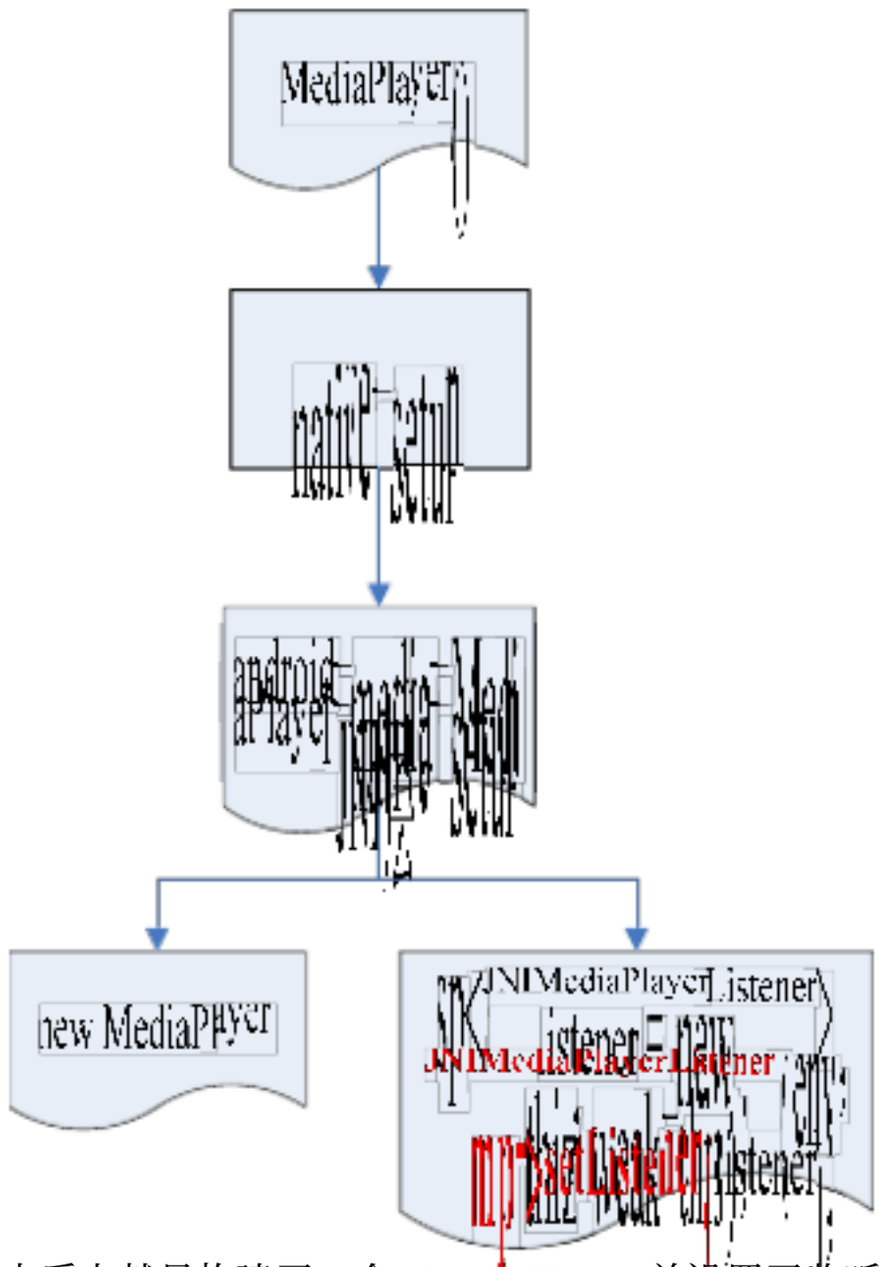
我们回到先前的例子：

```
MediaPlayer mp=new MediaPlayer();
mp.setDataSource(PATH_TO_FILE);
mp.prepare();
mp.start();
```

我们来一步步看看，里面是怎么实现的！

4.1 New MediaPlayer 的流程

如下：



本质上就是构建了一个 MediaPlayer,并设置了监听者，用于通知上层关于 mp的事件；

注意这时候并没有和 MediaPlayerService端建立连接，也就是说这时候仅仅是 MediaPlayer还不是IMediaPlayer*;

4.2 setDataSource 逻辑

它主要完成的功能是：

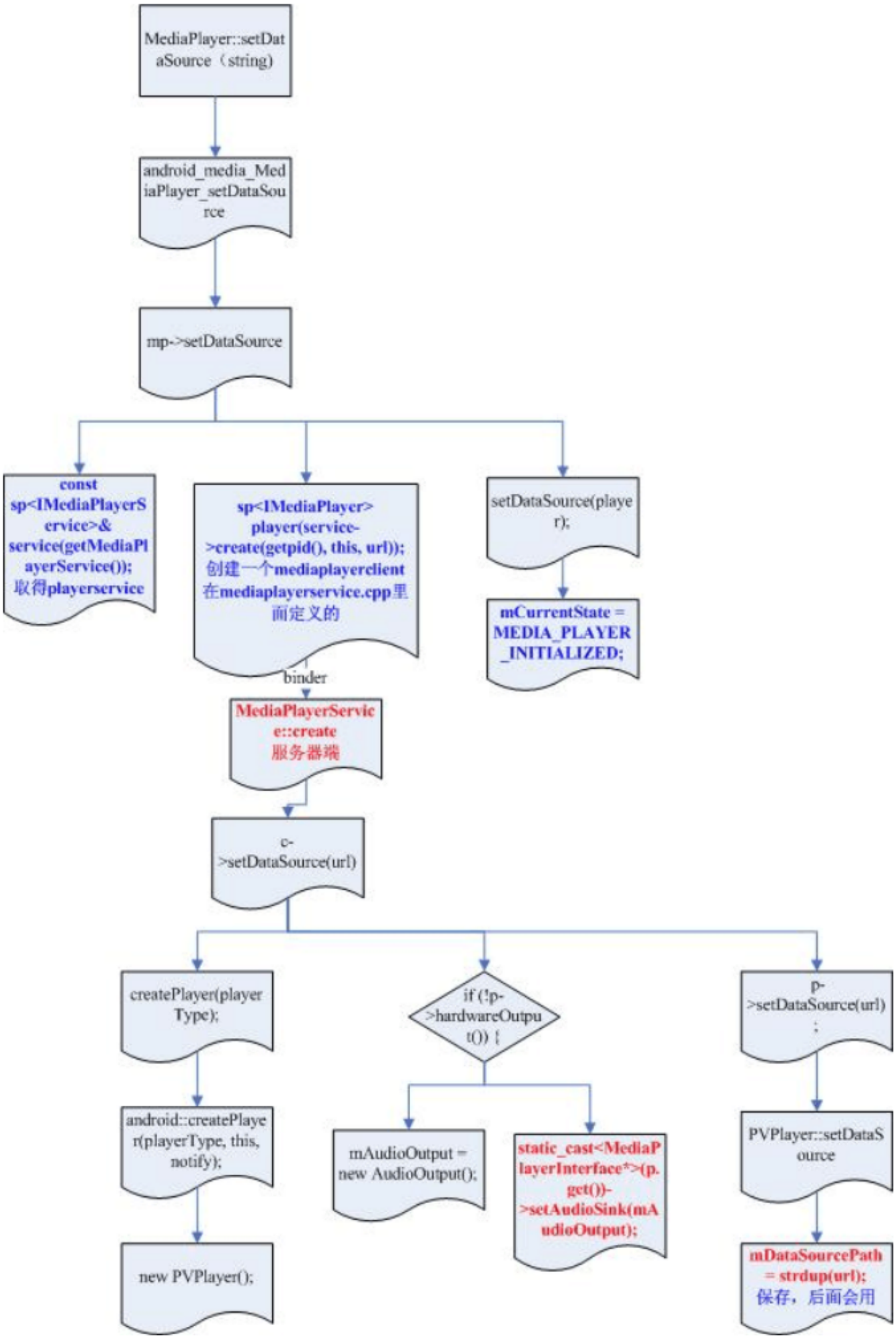
A） 客户端的 mPlayer的创建；

B） 对于服务器端的 mPlayer （=new PVPlayer（）） 的创建 ；

C） mAudioOutput = new AudioOutput(); 这个用于 audio输出，不过我不打算讨论这块。

D） mDataSourcePath = strdup(url);这里保存源地址；

逻辑图如下：


MediaPlayer::setDataSource（string)
android_media_MediaPlayer_setDataSource
mp->setDataSource
const sp<IMediaPlayerService>& service(getMediaPlayerService());
取得playerservice
sp<IMediaPlayer> player(service->create(getpid(), this, url));
创建一个mediaplayerclient
在mediaplayerservice.cpp里面定义的
setDataSource(player);
mCurrentState = MEDIA_PLAYER_INITIALIZED;
binder
MediaPlayerService::create
服务器端
c->setDataSource(url)
createPlayer(playerType);
android::createPlayer(playerType, this, notify);
new PVPlayer();
if (!p->hardwareOutput()) {
mAudioOutput = new AudioOutput();
static_cast<MediaPlayerInterface*>(p.get())->setAudioSink(mAudioOutput);
p->setDataSource(url);
PVPlayer::setDataSource
mDataSourcePath = strdup(url);
保存，后面会用

这里的真正工作基本上都是在服务器端实现的，主要干了一下几件事：

A） 创建PVPlayer, 它的作用前面已经说了，是连接 Android 的多媒体架构和 pv架构的桥梁；

B） 创建音频输出层，将来音频输出将从这里出来；

C） 保存源地址，供 prepare 的时候真正使用；

D） 更新状态为 INITIALIZED;

4.3 引擎层 prepare 前的流程

这个逻辑及其复杂，几乎所有的为执行 prepare的准备工作都是在这里做的，需要分成

A） PVPlayer及以上；

B） PlayerDriver的处理；

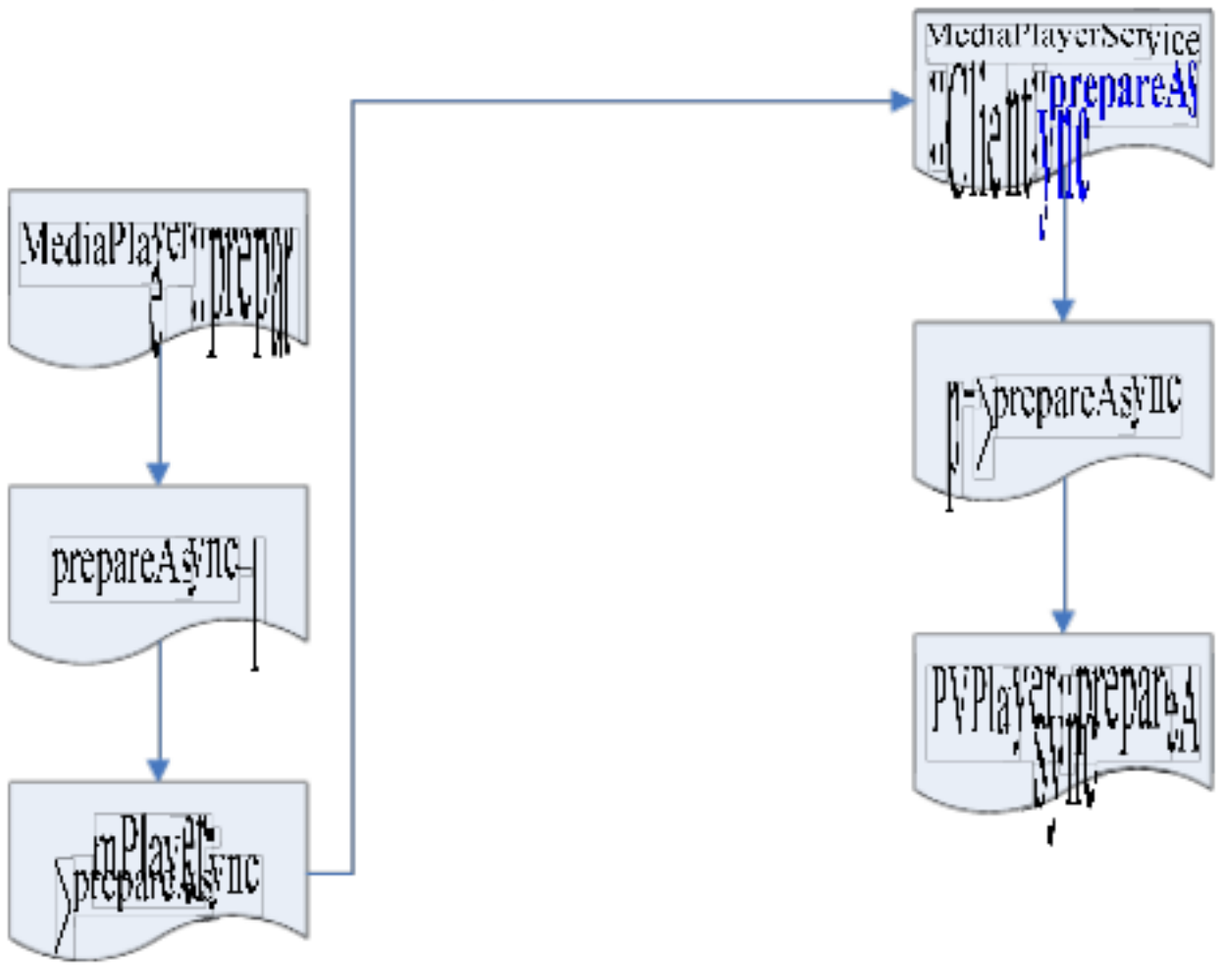
C） PVPlayerEngine的处理；

D） PVPlayerDatapath的处理；

下面分别讨论：

4.3.1 PVPlayer 的处理逻辑

先看看逻辑是怎么到 PVPlayer的：

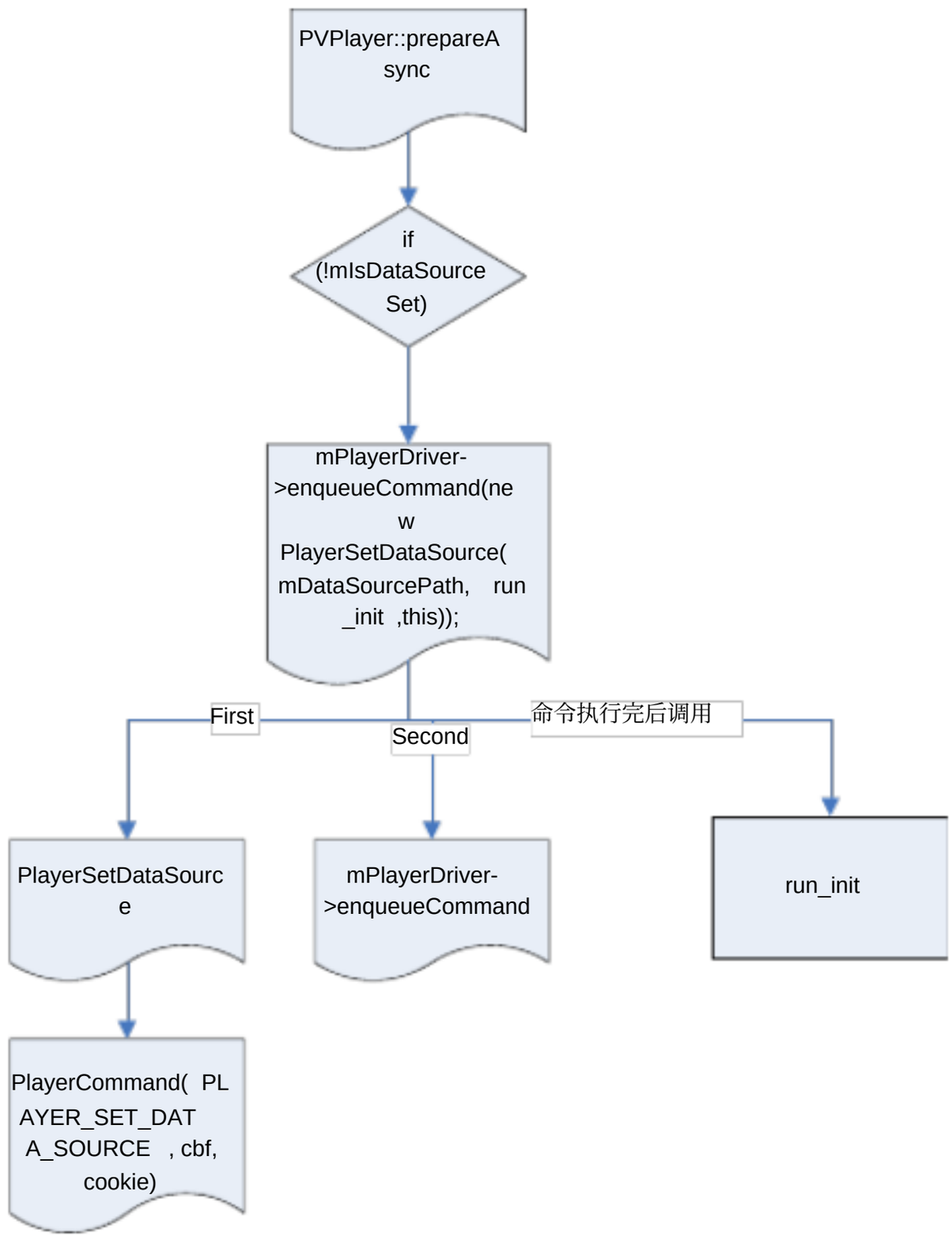


上层的 prepare到了这里就变成了 prepareAsync了，前面有提到过，这里就是把命令加入到队列，然后从 run 函数里面取出来处理，先看看命令的加入，如下图，主要做了两件事：

A） 构造并加入 PLAYER_SET_DATA_SOURCE 命令到队列里面；

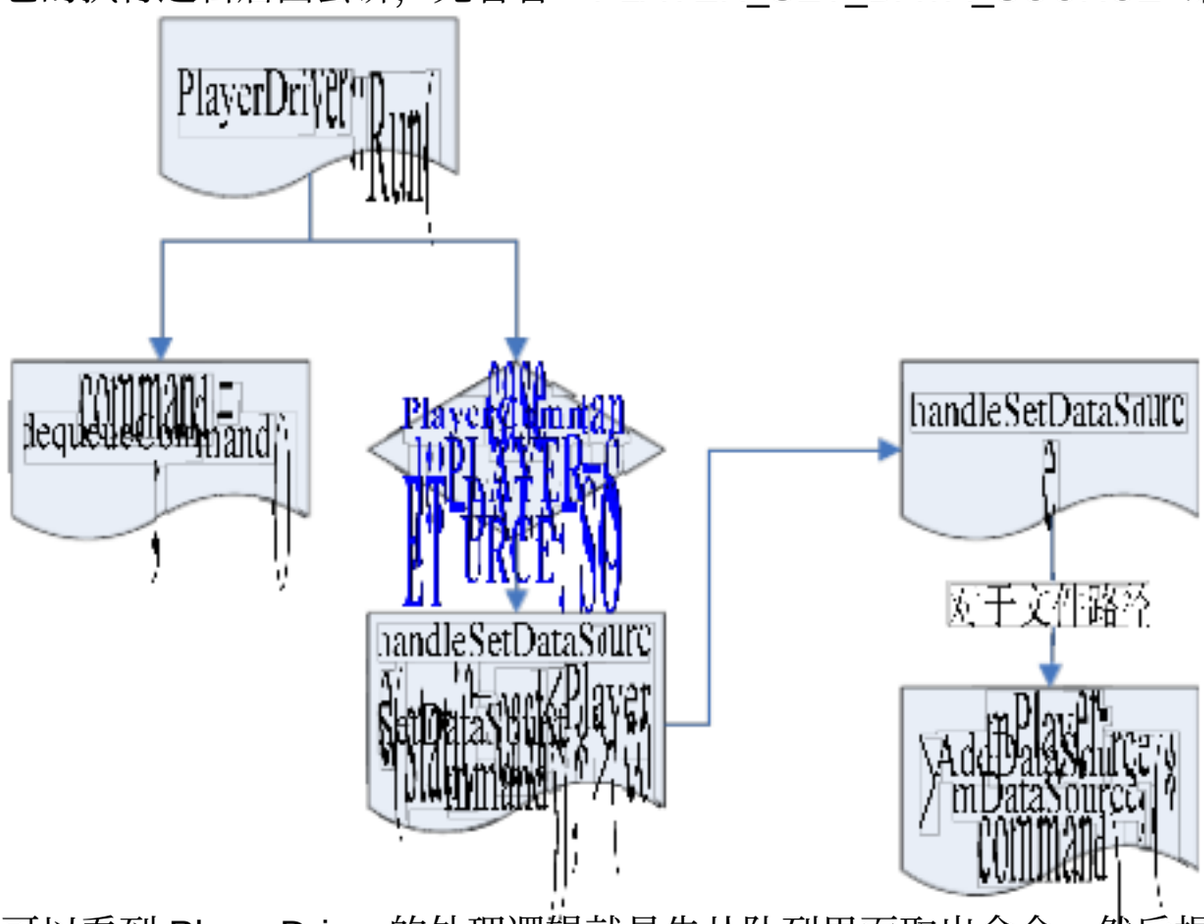
B） 设置回调函数；

这两步又会触发一些列的动作；



4.3.2 Playerdriver 的处理逻辑

可以看到先前的 setDataSource命令在这里会真正实现，而 run_init的执行要等到命令执行完毕才会被执行，它的执行逻辑后面会讲，先看看 PLAYER_SET_DATA_SOURCE 命令的处理：

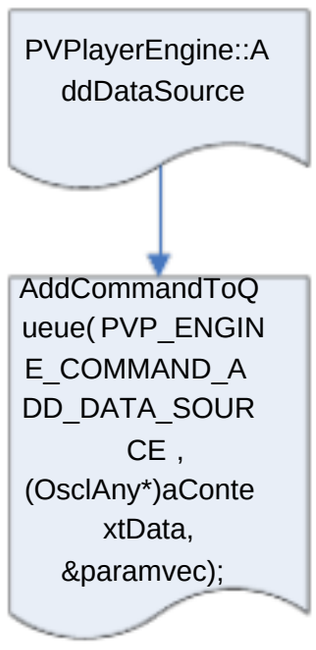


可以看到 PlayerDriver的处理逻辑就是先从队列里面取出命令，然后根据命令类型，比如这次是

PLAYER_SET_DATA_SOURCE 就调用 handleSetDataSource来处理，在这里再根据 source的类型初始化一下成员，比如 source的格式等，最后调用引擎层的 AddDataSource来处理；

4.3.3 引擎层的处理

引擎层的处理就是把它加入到队列里面，如下：



在AddCommandToQueue里面会调用 RunIfNotReady函数，这个函数的调用逻辑很复杂，不过你可以简单的理解为它会触发这个 active object的run函数，于是我们看看引擎层的 run函数如何处理这个命令，看看它的注释吧：

```
/*  Engine AO will execute commands in the following sequence:
    * 1) If Engine state is Resetting, which will happen when Engine does ErrorHandling,
    *    or is processing Reset or CancelAllCommands issued by the app, engine will NOT execute
    *    any other command during this state.
    * 2) If Engine is not in Resetting state, then it will process commands in the following order,
    *    which ever is true:
    *     (i) If Engine needs to do Error handling because of some error from Source Node or Datapath,
    *         either start error handling or complete it.
    *     (ii) If Engine has Reset or CancelAllCommands in CurrentCommandQueue,
    *         Engine will do CommandComplete for the CurrentCommand.
    *    (iii) If Engine has Prepare in CurrentCommandQueue, Engine will call DoPrepare again
    *         as a part of track selection logic.
    *     (iv) If Engine has CancelAllCommands or CancelAcquireLicense in Pending CommandQueue,
    *         Engine will start Cancel commands.
    *      (v) Go for Rollover if in Init State and Roll-over is ongoing.
    *     (vi) Process which ever command is pushed in Pending queue.
    * Engine will process any one of the command as listed above in the same order.
    * Every time engine AO is scheduled, engine will go through these steps.
    */
```

简单的说，它需要先考虑错误处理，然后考虑一些 reset，或者取消等命令，再次考虑是不是在 Prepare状态，如果是就会再次调用 DoPrepare，这在后面我们会看到对于 prepare的复杂的处理！！！然后考虑是不是正处于取消命令等的处理过程中，然后考虑需不需要回滚，最后才是处理在 iPendingCmds里面的命令，现在对于我们正在讨论的情况，就要到 iPendingCmds里面去处理了，先取出命令判断类型，如果是 PVP_ENGINE_COMMAND_ADD_DATA_SOURCE 类型就调用 DoAddDataSource来做实际的事情，它的调用逻辑如下：

在DoAddDataSource，分成格式是否被识别，做不同的处理，比如对于一个普通的视频，这时候它的格式是没有被识别的所以需要走的路线是：

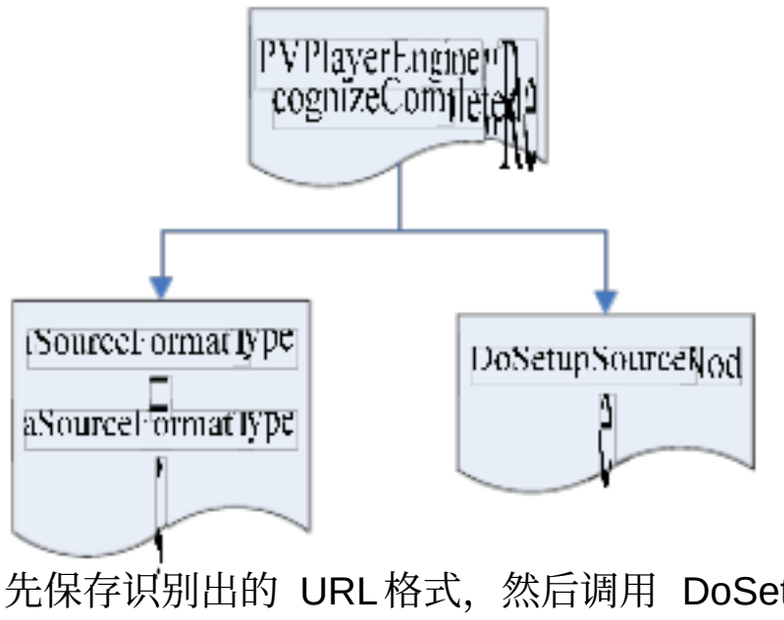
DoQuerySourceFormatType(aCmd.GetCmdId(), aCmd.GetContext());

也就是先调用识别器来识别源文件的格式：

iPlayerRecognizerRegistry.QueryFormatType（…）

至于它是如何识别的，就不分析了，只需要知道的是，如果识别以后，会通过：

iObserver->RecognizeCompleted(iSourceFormatType, iCmdContext)来通知上层，比如对于引擎层，它的RecognizeCompleted函数将会被调用，调用逻辑如下：



先保存识别出的 URL 格式，然后调用 DoSetupSourceNode来处理，

这个函数主要的作用就是：

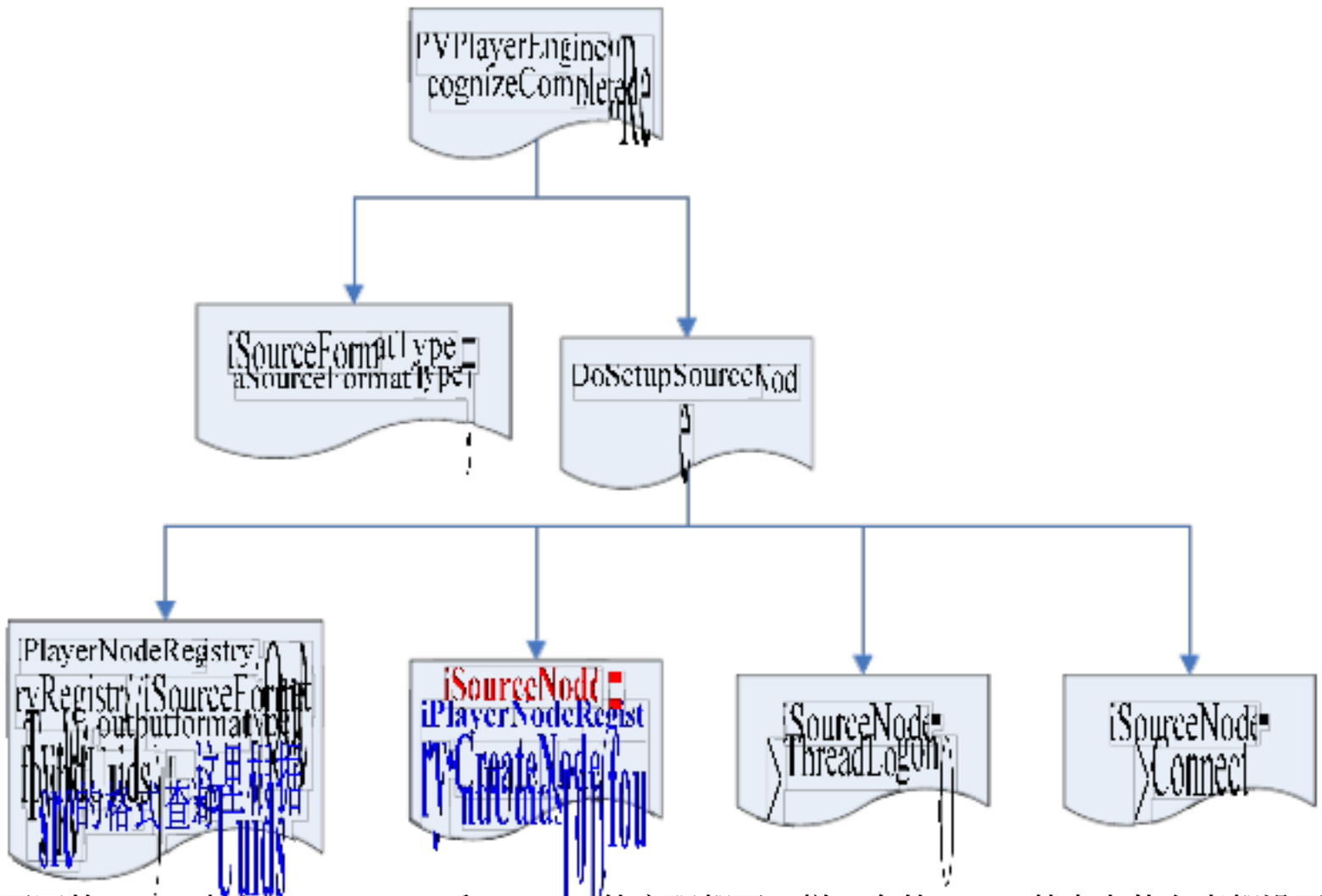
1， 根据源文件格式查找出对应的 UUID ；

2， 根据 UUID 创建source node；

3， 连接 source node;

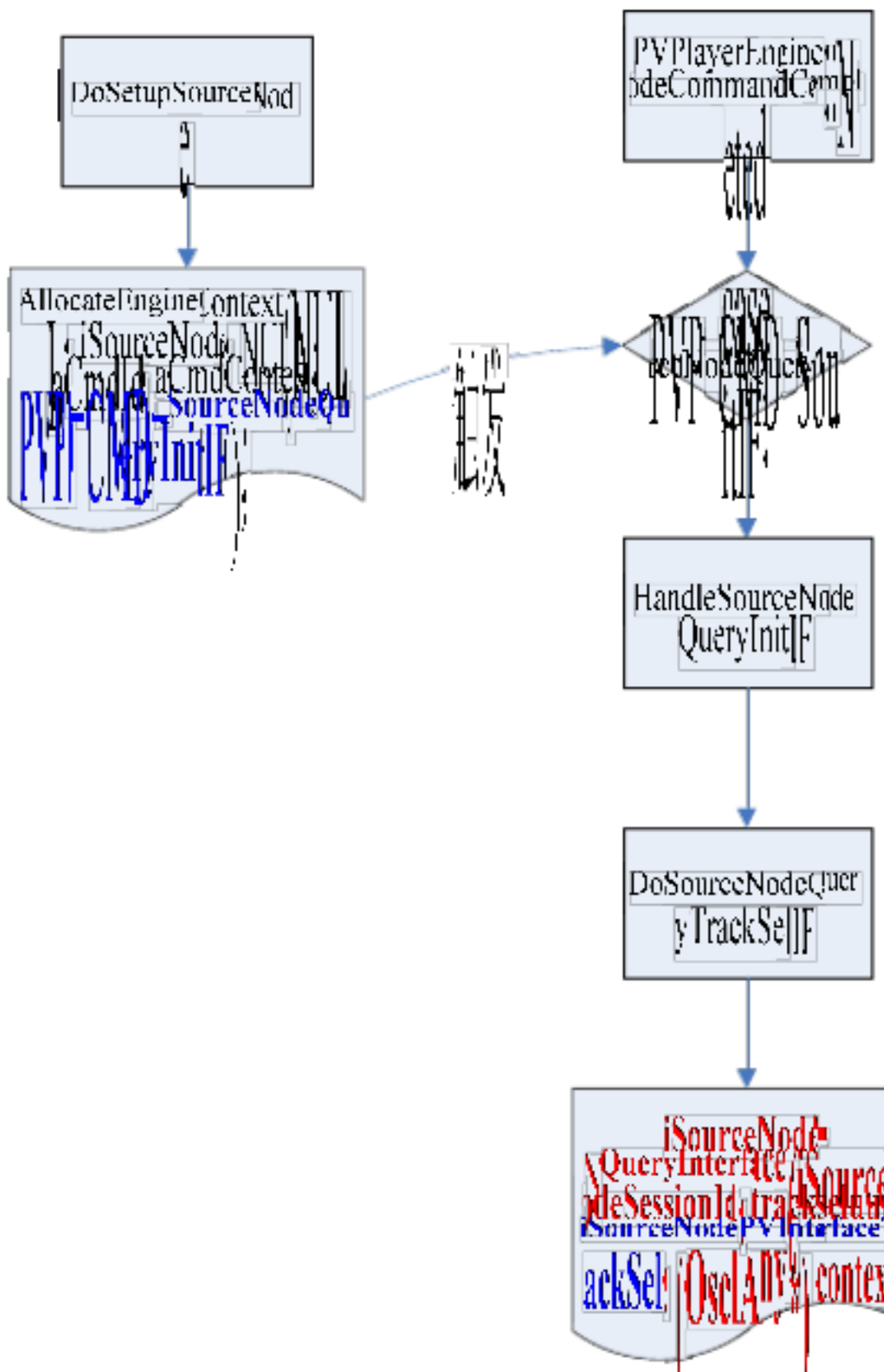
4， 查询 PVP_CMD_SourceNodeQueryInitIF接口；

如下：



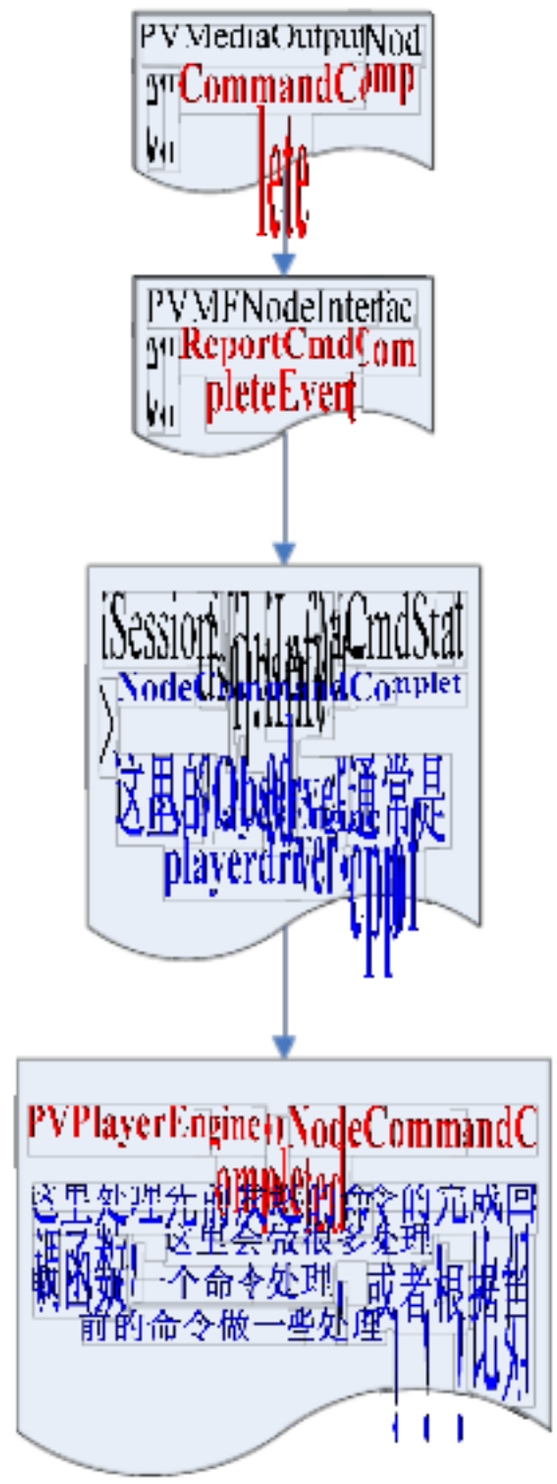
不同的 Node对于ThreadLogon和connect的实现都不一样，有的 node基本上什么事都没干，这两个函数的目的， 就有点像 session也就是说如果事情需要在一个会话里面初始化的，那么就需要把事情在那里做，比如特定于 session的空间的分配等（主要目的是为了多线程的并发处理）；

这里需要重点说一下的是第四步，这里通过查询 PVP_CMD_SourceNodeQueryInitIF 接口后会触发HandleSourceNodeQueryInitIF的执行，它的逻辑如下：

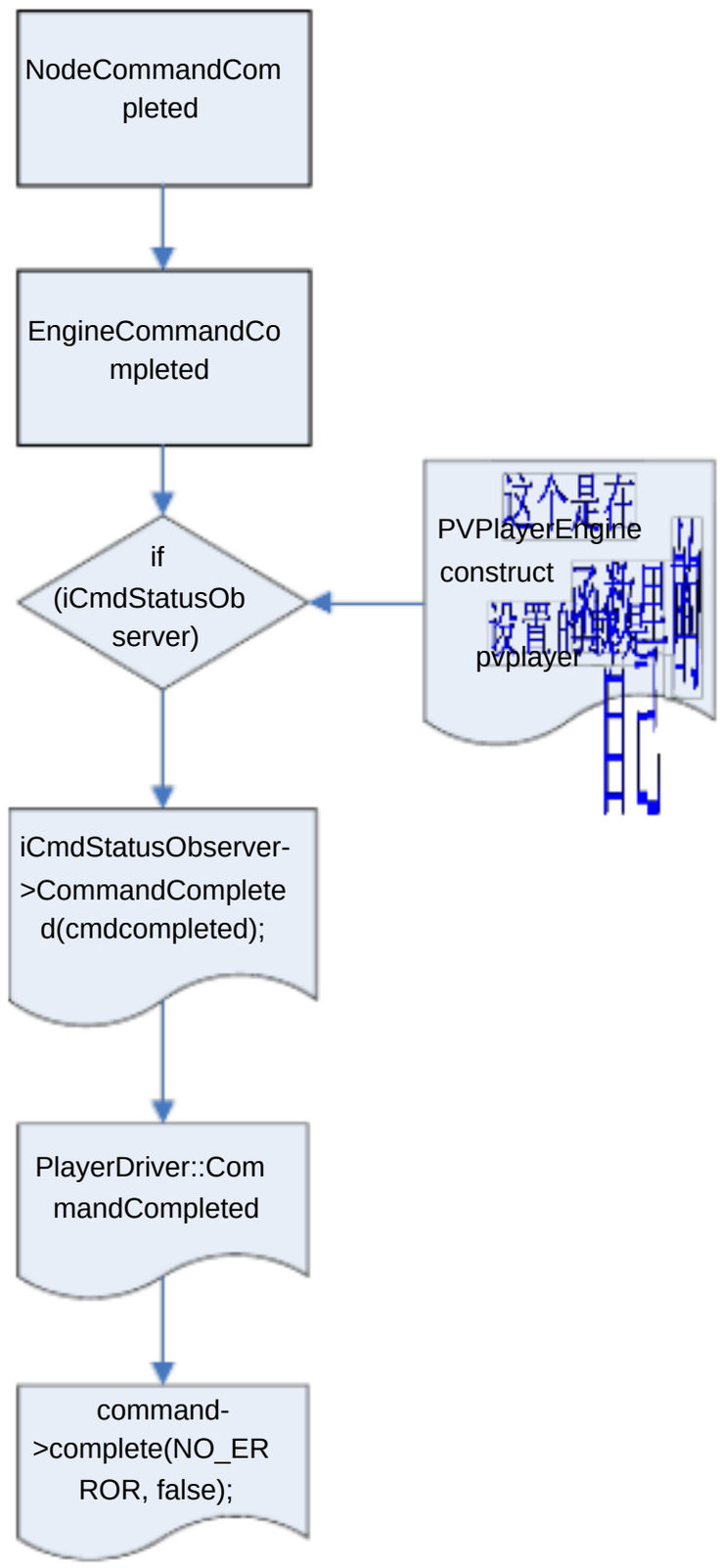


如果成功的话， iSourceNodePVInterfaceTrackSel会被初始化，这个变量非常重要，因为它负责文件的解析，也就是 track的建立，详细信息，后面会讲；

OK，我们现在假定我们这些事都已经成功的完成了，那么先前设好的 run_init 函数将会被执行了，也许你会说，它的回调逻辑也就是执行逻辑是怎么来的， OK，好吧，我们来看看：



每当一个 node执行完一个命令后，将会触发它的 CommandComplete函数，在那里，它会调用 node的观察者的 NodeCommandCompleted，在这里，我们的观察者就是 PVPlaerEngine，于是它的 NodeCommandCompleted函数将会被执行，它当然还得回到 PlayerDriver层才行，它的逻辑如下：



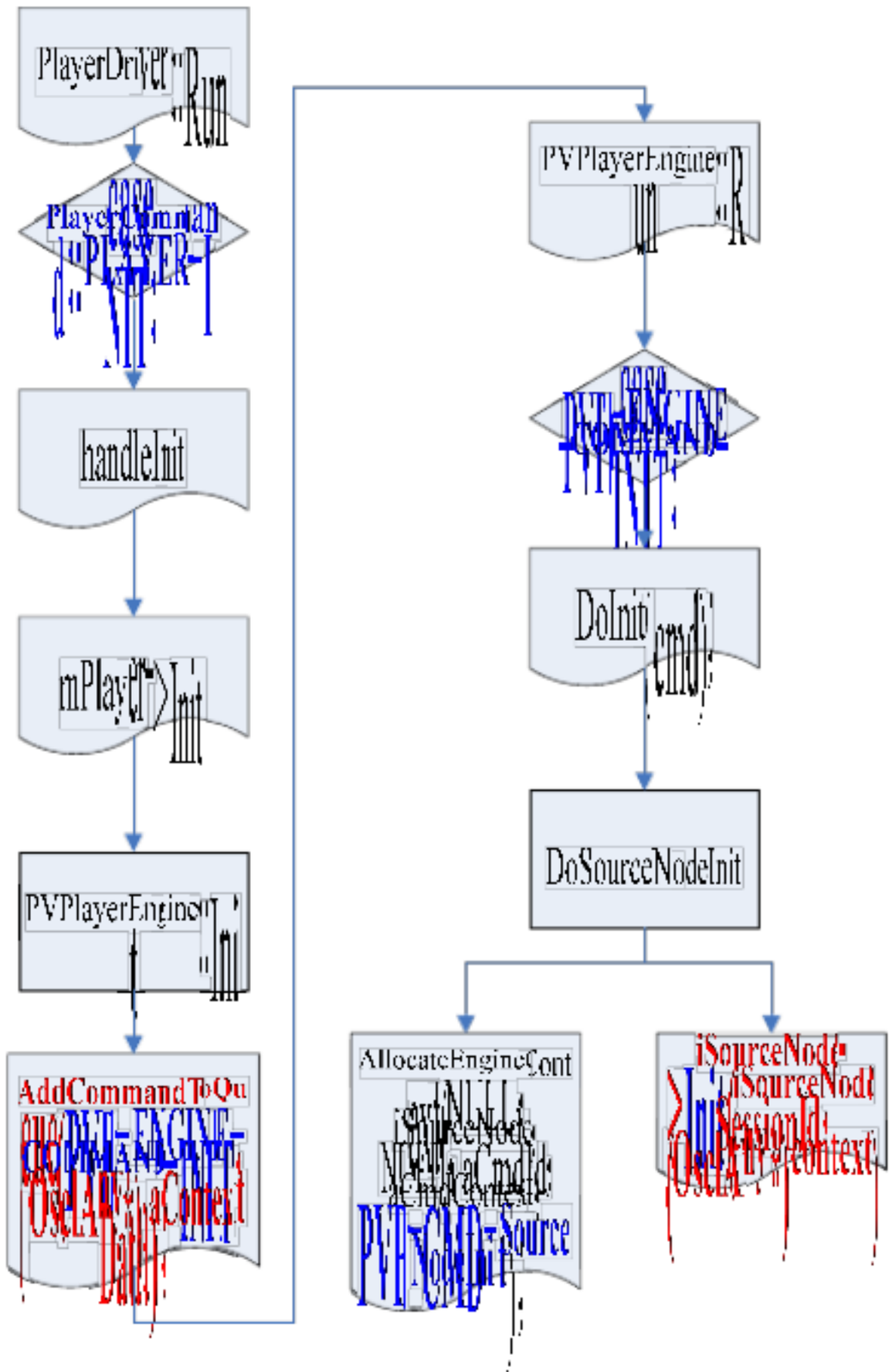
到这里，我们先前设定的 command的complete函数终于得以执行，也就是 run_init函数被执行了，现在看看它的逻辑：

4.3.4 Run_init 的逻辑

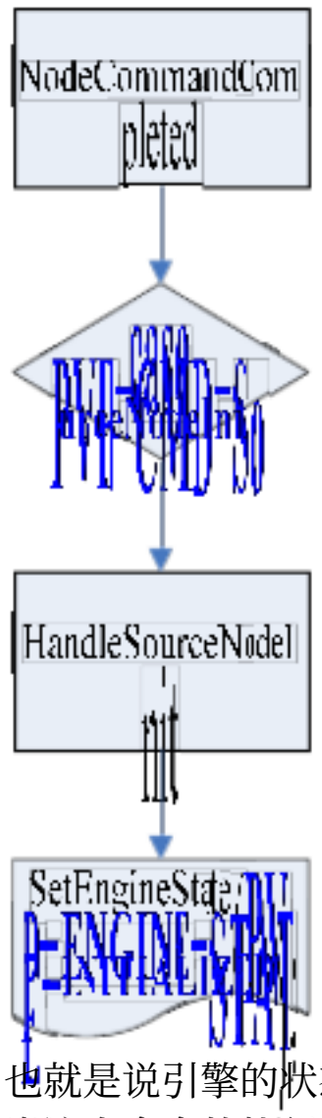
这个代码很短，如下：

```
void PVPlayer::run_init(status_t s, void *cookie, bool cancelled)
{
    LOGV("run_init s=%d, cancelled=%d", s, cancelled);
    if (s == NO_ERROR && !cancelled) {
        PVPlayer *p = (PVPlayer*)cookie;
        p->mPlayerDriver->enqueueCommand (new PlayerInit(run_set_video_surface , cookie));
    }
}
PlayerInit(media_completion_f cbf, void* cookie) :
        PlayerCommand(PLAYER_INIT, cbf, cookie) {}
```

和其它命令的处理逻辑一样，就是加入到 PlayerDriver的命令队列里面，等待被处理；像run_init 本身的执行逻辑一样，这里的 run_set_video_surface ， 也就是命令本身设置的回调函数，它的调用逻辑前面已经讲过了，它的执行逻辑后面再讲，这里先看看对于 PLAYER_INIT 的命令， playerdriver是怎么处理的，如下：



和上面的 setDataSource 逻辑一样，最后这些命令的处理都得到引擎层去实现，对于 init 命令，最后会调用到 sourceNode的init 命令来处理（后面会讲到 decode以及 sinknode的初始化）；这个命令处理后会先触发引擎层的回调函数，逻辑如下：



也就是说引擎的状态现在已经变成了 PVP_ENGINE_STATE_INITIALIZED ！！！

当这个命令的执行完毕以后，我们先前设置好的 playerdriver层的命令回调函数 run_set_video_surface 将会被触发了，在讨论它的逻辑之前，需要先看看这段代码，前面已经贴出来过：

```
static void
android_media_MediaPlayer_prepareAsync(JNIEnv *env, jobject thiz)
{
    sp<MediaPlayer> mp = getMediaPlayer(env, thiz);
    if (mp == NULL ) {
        jniThrowException(env, "java/lang/IllegalStateException", NULL);
        return;
    }
    jobject surface = env->GetObjectField(thiz, fields.surface);
    if (surface != NULL) {
        const sp<Surface> native_surface = get_surface(env, surface);
        LOGV("prepareAsync: surface=%p (id=%d)",
             native_surface.get(), native_surface->ID());
        mp->setVideoSurface(native_surface);
    }
    process_media_player_call( env, thiz, mp->prepareAsync()  , "java/io/IOException", "Prepare Async failed." );
}
```

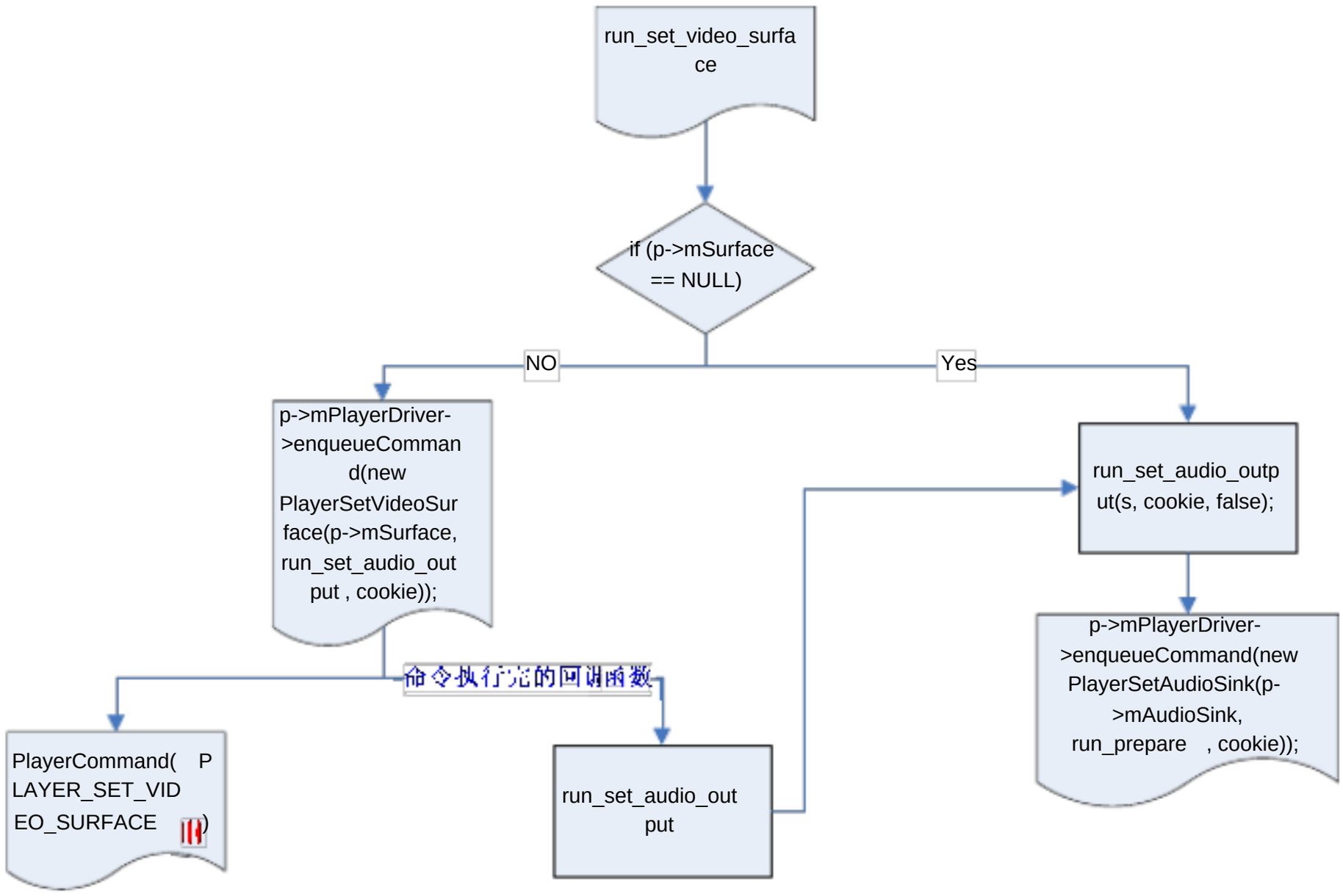
也就是说在 JNI层调用 prepareAsync的时候会先设置 surface，对于PVPlayer,这个逻辑很简单， 只是保存一下：

```
status_t PVPlayer::setVideoSurface(const sp<ISurface>& surface)
{
    LOGV("setVideoSurface(%p)", surface.get());
    mSurface = surface;
    return OK;
}
```

OK，现在可以看看 mp->setVideoSurface(native_surface) 了：

4.3.5 Audio 输出和 video 输出的设置

先看看这个简单而丑陋的图：



对于命令的加入和取出以及回调函数的逻辑，前面已经说过，以后我都直接讨论函数本身了，这里需要分成至少三部分：

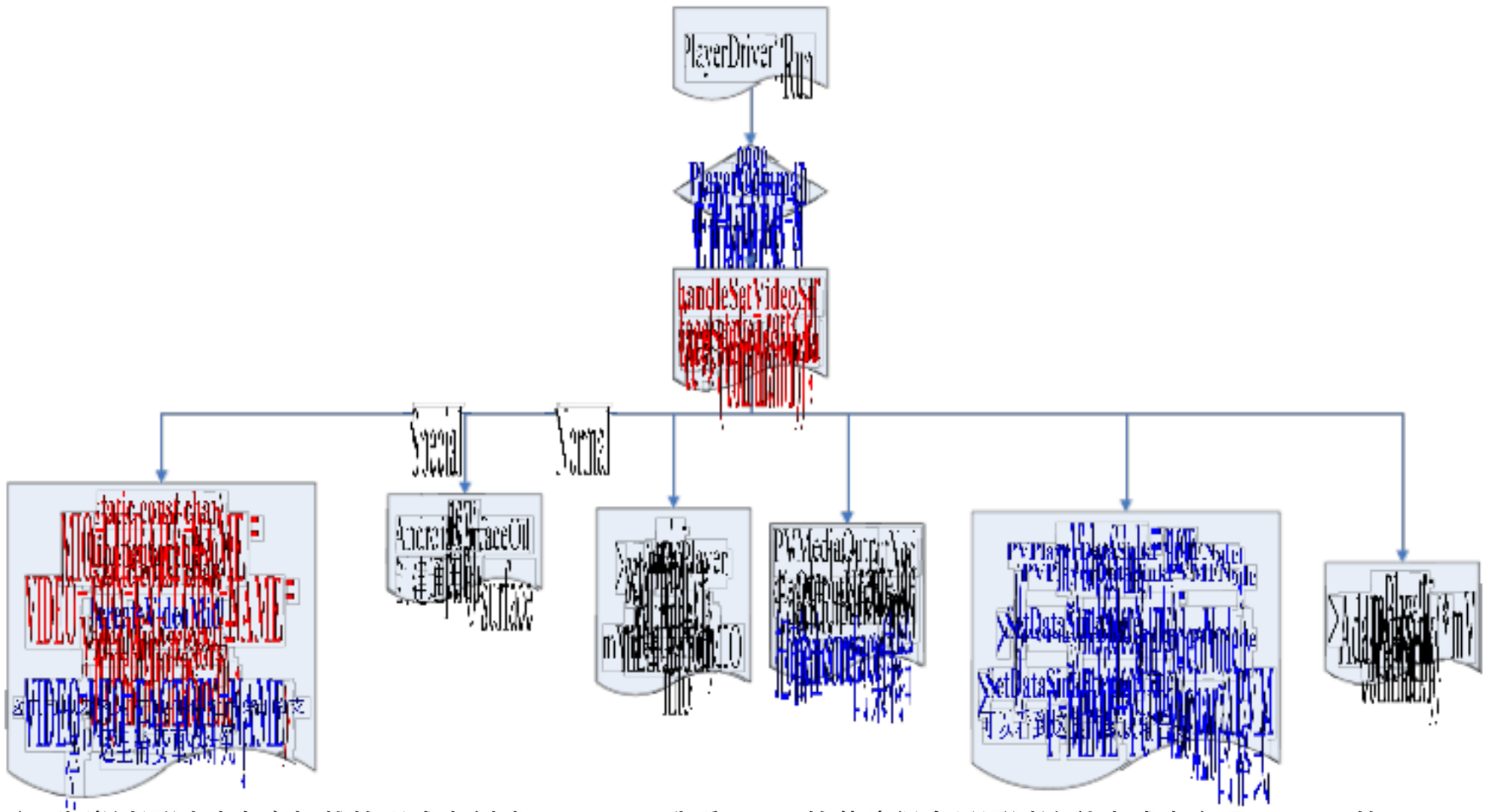
A） PLAYER_SET_VIDEO_SURFACE 的命令的处理；

B） 对于上面的命令执行完毕后的回调函数 run_set_audio_output，即PLAYER_SET_AUDIO_SINK 命令的处理;

C） 对于PLAYER_SET_AUDIO_SINK 命令处理完毕后的回调函数 run_prepare;

4.3.5.1 PLAYER_SET_VIDEO_SURFACE 处理

它的处理逻辑如下：



a） 先尝试通过动态库加载的形式来创建 MIO ， 我看omap的芯片很多是通过这种方式来实现 MIO 的；

b) 如果没有实现硬件特定的 MIO ， 就创建一个 Android 框架提供的通用的 MIO ；

c) 为MIO 设置输出 surface，以及传递一个 PVPlayer ，用于接收 MIO 传递上来的事件（目前未看见 marvel 的相关实现）；

d) mVideoOutputMIO = mio; 保存此 MIO ；

e) 创建 mVideoNode，这个是用来和 MIO 交互的 component ；

f) 创建 mVideoSink 并设置 sink node，即 SetDataSinkNode(mVideoNode);

g) 设置视频输出格式为 PVMF_MIME_YUV420 ；

h) mPlayer->AddDataSink(*mVideoSink, command) ，让引擎层添加 data sink ；

这里需要特别说明的是 marvel用的是通用的 MIO，也就是 AndroidSurfaceOutput,然后在通过 set把fb2的surface设置进去（我的猜想），在输出 component也就是 mVideoNode里面利用这个 fb2把数据写出去，而omap是通过一个扩展的库 libopencorehw.so 来扩展实现的；

另外，这里的 mVideoNode和mVideoSink有点混淆，其实可以这么理解， mVideoSink是OPENCORE层需要的 node，而mVideoNode是所谓的 component，它是和解码器一个层次的， node在更上面一层；

下面来看看引擎层的 AddDataSink：

它先是加到命令队列，然后从命令队列取出来调用 DoAddDataSink处理，简略示意如下：

```
PVPlayerDataSink* datasink = (PVPlayerDataSink*)(aCmd.GetParam(0).pOsclAny_value);
 PVPlayerEngineDatapath newdatapath;
 newdatapath.iDataSink = datasink   ;

  // Add a new engine datapath to the list for the data sink
  iDatapathList.  push_back (newdatapath);

  EngineCommandCompleted(aCmd.GetCmdId(), aCmd.GetContext(), PVMFSuccess);
```

简单的说就是加入到引擎所拥有的 datapathlist里面，然后调用回调函数；

4.3.5.2 run_set_audio_output 的处理逻辑

当PLAYER_SET_VIDEO_SURFACE 处理完后，对应的 command的回调函数将会被触发了，代码如

下：

```
void PVPlayer::run_set_audio_output(status_t s, void *cookie, bool cancelled)
{
    LOGV("run_set_audio_output s=%d, cancelled=%d", s, cancelled);
    if (s == NO_ERROR && !cancelled) {
        PVPlayer *p = (PVPlayer*)cookie;
        p->mPlayerDriver->enqueueCommand(new PlayerSetAudioSink   (p->mAudioSink, run_prepare,
cookie));
    }
}
PlayerCommand(PLAYER_SET_AUDIO_SINK     , cbf, cookie), mAudioSink(audioSink) {}
```

它的处理逻辑和 surface的逻辑基本差不多，如下：

```
void PlayerDriver::handleSetAudioSink(PlayerSetAudioSink* command)
{
    int error = 0;
    if (command->audioSink()->realtime()) {
        LOGV("Create realtime output");
        mAudioOutputMIO = new AndroidAudioOutput();
    } else {
        LOGV("Create stream output");
        mAudioOutputMIO = new AndroidAudioStream();
    }
    mAudioOutputMIO-> setAudioSink(command->audioSink());

    mAudioNode =  PVMediaOutputNodeFactory::CreateMediaOutputNode(mAudioOutputMIO);
    mAudioSink = new PVPlayerDataSinkPVMFNode;

    ((PVPlayerDataSinkPVMFNode *)mAudioSink)->  SetDataSinkNode (mAudioNode);
    ((PVPlayerDataSinkPVMFNode
*)mAudioSink)->SetDataSinkFormatType((char*) PVMF_MIME_PCM16     );

    OSCL_TRY(error, mPlayer->AddDataSink  (*mAudioSink, command));
    OSCL_FIRST_CATCH_ANY(error, commandFailed(command));
}
```

如果您要问这个 audioSink的来源，那就要追溯到 mediaplayerservice::Clinet里面的 setDataSource了，相关代码如下：

```
sp<MediaPlayerBase> p =createPlayer(playerType);
        if (p == NULL) return NO_INIT;
if (!p->hardwareOutput()) {
            mAudioOutput = new AudioOutput();
            static_cast<MediaPlayerInterface*>(p.get())->    setAudioSink(mAudioOutput)    ;
 }
```

这个P就是返回给客户端的那个 player，所以 audiosink来自于此，其它的逻辑和视频差不多，也是创建了一个 MIO，并且调用 mPlayer的AddDataSink加入到 datapathlist里面去了；

4.3.5.3 run_prepare 的处理

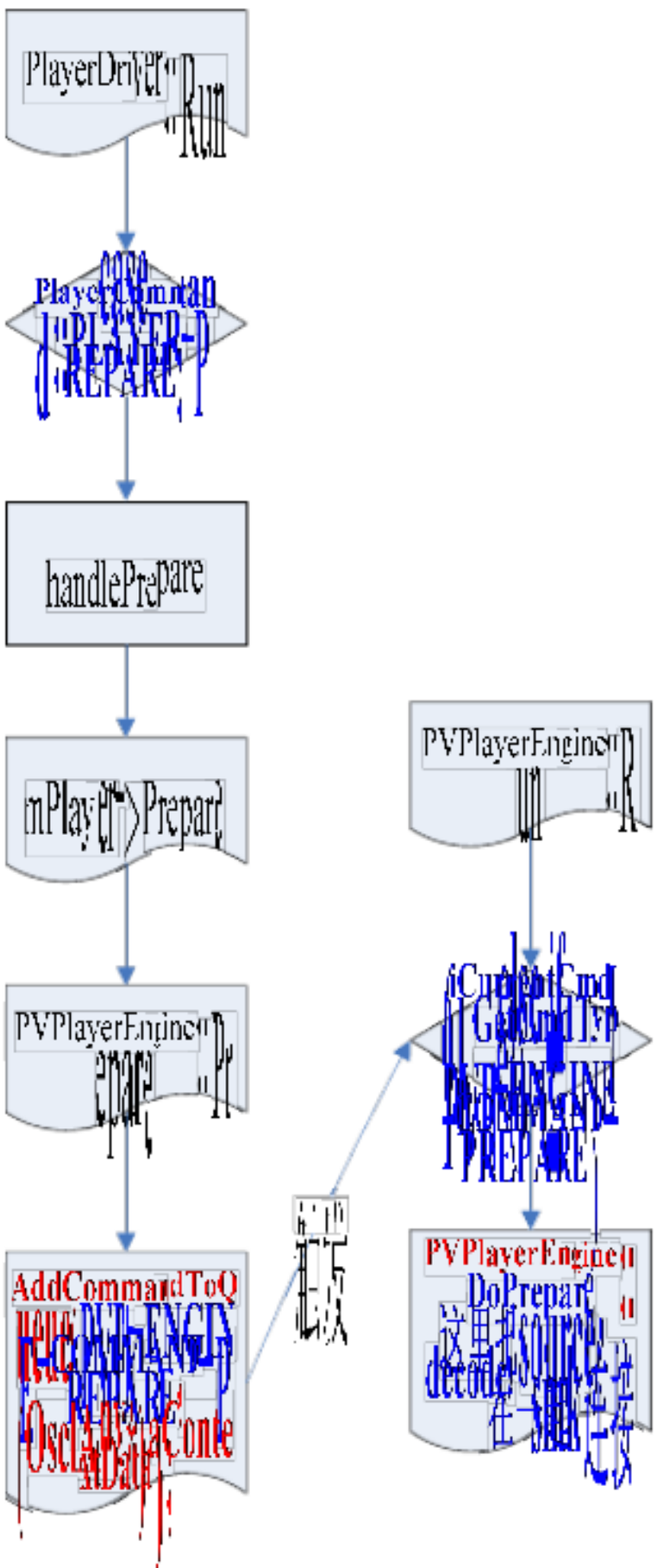
prepare 的处理终于开始了，它本身只是加入到队列里面去，然后调用 handlePrepare，在里面基本上就是先设置参数，然后调用：

mPlayer->Prepare(command) 来处理；

这个mPlayer 就是我们所说的引擎层的 player 了；

4.4 引擎层 prepare 的处理

前面已经提到 run_prepare最后都是通过 mPlayer->Prepare来实现的，下面开始看看它的入口逻辑：



在进入 DoPrepare之前，需要先说明一下，

引擎层对于 prepare的处理分成四个阶段，如下：

A） 引擎处于 PVP_ENGINE_STATE_INITIALIZED ，这是第一次进入 DoPrepare的情况；

B） 引擎处于 PVP_ENGINE_STATE_TRACK_SELECTION_1_DONE ；

C） 引擎处于 PVP_ENGINE_STATE_TRACK_SELECTION_2_DONE ；

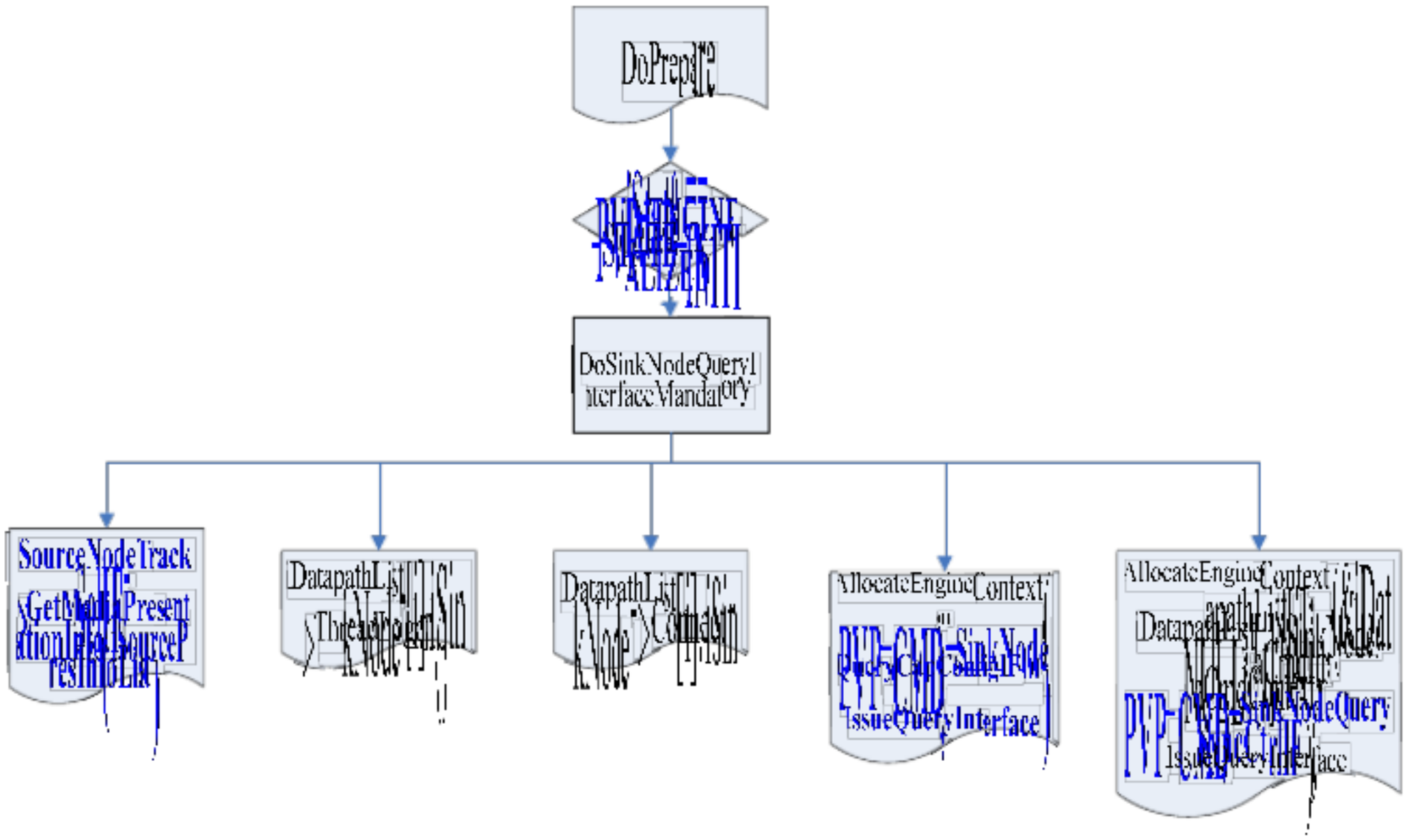
D） 引擎处于 PVP_ENGINE_STATE_TRACK_SELECTION_3_DONE ；

下面分别对这几种情况进行讨论：

4.4.1 PVP_ENGINE_STATE_INITIALIZED 状态时的处理

4.4.1.1 引擎层处理

先看看这个逻辑图：



处于第一个状态时做的事情包括：

1, 根据iSourceNodeTrackSelIF接口对象取得源文件的 track信息（后面会讲）；

2, sinknode->threadLogon;sinknode->Connect就是做一些与线程相关的初始化；

3, 查询这个 sink node的配置接口；

4, 查询这个 sink node的同步控制接口；

这两个命令都会通过 IssueQueryInterface的方式来发送给节点层，然后当命令返回后在引擎层的 NodeCommandCompleted函数里面会根据当前的状态（ PVP_ENGINE_STATE_PREPARING ）然后根据命令的类型做处理，比如刚才这个两个命令的逻辑如下：

```
case PVP_CMD_SinkNodeQuerySyncCtrlIF:
 case PVP_CMD_SinkNodeQueryCapConfigIF:
        HandleSinkNodeQueryInterfaceMandatory(*nodecontext, aResponse);
        break;
```

而在函数 HandleSinkNodeQueryInterfaceMandatory 里面会保存这个配置接口和同步接口，相关代码如下：

```
OSCL_EXPORT_REF bool PVMediaOutputNode::queryInterface(const PVUuid& uuid,
PVInterface*& iface)
{
   if (uuid == PvmfNodesSyncControlUuid)
   {
      PvmfNodesSyncControlInterface* myInterface =
OSCL_STATIC_CAST(PvmfNodesSyncControlInterface*,      this);
      iface = OSCL_STATIC_CAST(PVInterface*, myInterface);
      ++iExtensionRefCount;
   }
   else if (uuid == PVMI_CAPABILITY_AND_CONFIG_PVUUID)
   {
```

```
        PvmiCapabilityAndConfig* myInterface =
OSCL_STATIC_CAST(PvmiCapabilityAndConfig*,        this);
        iface = OSCL_STATIC_CAST(PVInterface*, myInterface);
        ++iExtensionRefCount;
    }
    else
    {
        iface = NULL;
        return false;
    }

    return true;
}
```

所以，我们都提供了相关接口，并且就是 PVMediaOutputNode本身（this）；

取得这个两个接口后， HandleSinkNodeQueryInterfaceMandatory 先保存：

```
aNodeContext.iEngineDatapath->iSinkNodeCapConfigIF =
(PvmiCapabilityAndConfig*)aNodeContext.iEngineDatapath->iSinkNodePVInterfaceCapConf
ig;
```

函数最后会调用 DoSinkNodeInit来对 SinkNode进行初始化；

DoSinkNodeInit ——>IssueSinkNodeInit——>aDatapath->iSinkNode->Init

最后就是这个 SinkNode的初始化了（ 这里已经看到了 sink的初始化 ）， 下面来看看 node本身的处理吧；

4.4.1.2 Node 层及以下的处理

简单的看一下示意代码：

```
OSCL_EXPORT_REF PVMFCommandId PVMediaOutputNode::Init(PVMFSessionId s, const
OsclAny* aContext)
{
    PVLOGGER_LOGMSG(PVLOGMSG_INST_LLDBG, iLogger,
PVLOGMSG_STACK_TRACE,
                    (0, "PVMediaOutputNode::Init() called"));
    PVMediaOutputNodeCmd cmd;
    cmd.PVMediaOutputNodeCmdBase::Construct(s, PVMF_GENERIC_NODE_INIT,
aContext);
    return QueueCommandL(cmd);
}
```

从这里可以看出， Node层的处理几乎和它的父亲或者祖父如出一辙，就是一个命令队列，然后把命令一个个加入进去，在处理函数 run里面调用 ProcessCommand来处理，另外调用CommandComplete来通知上层，命令已经执行完毕，如下：

```
void PVMediaOutputNode::Run()
{
    //Process async node commands.
    if (!iInputCommands.empty())
    {
        ProcessCommand();
    }

    //Check for completion of a flush command...
    if (iCurrentCommand.size() > 0
            && iCurrentCommand.front().iCmd == PVMF_GENERIC_NODE_FLUSH
            && PortQueuesEmpty())
    {
        //Flush is complete.
```

```
        CommandComplete(iCurrentCommand, iCurrentCommand.front(), PVMFSuccess);
    }
}
```

对于 init 命令来说，处理逻辑如下：

```
  case PVMF_GENERIC_NODE_INIT:
                cmdstatus = DoInit(aCmd);
                break;
```

DoInit ——> SendMioRequest(aCmd,EQueryClockExtension);

这里需要注意的是后面的那个参数，并不是 EInit,而是 EQueryClockExtension,

所以处理的逻辑就是查询 MIO 是否支持 PvmiClockExtensionInterfaceUuid 的扩展，而对于 AndroidSurfaceOutput 的查询来说：

```
PVMFCommandId AndroidSurfaceOutput::QueryInterface(const PVUuid& aUuid,
PVInterface*& aInterfacePtr, const OsclAny* aContext)
{
    PVLOGGER_LOGMSG(PVLOGMSG_INST_LLDBG, iLogger,
PVLOGMSG_STACK_TRACE, (0, "AndroidSurfaceOutput::QueryInterface() called"));

    PVMFCommandId cmdid = iCommandCounter++;

    PVMFStatus status = PVMFFailure;
    if (aUuid == PVMI_CAPABILITY_AND_CONFIG_PVUUID)
    {
        PvmiCapabilityAndConfig* myInterface =
OSCL_STATIC_CAST(PvmiCapabilityAndConfig*, this);
        aInterfacePtr = OSCL_STATIC_CAST(PVInterface*, myInterface);
        status = PVMFSuccess;
    }
    else
    {
        status = PVMFFailure;
    }

    CommandResponse resp(status, cmdid, aContext);
    QueueCommandResponse(resp);
    return cmdid;
}
```

除了 PVMI_CAPABILITY_AND_CONFIG_PVUUID 都不支持

OK，到这里基本上 Node层即 MIO 的处理就完毕了，现在要回到引擎层处理 PVP_CMD_SinkNodeInit 命令的回调函数了；

4.4.1.3 HandleSinkNodeInit 的处理

当 PVP_CMD_SinkNodeInit 命令在 Node层执行完毕后， node层的 run函数里面会通过 CommandComplete函数通知上层， 最终，在 PVPlayerEngine::NodeCommandCompleted函数里面讲会根据状态和命令类型调用这个函数 HandleSinkNodeInit，它的逻辑比较简单，就干了这件事：

SetEngineState(PVP_ENGINE_STATE_TRACK_SELECTION_1_DONE);

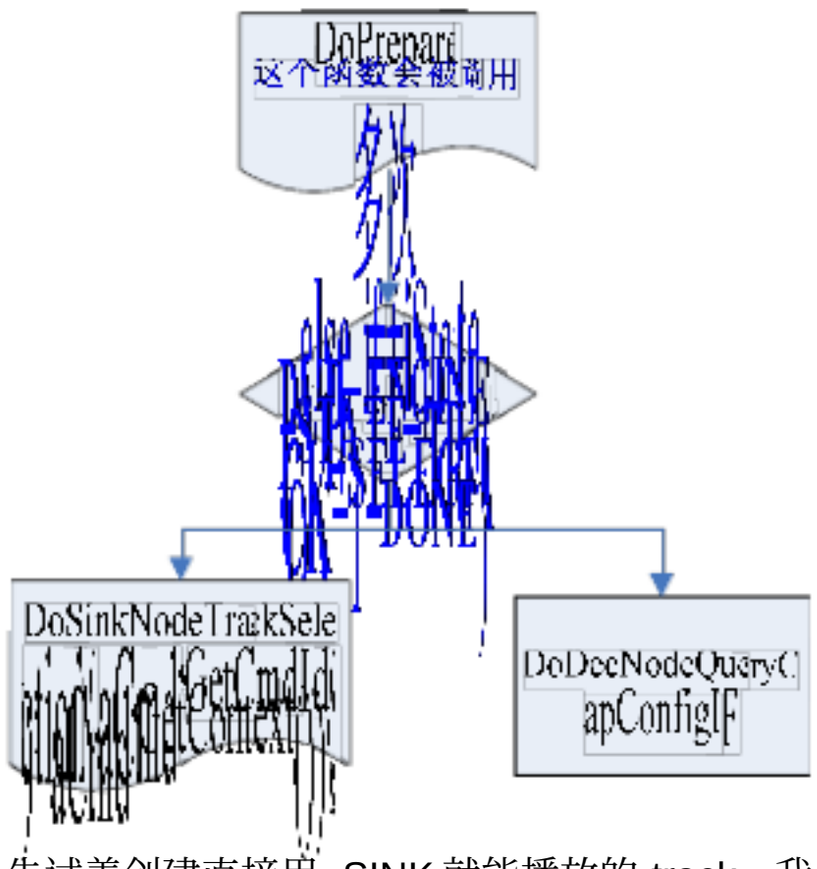
这里终于更新了状态，可以进入下一个轮回了，然后调用

RunIfNotReady();

重新启动引擎层的 run来函数处理新的状态（状态机的轮回）；

4.4.2 PVP_ENGINE_STATE_TRACK_SELECTION_1_DONE 逻辑

简单示意如下：



先试着创建直接用 SINK 就能播放的 track，我只讨论简单正常的情况，所以对于这种特例我就不分析了，接着看另外一种情况， 需要解码器才能工作的 track，简单说一下所谓 track，我的理解就是一个纯的流的描述，比如：音频，视频，字幕；一个视频文件一般都包括这三个 track，所以需要分开控制，当然必须得有一种机制实现复杂的同步，这个会在后面讲；

我们先看看 DoDecNodeQueryCapConfigIF的逻辑， 为了不涉及复杂的解码器的逻辑， 所以我这次就不进入 decode的内部逻辑来，对于后面分析数据的流动的时候，我们再去分析；目前我们只讨论引擎层的内部逻辑，否则写上几本书也写不完：

大体的逻辑是这样的，对于每一个 track，我们都需要有一个 pathlist，而对于每一个 pathlist，我们都需要为其寻找 decode component，所以呢，代码里面的处理逻辑就是遍历 tracklist，为每一个对应的 pathlist查找decoder，查找的方法为：

A） 先根据源文件的格式取得 uuid；

B） 根据uuid创建decoder，并加入到 pathlist里面；

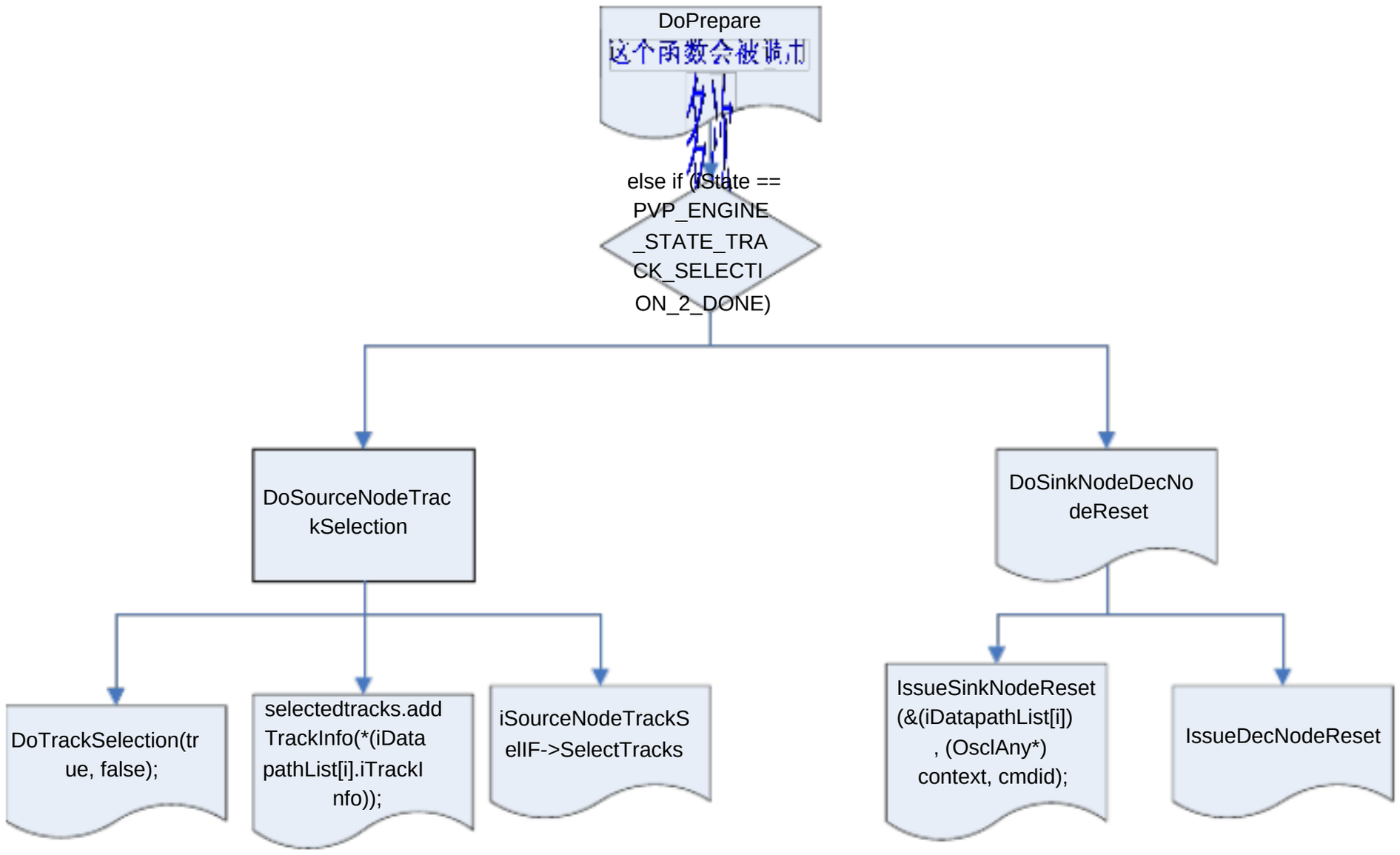
C） 对于新的 decode调用其初始化函数；

D） 初始化函数执行完毕后会在 PVPlayerEngine::NodeCommandCompleted里面调用 HandleDecNodeInit处理；

E） 在这里面会调用 SetEngineState(PVP_ENGINE_STATE_TRACK_SELECTION_2_DONE); 设置新的状态；从而触发状态机往前走；

4.4.3 PVP_ENGINE_STATE_TRACK_SELECTION_2_DONE 的逻辑

简单示意如下：



在这个状态里面主要做了几件事：

A） 根据 src 的格式以及用户的喜好设置确定 tracklist ；

B） aDatapath->iSinkNode->Reset 调用 sinkNode 的 Reset 函数进行状态复位， 同时也要对 MIO 进行 reset 复位；

C） 对于 decode 进行 reset 复位；

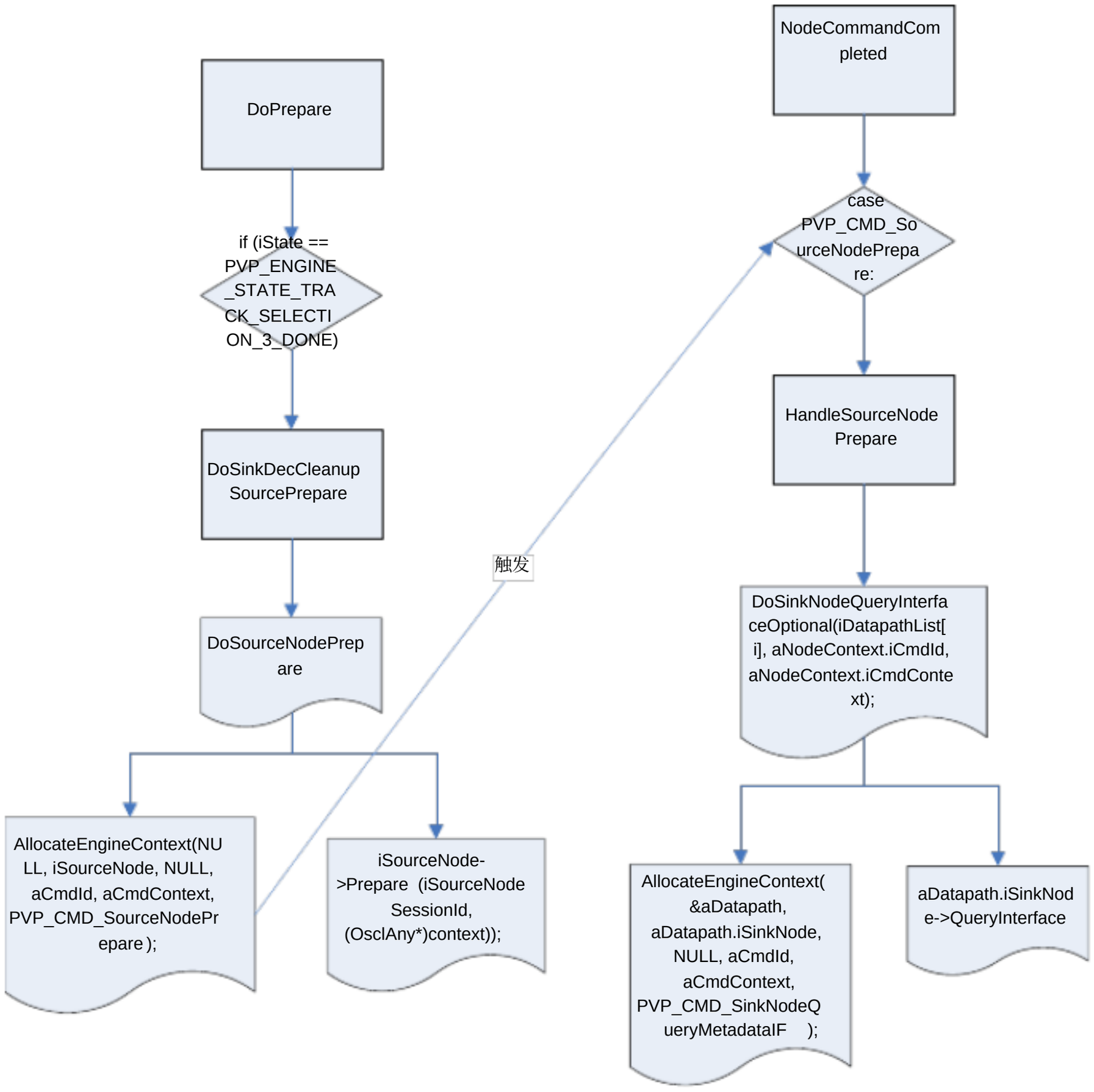
等这两个命令执行完毕后， NodeCommandCompleted 会被执行再调用 HandleSinkNodeDecNodeReset 进行状态的更新：

SetEngineState(PVP_ENGINE_STATE_TRACK_SELECTION_3_DONE);

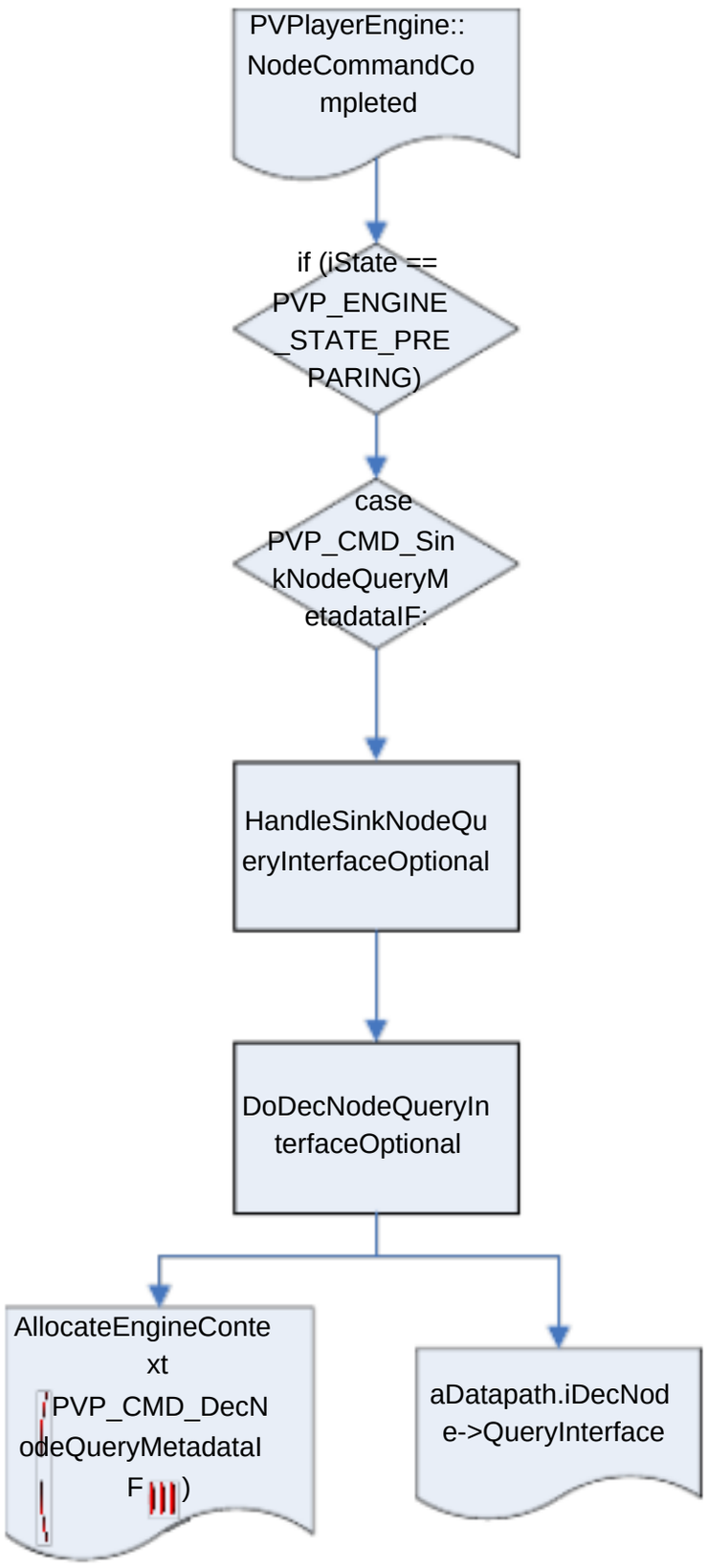
于是 DoPrepare 进入 prepare 里面的最后一个状态；

4.4.4 PVP_ENGINE_STATE_TRACK_SELECTION_3_DONE 的处理

先看看示意图：

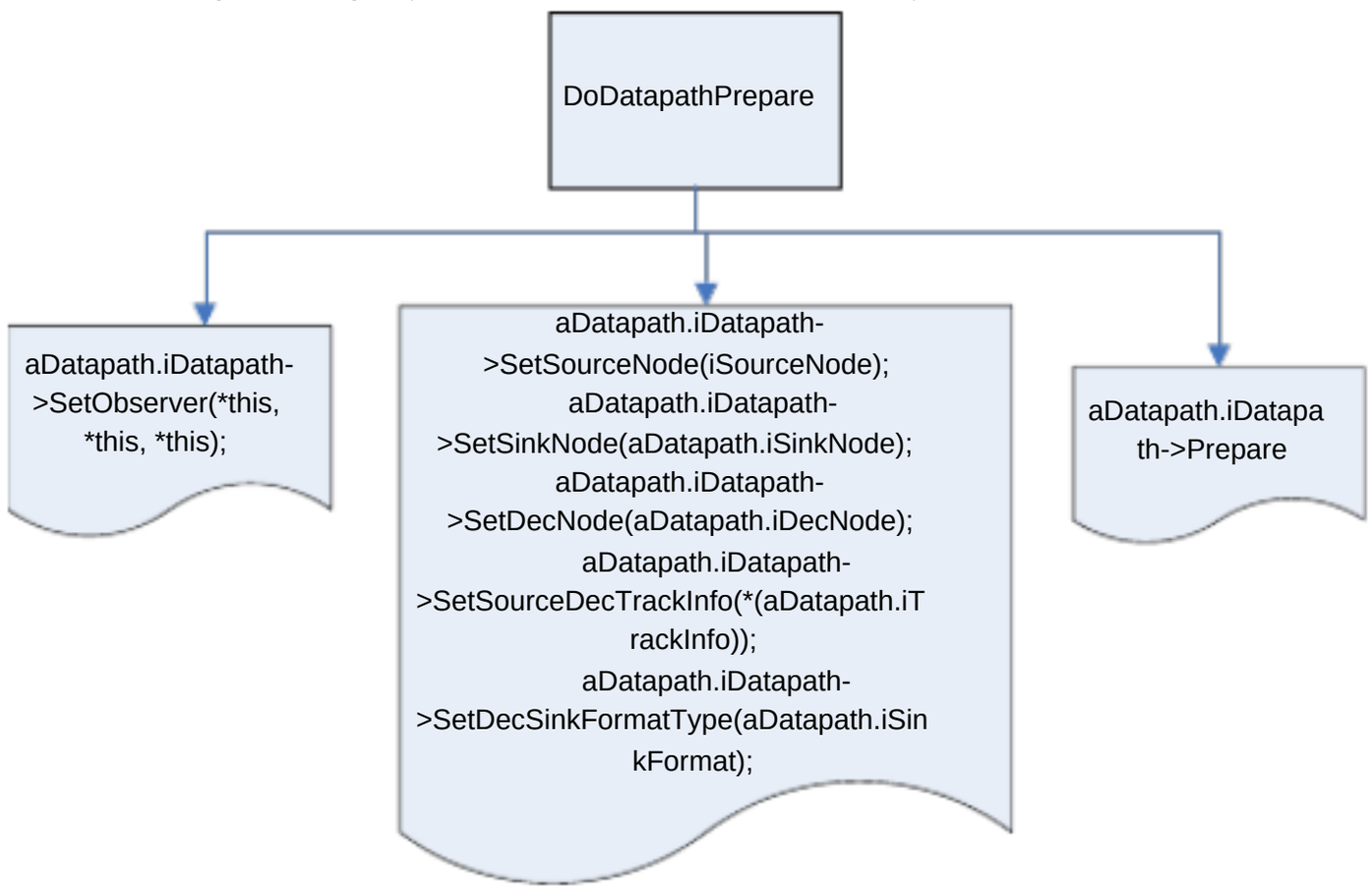


1, 主要就是调用 srccode的Prepare处理；
2, 取得 iSinkNodePVInterfaceMetadataExt控制对象；
等上面这些做完后，逻辑如下：



也就是说这时候需要取得的是 decnode 的扩展接口 aDatapath.iDecNodePVInterfaceMetadataExt;
并触发 NodeCommandCompleted 进入下一个状态，逻辑如下：

到了 DoDatapathPrepare，我们来看看这个函数的逻辑，如下：



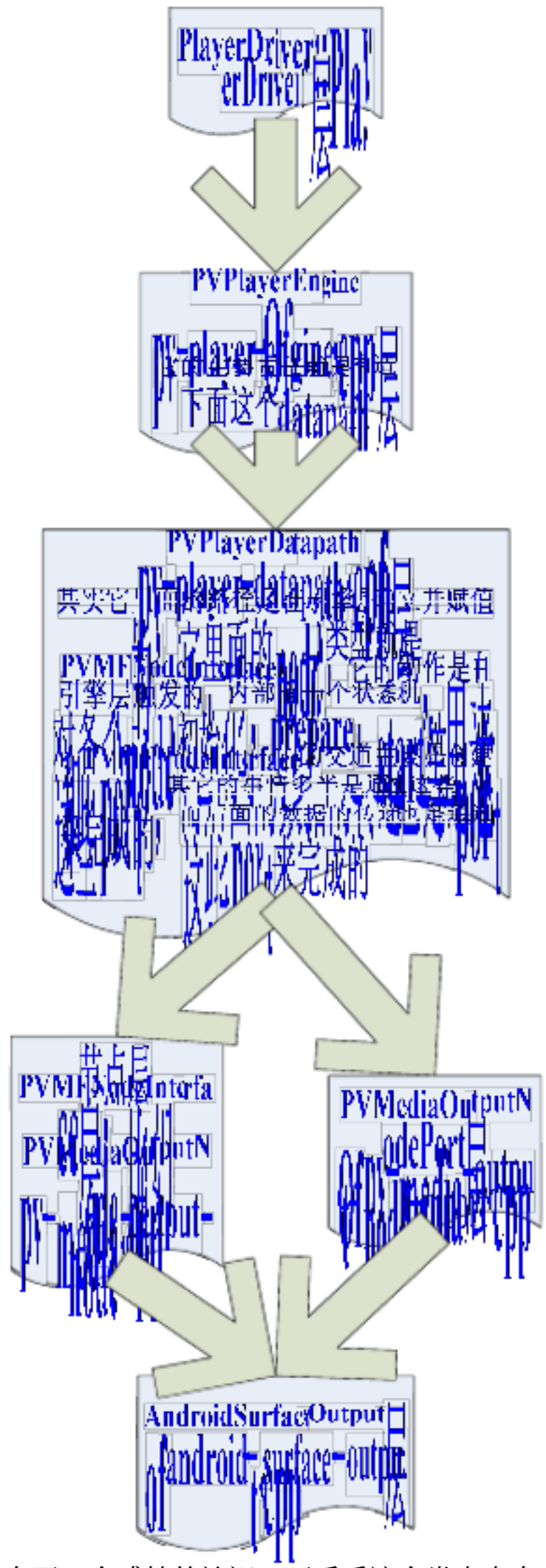
先是给 aDatapath.iDatapath设置观察者，这样 datapath可以通过调用观察者的回调函数来通知引擎层状态的更新，事件的传递等，中间那一步就是给 iDatapath设置源节点，解码节点， sink 节点，还有 track信息，已经解码输出的格式等，最后调用 iDatapath的Prepare函数，并在 NodeCommandCompleted里面等待这个命令的完成；

唉，终于进入了 dataPath的处理范围了，下面需要详细分析的是 datapath的对于 prepare的处理逻

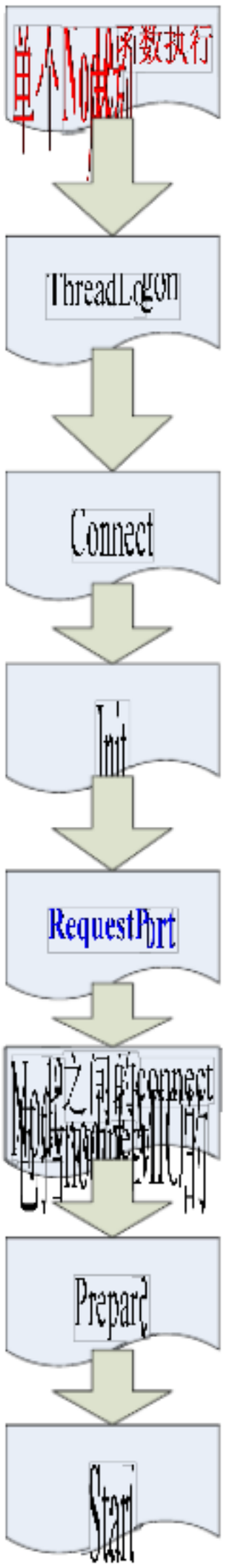
辑，有点复杂，请慢慢看：

4.5 PVPlayerDatapath 层的 prepare 相关处理逻辑

在进入到它的 prepare 处理之前，我们先温习一下它本身在整个逻辑控制流中的位置，如下图：



有了一个感性的认识，再看看这个类它本身所起的作用，它就是一个个连接这些 node，并控制这些 node的状态变化的， 它的处理通常都是调用其内部拥有的 src，dec，sink 等node的对应函数来处理的，而对于一般的 node它的内部调用函数序列大约都是这样的，请看：



这里的箭头不表示调用关系，只表示先后关系。
好了，开始进入正题 ！

PVPlayerDatapath对于 Prepare的处理分成以下几个阶段：
A） 进入PREPARE_INIT 状态以前的处理；
B） PREPARE_INIT 状态的逻辑；
C） PREPARE_REQPORT状态的逻辑；
D） PREPARE_CONNECT 状态的逻辑；
E） PREPARE_PREPARE状态的逻辑；

下面分别来讨论；

4.5.1 进入 PREPARE_INIT 状态以前的处理

如下图所示：

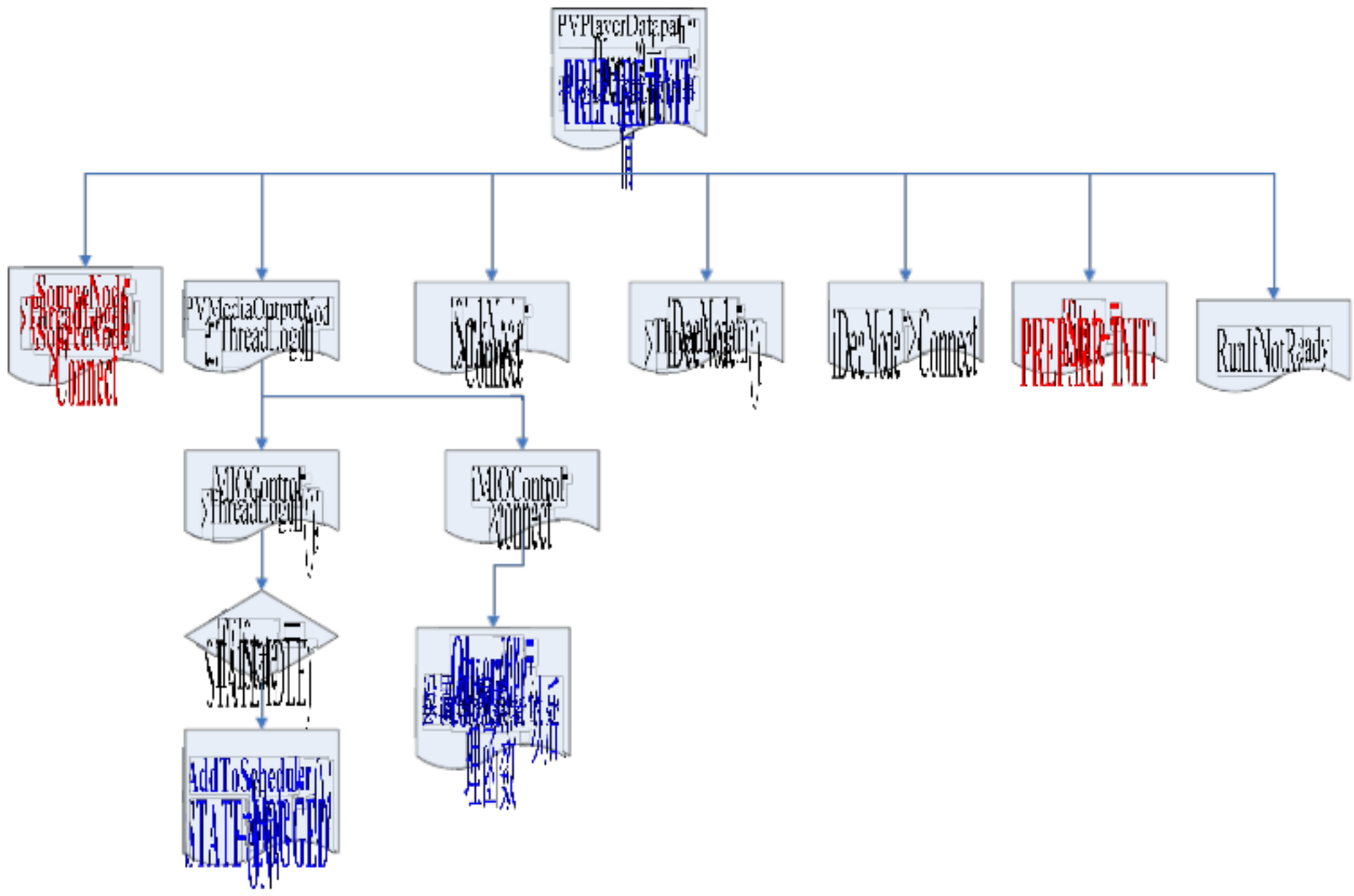


图 4.5.1.0

为了下面讨论 MIO 集成的需要，我只稍微描述了一下对于输出节点的 ThreadLogon的逻辑，其它的对于 src，decode的Logon和connect各不相同，我也没去分析；

这里对于输出节点的 logon，实际上调用了其内部拥有的那个 iMIOControl 的ThreadLogon和Connect函数，MIO 的threadLogon里面只是设置了一个标志位，而在 MIO 的Connect函数里面其实只是设置了一个观察者，这样当 MIO 发生什么事情的时候，可以有一个人知道；

也许你想问这个 iMIOControl 在什么时候建立的，那好吧，我们回头看看它的建立的逻辑，如下：

图 4.5.1.1

这也就是先前的设置 video输出的逻辑，在这个 Create的过程中，这个 iMIOControl 就建立了，对于 marvel的处理，这个 iMIOControl 其实就是 AndroidSurfaceOutput类型的指针，用于代表 MIO 对上提供的控制接口；

或许你需要回头看看 AndroidSurfaceOutput的类结构更清楚， 在3.11节有讲 ；

图4.5.1.0里面有这么一句话：

iState = PREPARE_INIT;

状态成了 PREPARE_INIT，通过

RunIfNotReady，于是启动了状态机的下一个状态处理，请看下一节；

4.5.2 PREPARE_INIT 状态的逻辑

这里的主要目的是调用 sink和dec node的init 函数， src节点的 init 处理在引擎层已经做了 ,逻辑图如下：

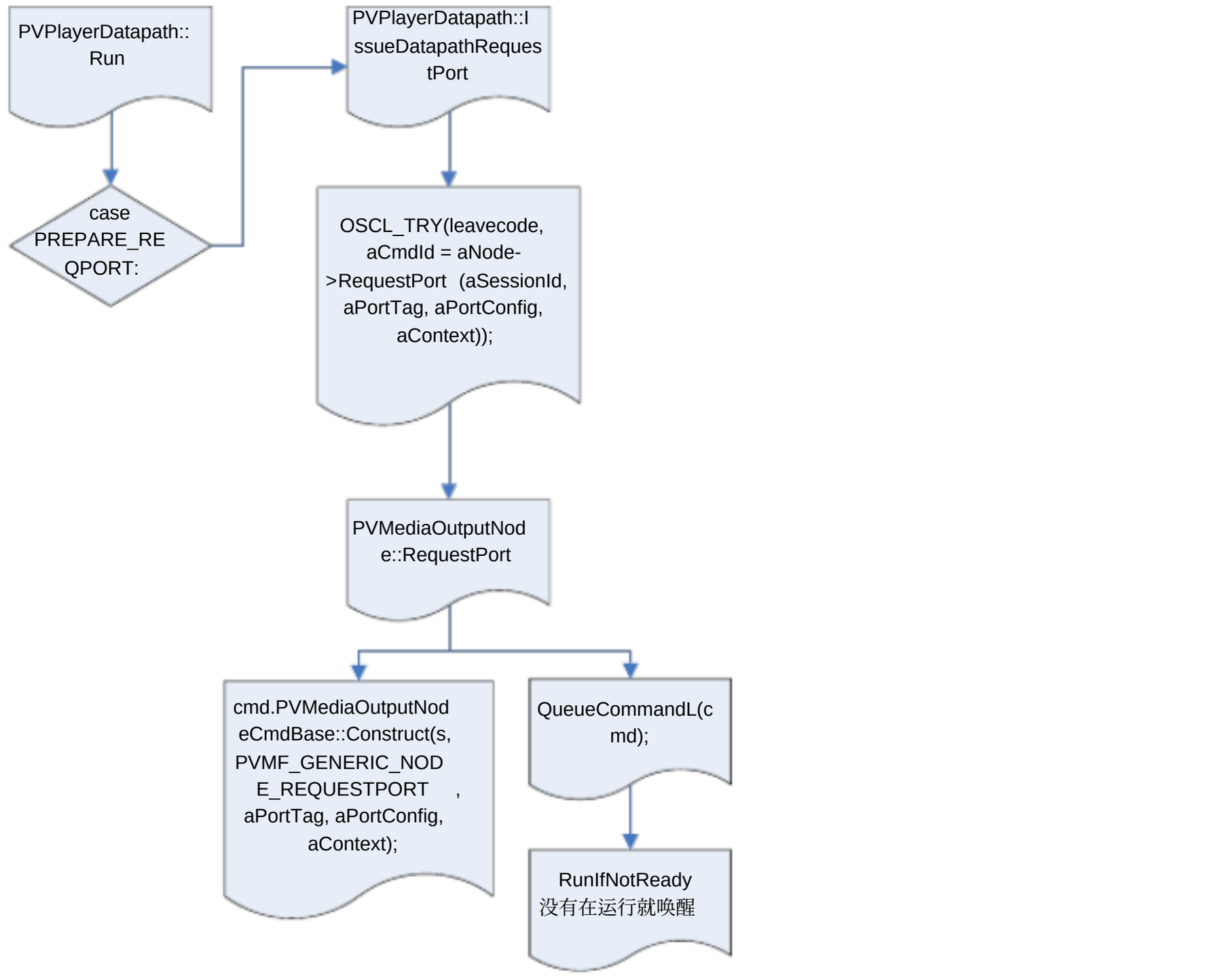
等这两个初始化完成后，在 PVPlayerDatapath::NodeCommandCompleted里面会调用：

```
iState = PREPARE_REQPORT;
 RunIfNotReady();
```

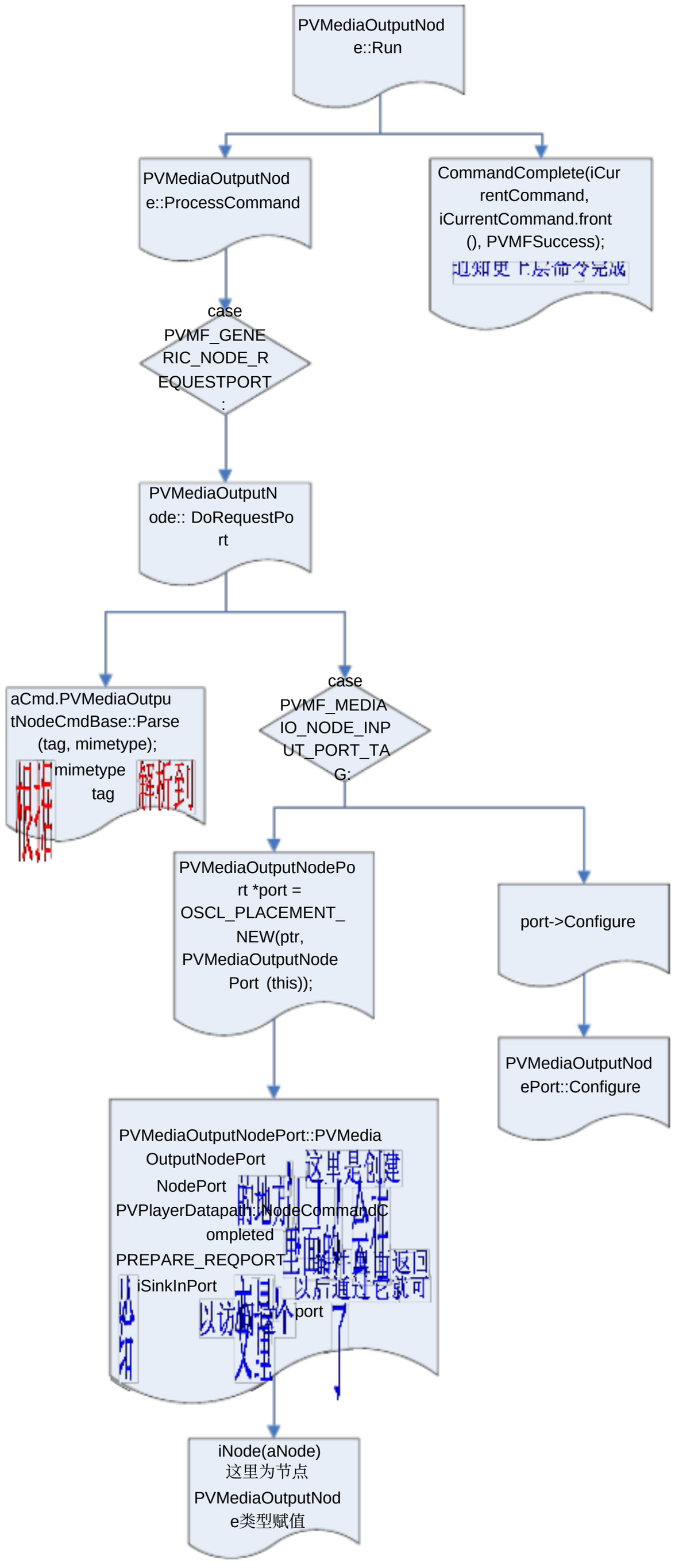
于是进入了下一个状态！

4.5.3 PREPARE_REQPORT 状态的逻辑

这里的处理非常重要，因为需要请求 port，而这些 port就是用来传递数据的，或者说用来描述节点之间的纽带关系的，它的逻辑如下：



它最后的处理，如我们前面分析的那样都会到节点层的，我这里只讨论对于输出节点的 port请求逻辑，其它的我们一般都用不着关心，它的逻辑如下：


PVMediaOutputNode::Run
PVMediaOutputNode::ProcessCommand
CommandComplete(iCurrentCommand, iCurrentCommand.front(), PVMFSuccess);
通知更上层命令完成
case PVMF_GENERIC_NODE_REQUESTPORT:
PVMediaOutputNode:: DoRequestPort
aCmd.PVMediaOutputNodeCmdBase::Parse(tag, mimetype);
mimetype
tag
case PVMF_MEDIAIO_NODE_INPUT_PORT_TAG:
PVMediaOutputNodePort *port = OSCL_PLACEMENT_NEW(ptr, PVMediaOutputNodePort (this));
port->Configure
PVMediaOutputNodePort::Configure
PVMediaOutputNodePort::PVMediaOutputNodePort
这里是创建
NodePort
PVPlayerDatapath::NodeCommandCompleted
PREPARE_REQPORT
iSinkInPort
port
iNode(aNode)
这里为节点
PVMediaOutputNode类型赋值

在节点层命令执行完毕后，PVPlayerDatapath::NodeCommandCompleted会被调用，于是取得 node 层创建的 port，如下：

```
iSourceOutPort  = (PVMFPortInterface*)(aResponse.GetEventData());
iSinkInPort   = (PVMFPortInterface*)(aResponse.GetEventData());
iDecInPort  = (PVMFPortInterface*)(aResponse.GetEventData());
iDecOutPort  = (PVMFPortInterface*)(aResponse.GetEventData());
```

这些port 非常重要，以后的数据流传递就全靠它们了；

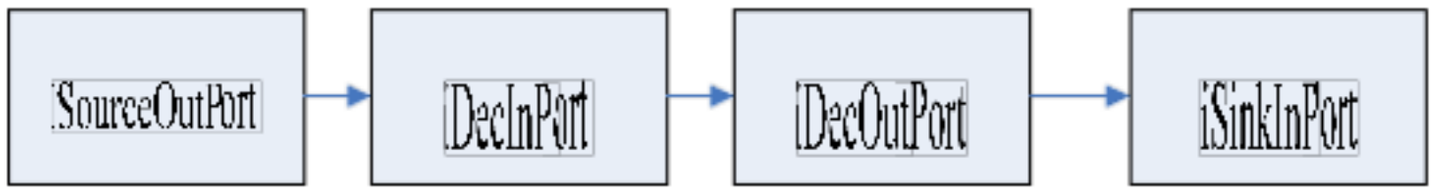
最后设置状态：

```
  iState = PREPARE_CONNECT;
  RunIfNotReady();
```

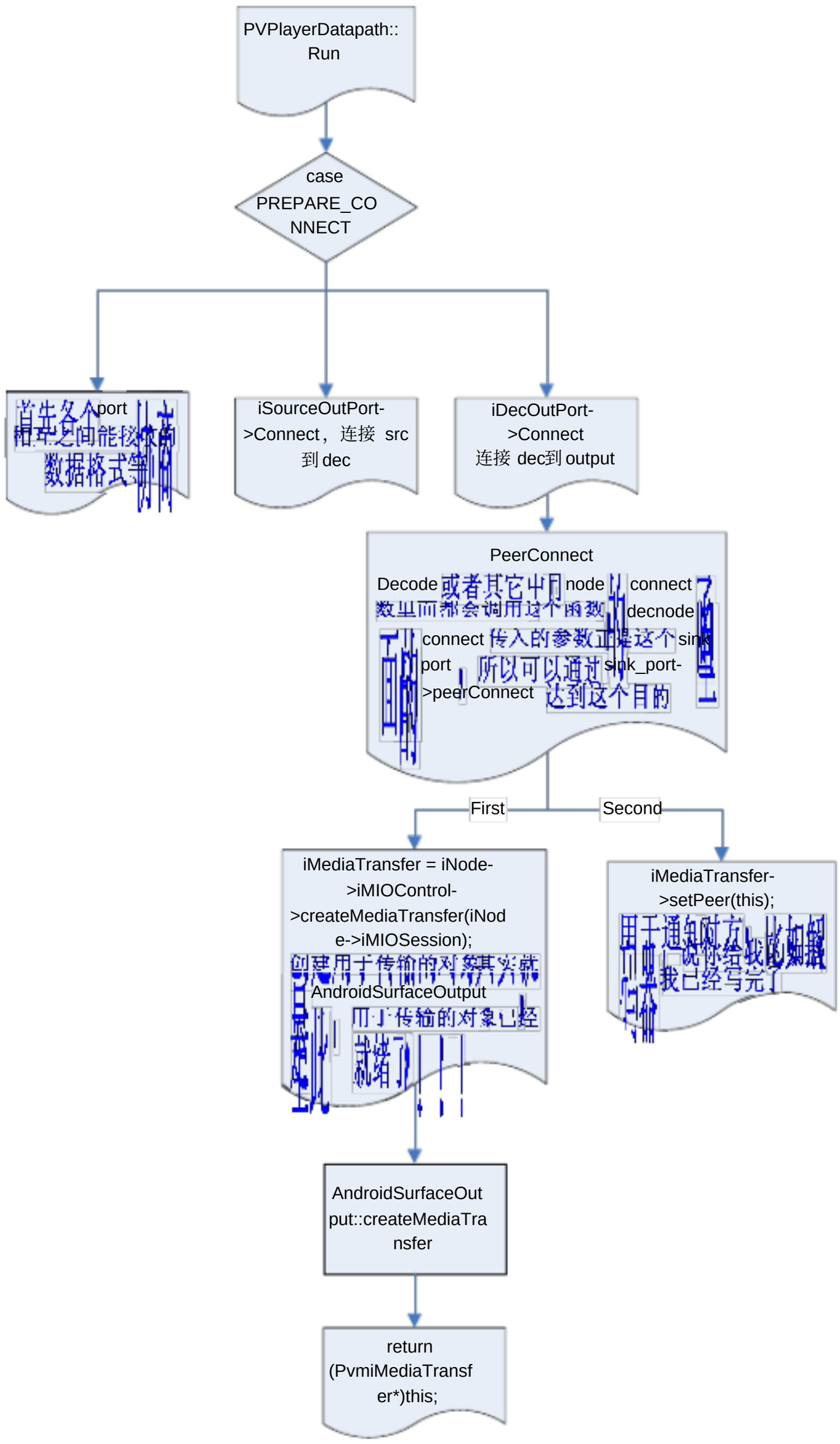
于是进入了下一个状态；

4.5.4 PREPARE_CONNECT 状态的逻辑

上面那步已经取得了各个节点的 port，但是并没有对其进行配置。在这个状态做的事情，主要就是协商各个 port所能接受的格式，参数等，然后相互连接起来，像这样：



当然这里只是个示意，不同的格式，中间的节点也不一样，而且，这里的 source不一定就是文件，有可能是parse等，这块需要以后分析，下面来看看它们之间的连接过程，如下图所示：



这里对于我们所关心的 MIO 的传输问题，有一个很重要的数据结构，那就是 iMediaTransfer, 这就是用来描述传输的对象，以后的传输的操作都是由它来控制，本质上就是 AndroidSurfaceOutput;

在连接完成后，就在这个 run函数里面，调用了：

iState = PREPARE_PREPARE;

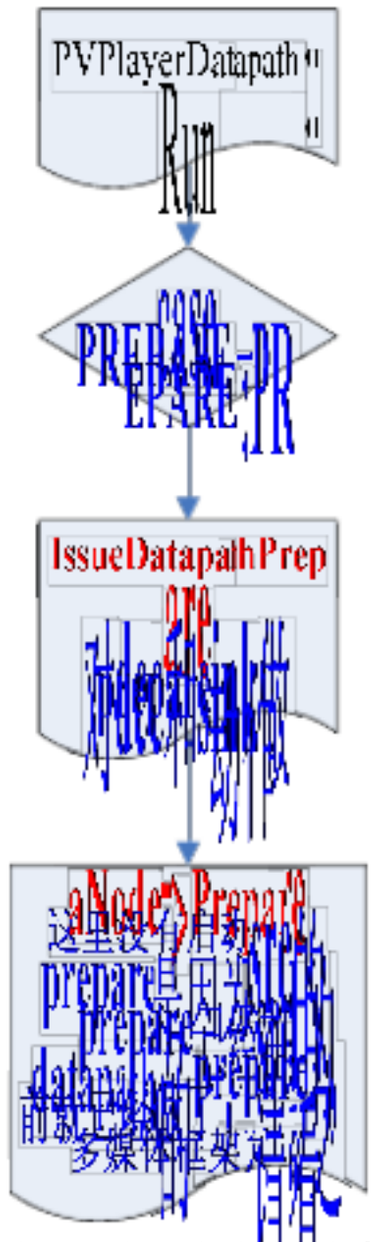
RunIfNotReady();

（这和多数情况通过 NodeCommandCompleted来改变状态的情况不一样）；

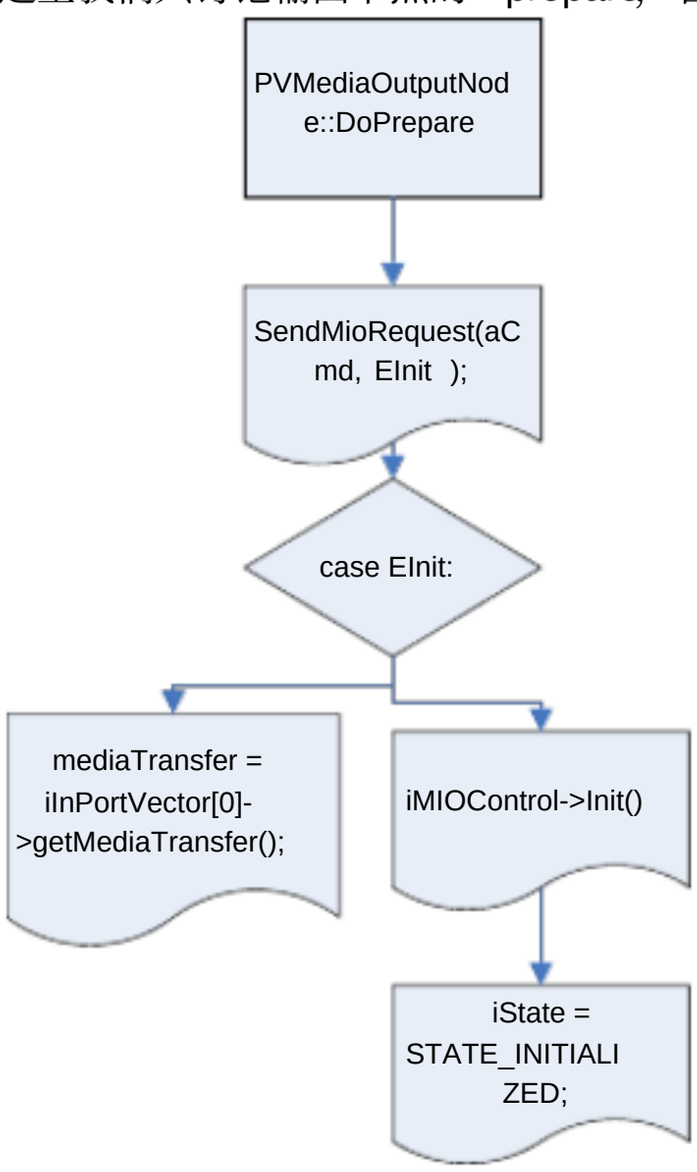
于是进入了下一个状态；

4.5.5 PREPARE_PREPARE 的逻辑

这个状态需要做的就是调用各个节点的 prepare函数，真正进入 prepare状态了，如下图所示：



这里我们只讨论输出节点的 prepare，看看它的逻辑，也比较简单：



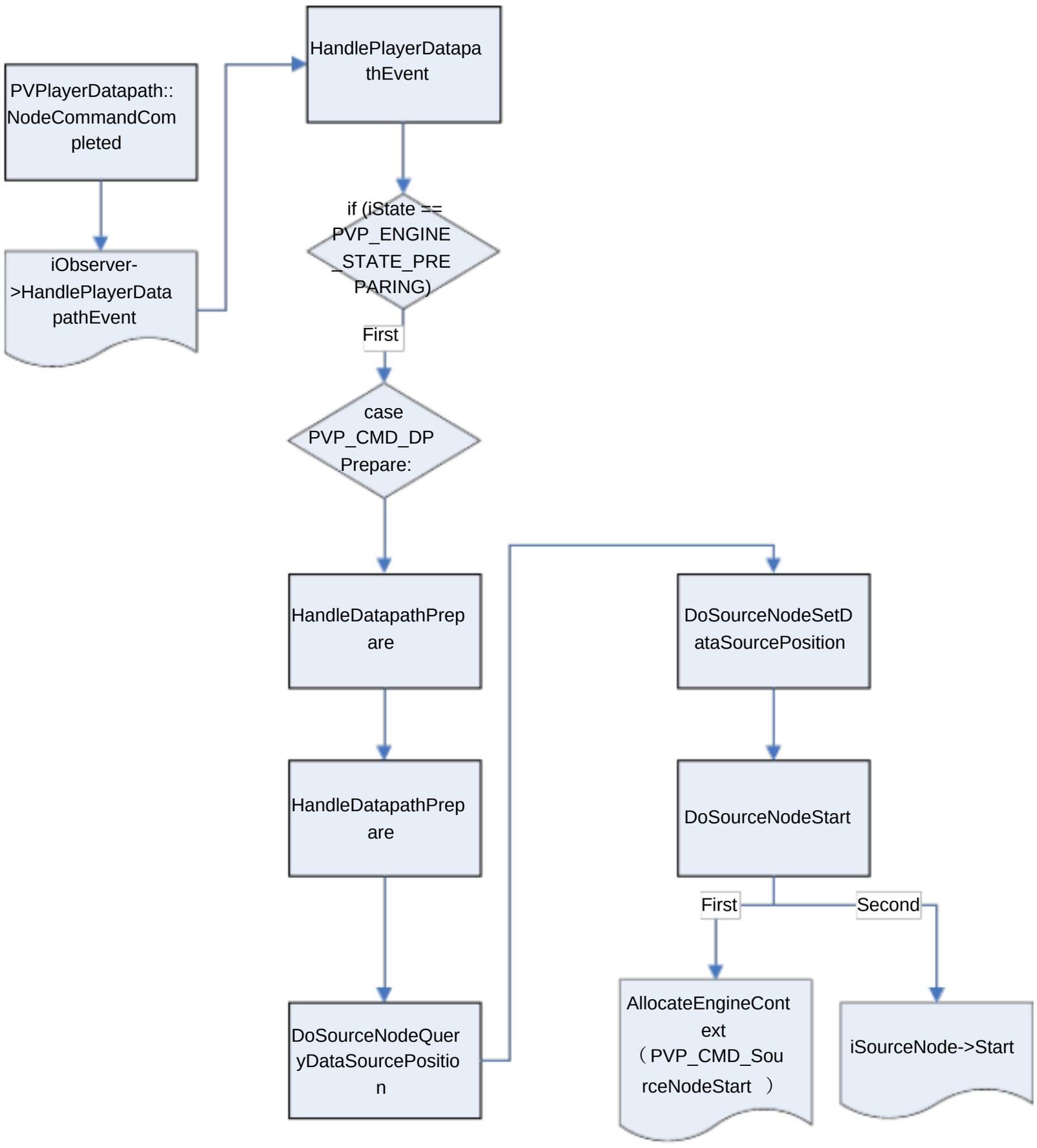
本质上就是调用输出节点的 MIO 的init函数，在那里，设置一个状态，就开始宣称命令已经完成，于是通知上层这个消息，

iState = PREPARED;
iObserver->HandlePlayerDatapathEvent(0, PVMFSuccess, iContext);

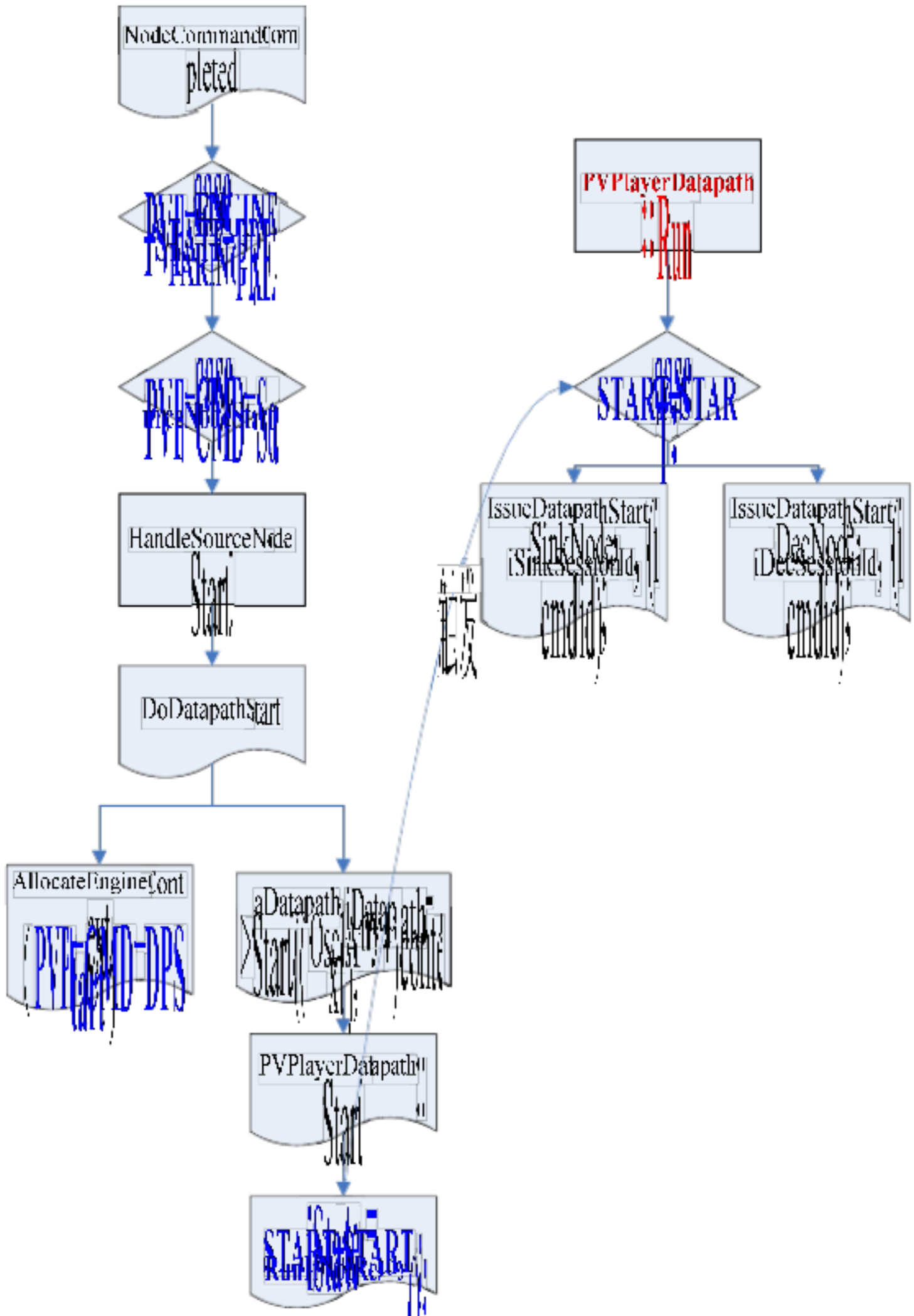
注意这里的 iObserver 代表的是引擎层，于是引擎层，知道 datapatch 对于 prepare 的处理已经完毕了；不过，伟大的 prepare 状态还没有结束，只是现在需要回到引擎层了；

4.6 Prepare 的收官之战

上回说到 iObserver->HandlePlayerDatapathEvent(0, PVMFSuccess, iContext);于是控制权又回到了引擎层，在这里还需要折腾一番，如下图所示：



为了能够看清楚里面的字母，而且不要占用太多的体积，所以我的图都画得比较丑陋，这个图想表达的意思就是说：引擎层在收到 datapath层的完成消息后会启动 src node的start函数，换句话说，这里提前让 src进入了start状态，它的逻辑不简单，所以以后再分析吧，这里重点在于控制逻辑的分析，等 src的start函数返回后，同样会触发 NodeCommandCompleted函数的执行，在这里还要触发 datapath的start，逻辑如下：

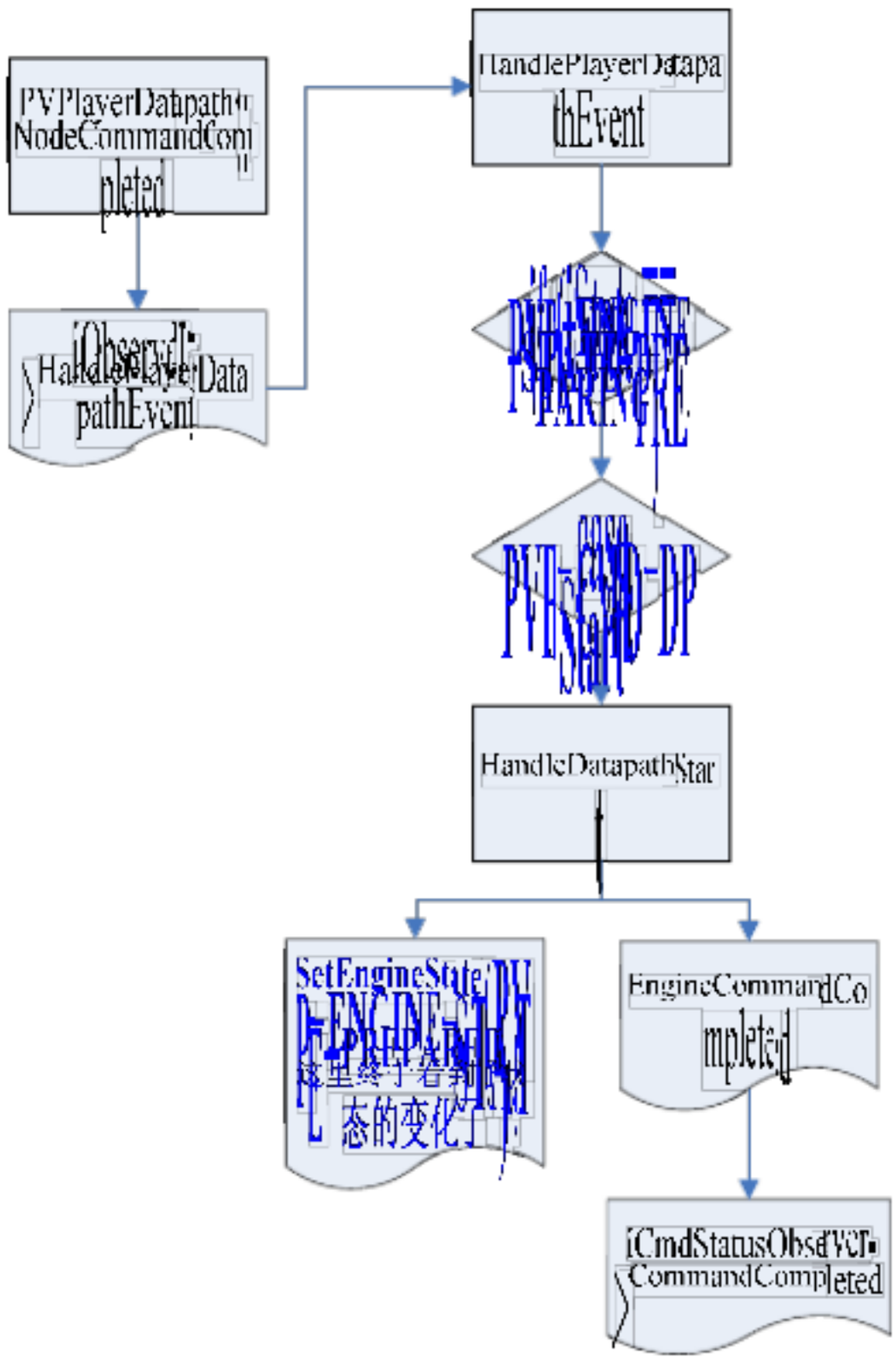


Datapath的处理也就是调用 node层的start函数，包括 sink以及可能的（如果有） decode的start函数；
对于 output节点，它说做的就是调用 MIO 的start函数，在 MIO 层，设置状态位：

iState = STATE_STARTED;

并通知上层；

其处理逻辑如下：



可以看到引擎层的状态终于转变为 PVP_ENGINE_STATE_PREPARED 了；

最后是通知更上层，也就是 playerdriver层；

Playerdriver层在CommandCompleted函数里面的处理逻辑如下：

```
case PlayerCommand::PLAYER_PREPARE:
                    mPrepareDone = true;
                // If we are streaming from the network, we
                // have to wait until the first PVMFInfoDataReady
                // is sent to notify the user that it is okay to
                // begin playback.  If it is a local file, just
                // send it now at the completion of Prepare().
                if ((mDownloadContextData == NULL) || mDataReadyReceived) {
                    mPvPlayer->sendEvent(MEDIA_PREPARED);
                }
                break;
```

现在通过 sendEvent通知 mediaplayerservice层；

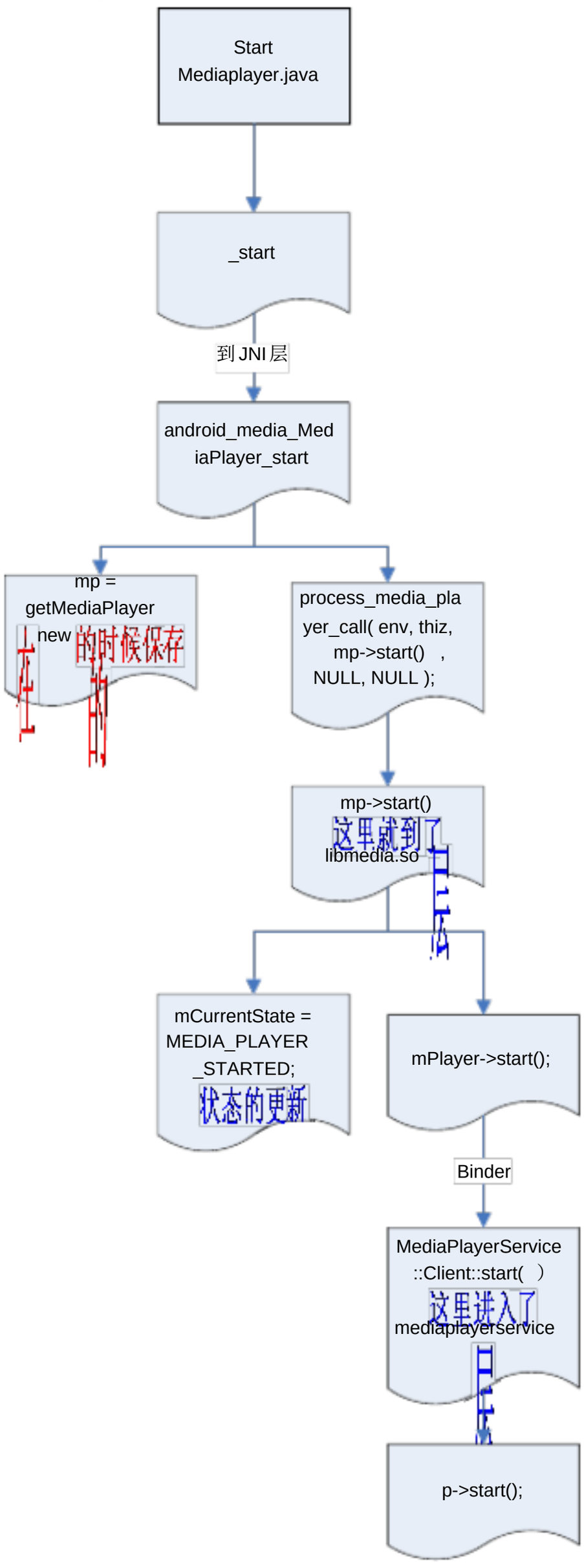
如此层层通知状态的更新，

好吧，我们姑且认为现在所有层的状态都已经进入 prepare了，接下来就是启动数据流了，也就是进入 start状态；

4.7 Start　　流程的分析

4.7.1 Android　　本身架构对 start 的处理

在 PVPlayer层以上，都是简单的二传手，所以我就不想讨论了，只是简单的画了个示意图，如下：

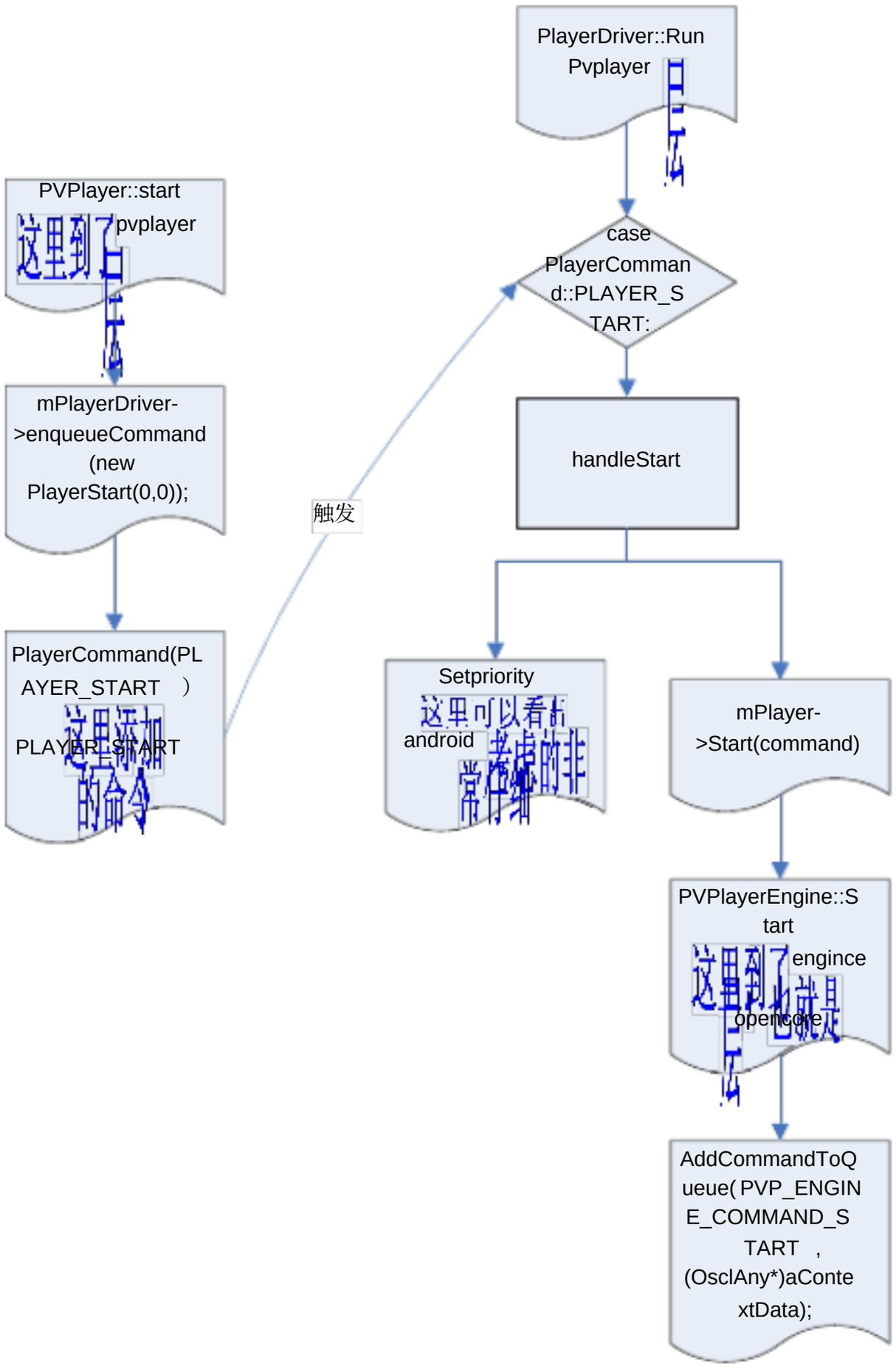


它的流程和 prepare本身差不了多少，都是一层一层的调用，中间还经过了　　binder才到service端，最后

调用 PVPlayer的start来实现；
OK，下面看看 PlayerDriver的处理

4.7.2 PlayerDriver 层的start 流程

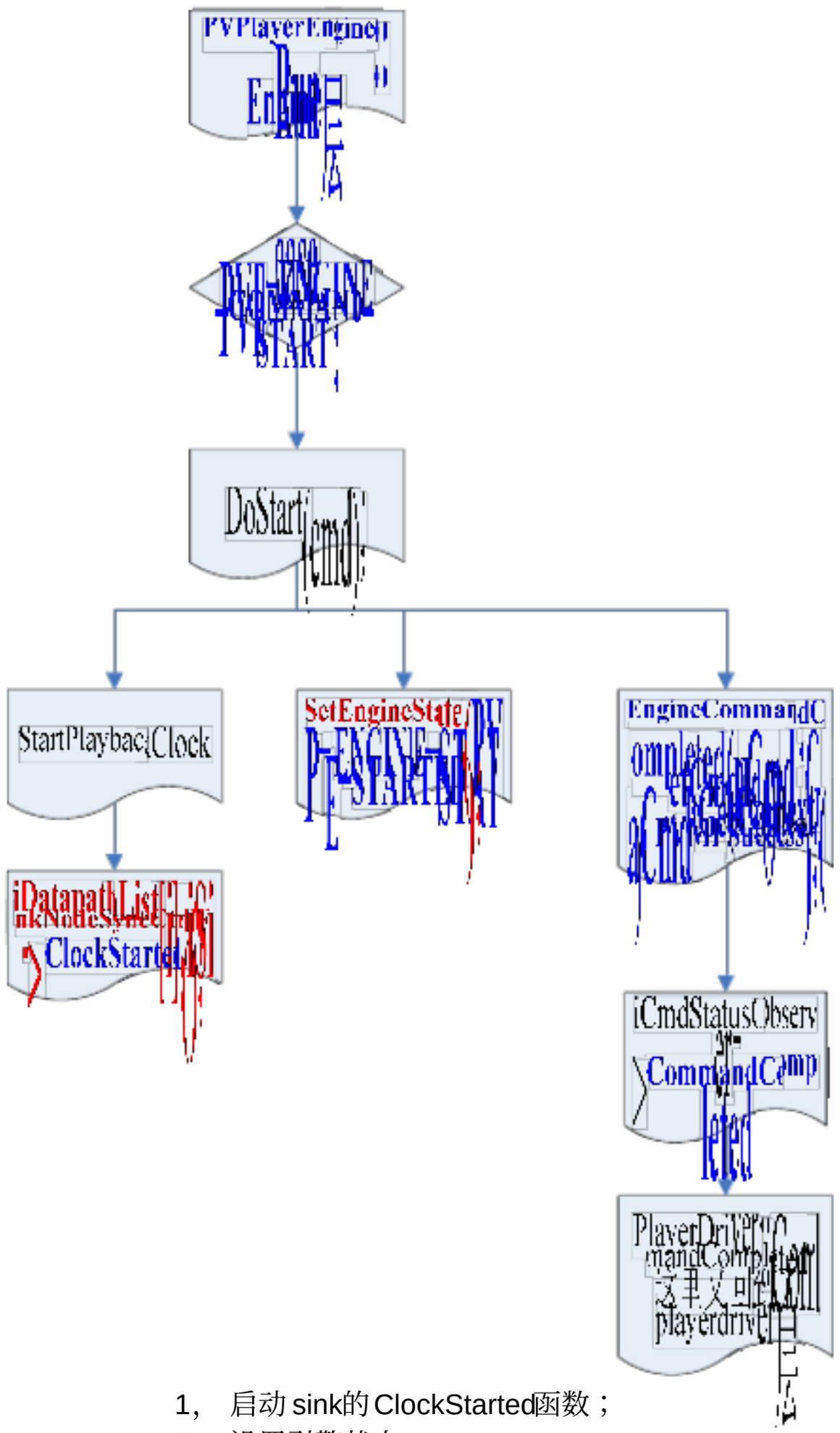
如下图所示：



可以看出，基本上工作都是通过引擎层的 start来实现的，不过这里有设置优先级；

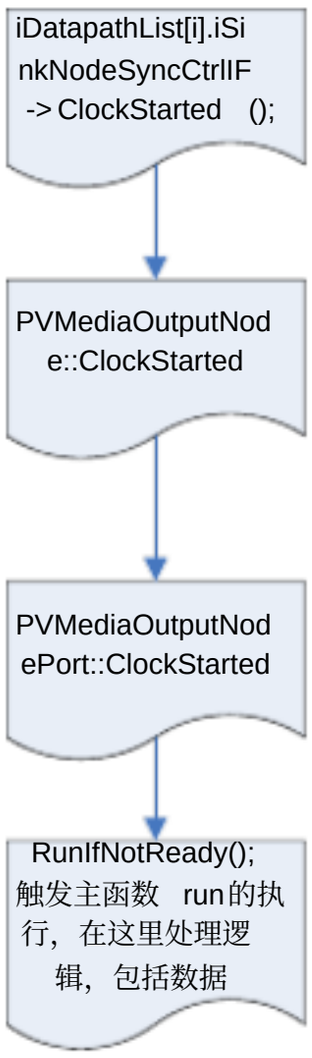
4.7.3 引擎层的 start处理

这里就比较重要了，如下：



1， 启动 sink的 ClockStarted函数；
2， 设置引擎状态；
3， 通知 PlayerDriver层状态更新；

其中最重要的，我想应该是 sink的ClockStarted函数来，猜测应该是这里开启了最后的数据流的开关，在进入这步以前，我们可以看出，对于 src，decode的开关都已经启动，但是数据流依然不会动，因为最后的开关由 sink端来控制，简单说，如果一条水管由三个开关控制，分别较 src，dec，sink，那么第一步，需要打开 src 的开关，其次是 dec，而最后才是 sink，前面两步，在 prepare阶段就已经做了，这里就是最后一步，我们来看看逻辑，如下：



可以看到，终于，在最后的输出节点的输入 port里面开始了数据的处理，在这里已经等候多时的数据开始被发送，发送的逻辑请看下一节；

4.8 数据 的流动

对于PV的架构，数据的传递分成两种模式，

A） tunnel模式；

B） 非tunnel模式；

3.10节的那个图有一点介绍，可以回去温习一下，下面具体讲；

4.8.1 Component 的初始化

在进行数据传输之前 component都需要进行一定的初始化，比如缓冲区的分配，格式的设置等等，我们前面的讨论都在 node层及以上讨论， 没有下到 component，现在我们来看看一般的 component的初始化都包括那些，如下图所示：

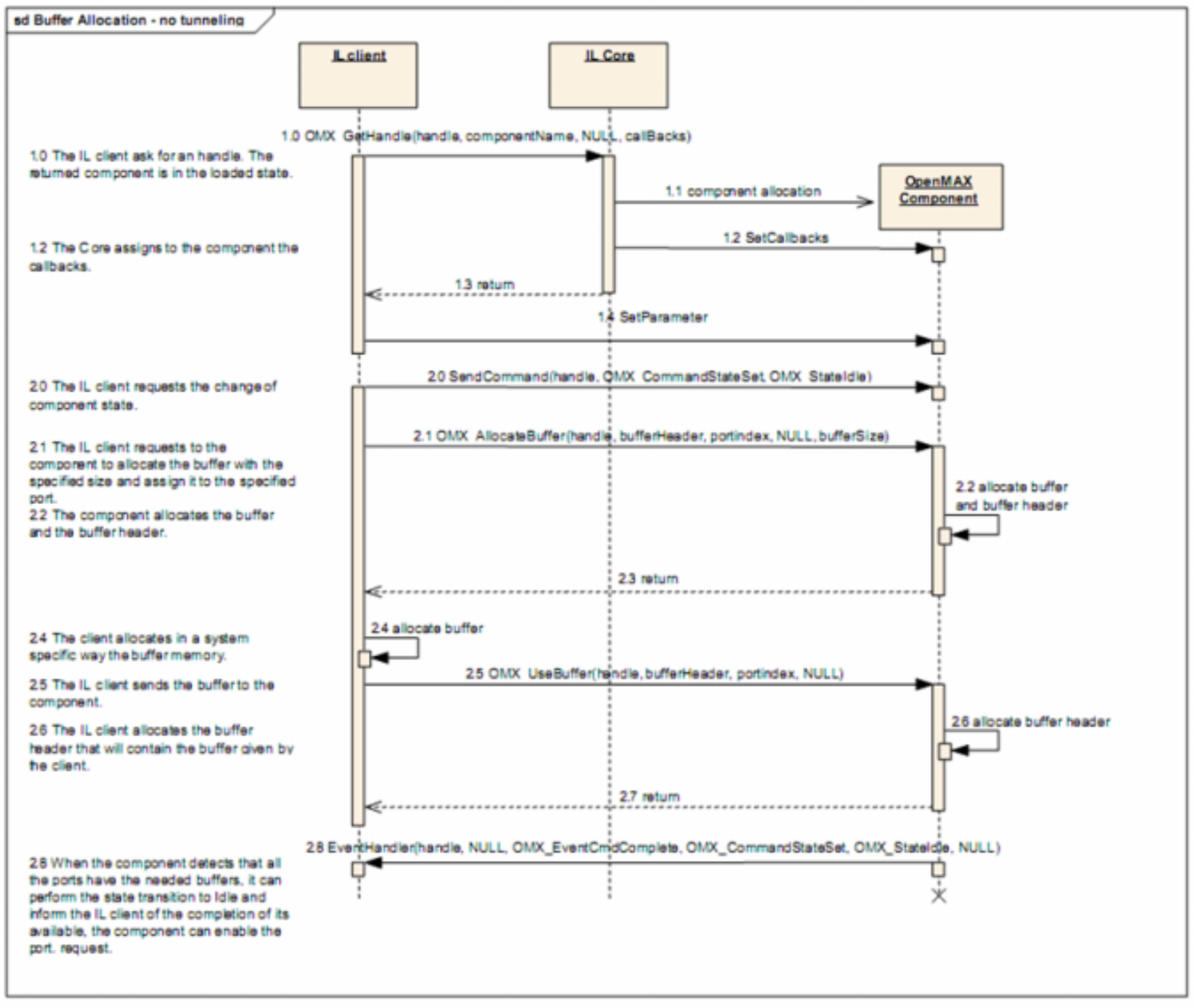


Figure 3-5. Component initialization

至于这些事件发生在那个阶段，那么可以认为就发生在各个 Node进行 Prepare的时候， component所属有的那个 node会调用这些以“ OMX_“开头的命令对各自的 component进行初始化，命令序列就像上面那样；

简单描述如下：

1, 在node的prepare里面，比如PVMFStatus PVMFOMXBaseDecNode::DoPrepare，会先根据 roles 取得component的名字，这里的role就是这样的字符串：audio_decoder.aac，video_decoder.mpeg4等等；当然对于上层来说只需要识别出它的 mime 字符串就行了，比如像这样的： #define PVMF_MIME_H264_VIDEO "video/H264" ，在解码 node里面会自动根据这个字符串识别为： video_decoder.avc，利用这个 role去查询注册表，看能够胜任这种角色的component有哪些？这里返回的都是以字符串描述的，它分两步，一步是取得每种 role对应的component的个数，第二步调用同样的函数 OMX_MasterGetComponentsOfRole来把这些字符串保存起来；

2, OMX_MasterConfigParser通过这个函数配置参数，这个函数又通过一个类似于注册表的代理的东西，在里面根据传入的 component的名字找到对应的 pInterface，通过接口提供的GetpOMXConfigParser实现，而marvel的系统提供了自己的实现， 是通过 pOMXConfigParser = (tpOMXConfigParser)dlsym(ipConfigHandle, "MRVLOMXConfigParser "); 方式取得的，也就是说 marvel自己提供了一个库来实现了这个函数，在 marvel_omx_config_parser.cpp 里面；函数太长，就不贴出来了，里面先区分是 audio_decoder还是 video_decoder，再区分是什么类型的 audio或者什么类型的 video，比如，对于

```
if (0 == oscl_strcmp(pInputs->cComponentRole, (OMX_STRING) "video_decoder.wmv" ))
        {
```

aInputs.iMimeType = PVMF_MIME_WMV;

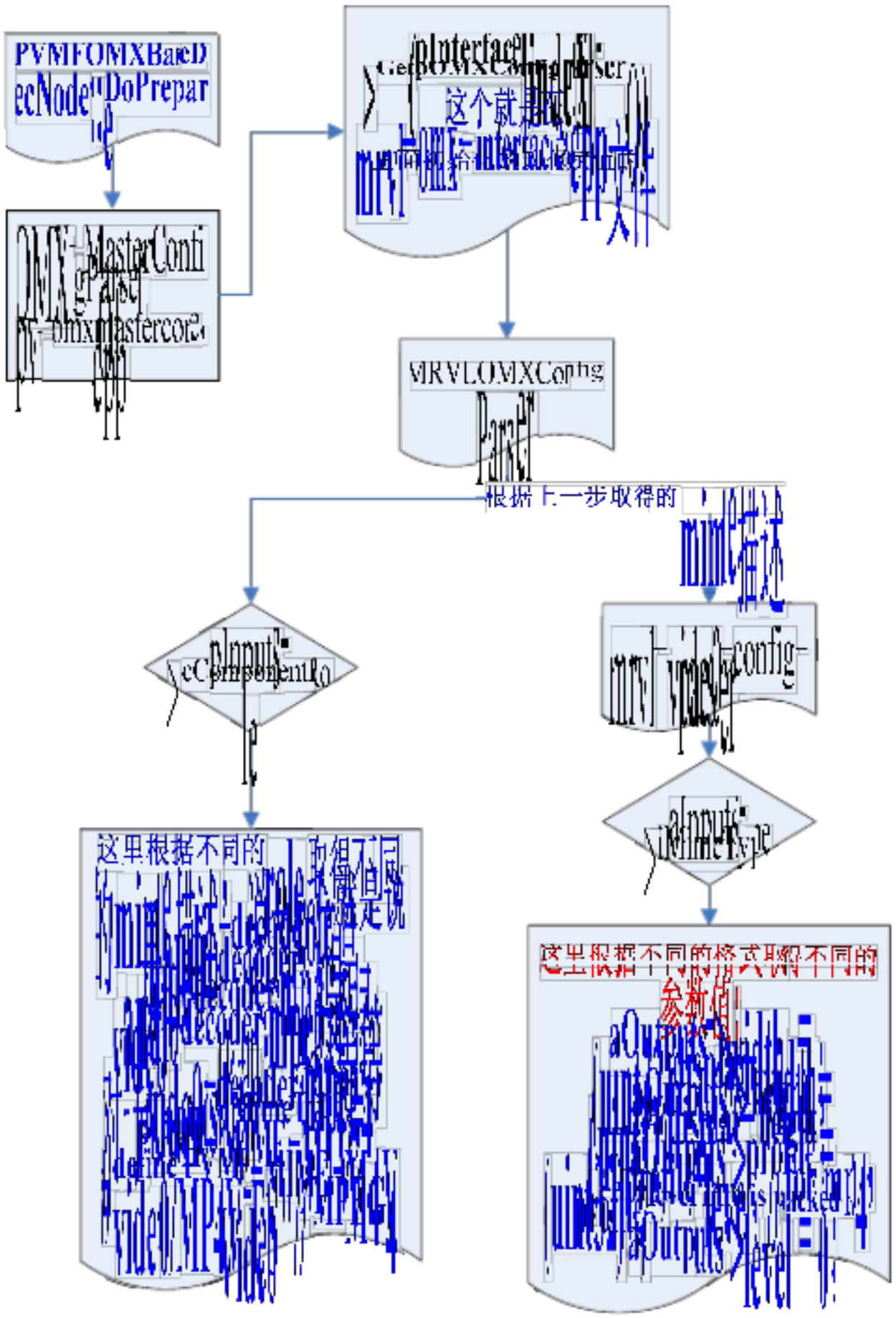
}

取得了 Mime 类型再调用 OSCL_EXPORT_REF int16 mrvl_video_config_parser(pvVideoConfigParserInputs *aInputs, pvVideoConfigParserOutputs *aOutputs)来处理，在这里面返回一些参数，比如宽度，高度，profile，level等；

3, 通过 OMX_MasterGetHandle(&iOMXDecoder, (OMX_STRING) inputParameters.cComponentName, (OMX_PTR) this, (OMX_CALLBACKTYPE *) & iCallbacks);创建 component的handle；这里还有对于 marvel的特殊处理，如果是 marvel的 component就设置 bIsMrvlOmxComp = true;

4, 做一些例行检查，缓冲区分配等等，看上面的图就行了，我已经不关心了；

如果一定要用图来描述的话，如下：



4.8.2 Tunnel 模式的数据流动

有了上面的了解，我们先来看看正题上，这个 tunnel模式到底是什么意思，如下图所示：

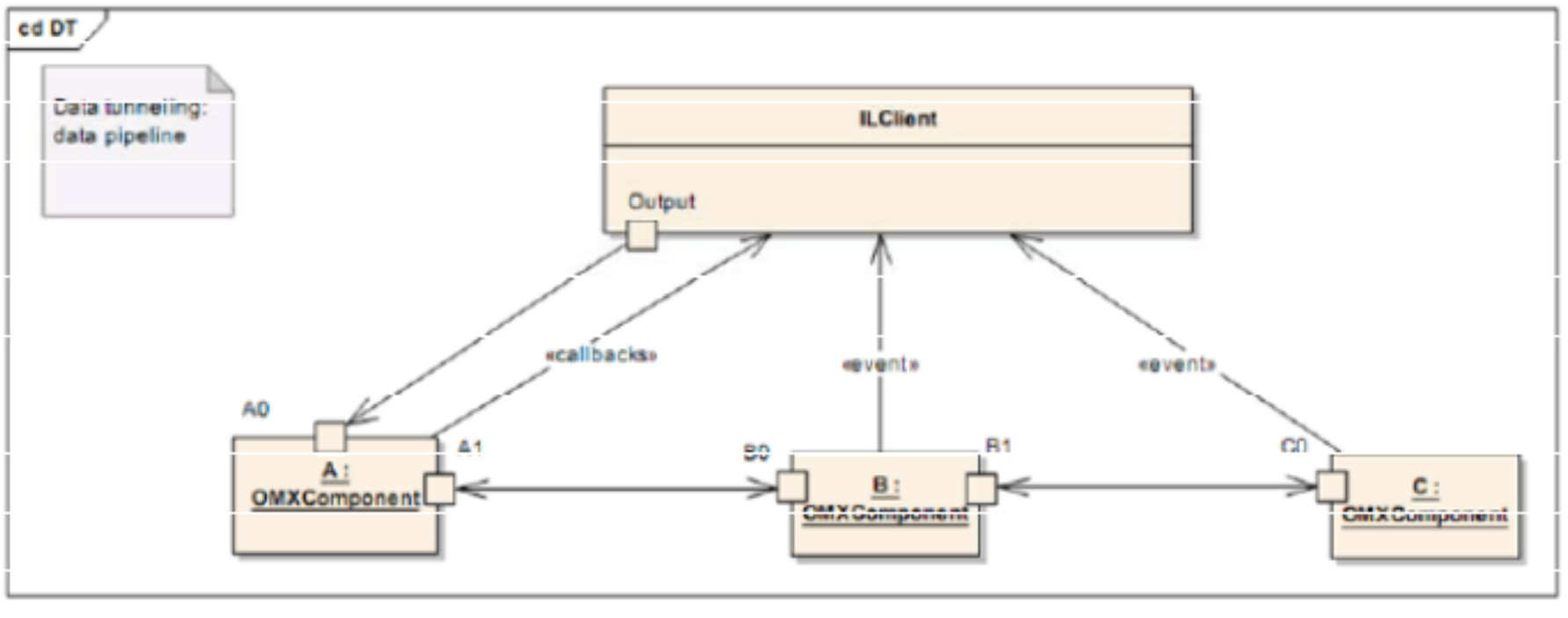


Figure 3-6. Example of Data Tunneling Among OpenMAX IL Components

数据从 A1直接到 B0，再从 B1到C0，所谓 0，1就是指输入和输出 port；

再来看看调用逻辑，如下：

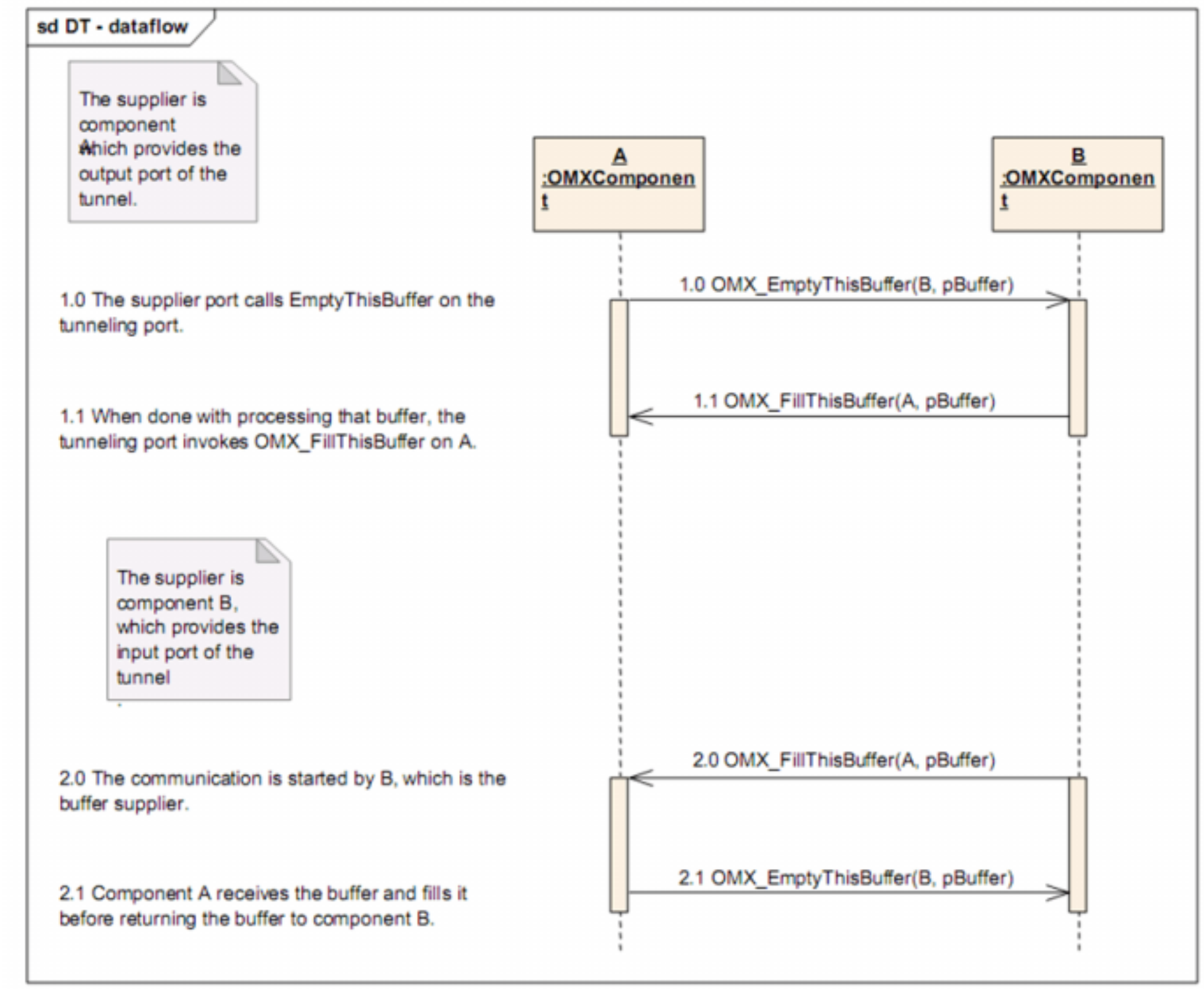
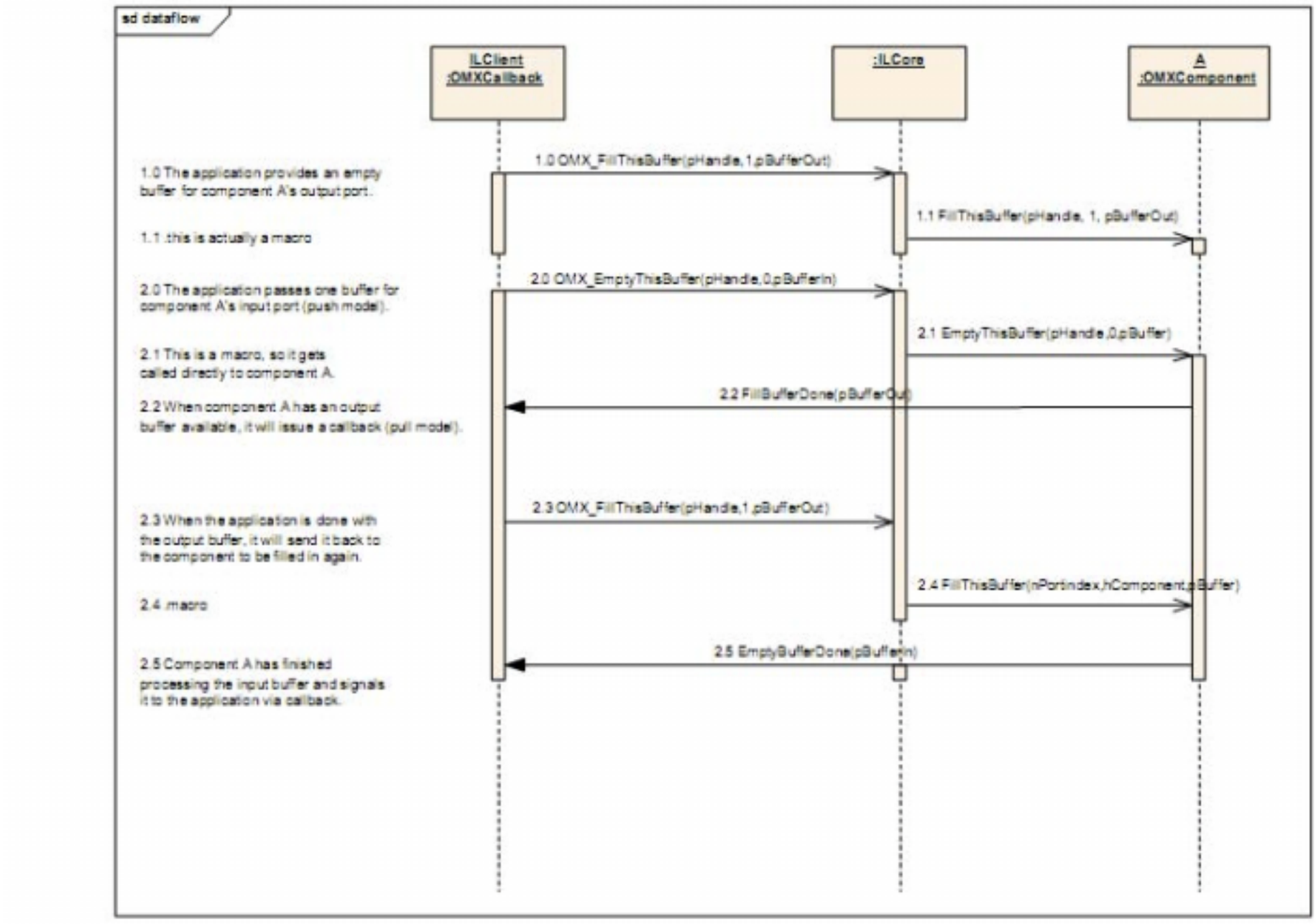


Figure 3-12. Data Flow Between Tunneled Components

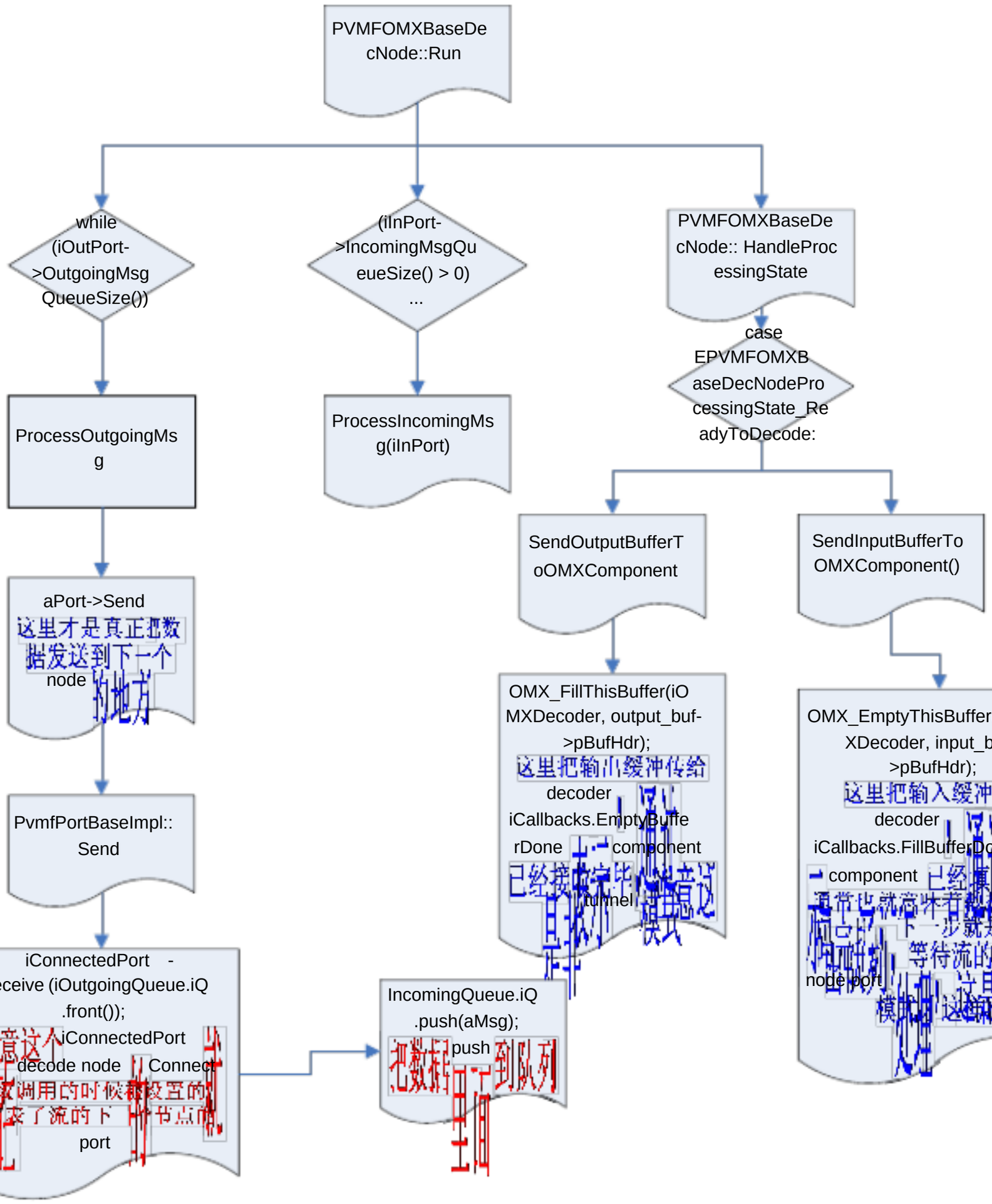
对于 tunnel模式，我们就讨论这么多了；

4.8.3 非tunnel 模式的数据流动

如下图所示：



简单的说就是每次需要发送数据给 decoder时调用 OMX_EmptyThisBuffer ，需要从 decoder取得数据时调用 OMX_FillThisBuffer 到component，然后 component会分别通过 EmptyBufferDone 和FillBufferDone 的回调函数通知 node层，empty结束或者 fill 结束，在里面会把消息的数据加入到特定的队列，在 run 函数里面会分别对消息队列做处理，比如对于发送出去的消息，会调用对端的 send函数发送，在里面其实也就是加入到下一个节点的接收队列而已，于是数据可以由下一个 node的run函数来处理了，流程逻辑如下：



OK，我们下面需要重点讨论的是和我们相关的数据如何从 MIO 输出的逻辑；

4.8.4 MIO 的数据处理

假设数据经过层层节点到了 output node 的输出 port，这时候，它会调用 iMediaTransfer 来传递数据，它的创建前面已经讨论过了（ 4.5.4节），后面 的逻辑如下：

可以看到 iMediaTransfer 本质上就是 AndroidSurfaceOutput 于是进入到了 MIO 的控制范围内了；

在writeFrameBuffer里面会对 fb2有特别的处理，如下：

A）如果使用了 overlay 并且处于屏幕的最上面，将会调用 output2Overlay 来函数来处理，从这个角度来说，以前的播放视频时菜单无法叠加的问题，或许可以避免；

B)通过overlay 的surface取得这个区域的矩形区域描述，但是，这块目前的 marvel 代码里面并没有涉及，也就是关于 overlay2 的创建，还没有完全实现；

C)取得src的YUV 地址以及长度；

D)判断是否需要旋转，方法为源地址的宽减高乘以目的地址的宽减高，如果小于 0就需要旋转，反之不旋转；

E）如果没有对 fb2进行配置调用 overlay2Config 进行配置，这里面会有对 fb2的打开，内存的mmap，目标 YUV 的地址及长度的取得等；

F）然后通过标准的 M2D 的操作把数据传到目的地，也就是 fb2 的地址；

至此，我们所讨论的数据流动，大约就结束了；

5. 同步问题

同步问题是个复杂的问题，我这里只是简单的说说我的理解。在 PV框架里面，工作流程如下图所示：

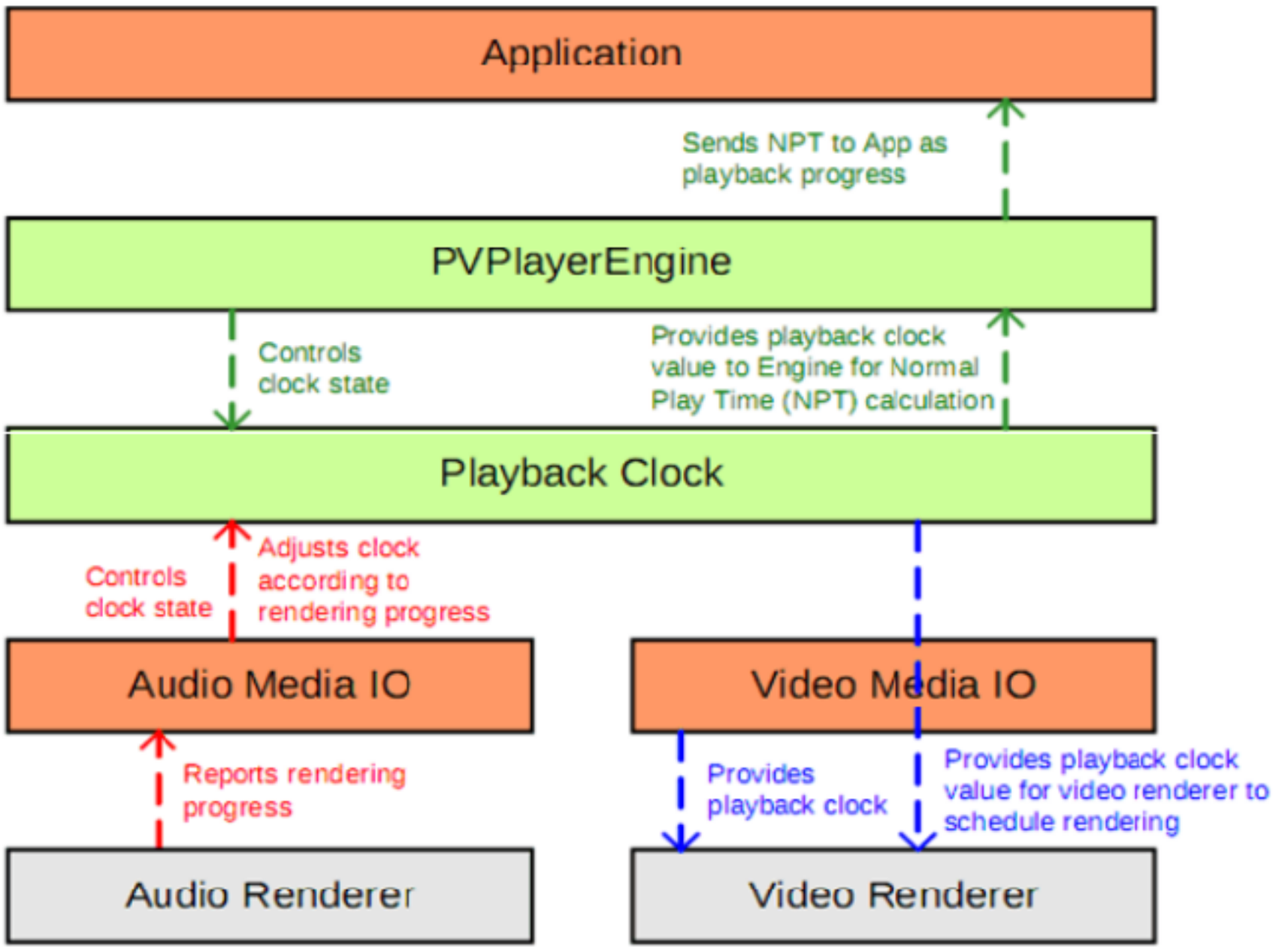


Figure 33: A diagram of the interactions involved in A/V synchronization.

它的思路就是在有音频的时候，以音频为准，否则有视频为准，每一个数据在发出的时候都有打上时间戳，在播放的时候，先取得时间戳，然后对比参考时钟，决定是放还是留，如下图所示：

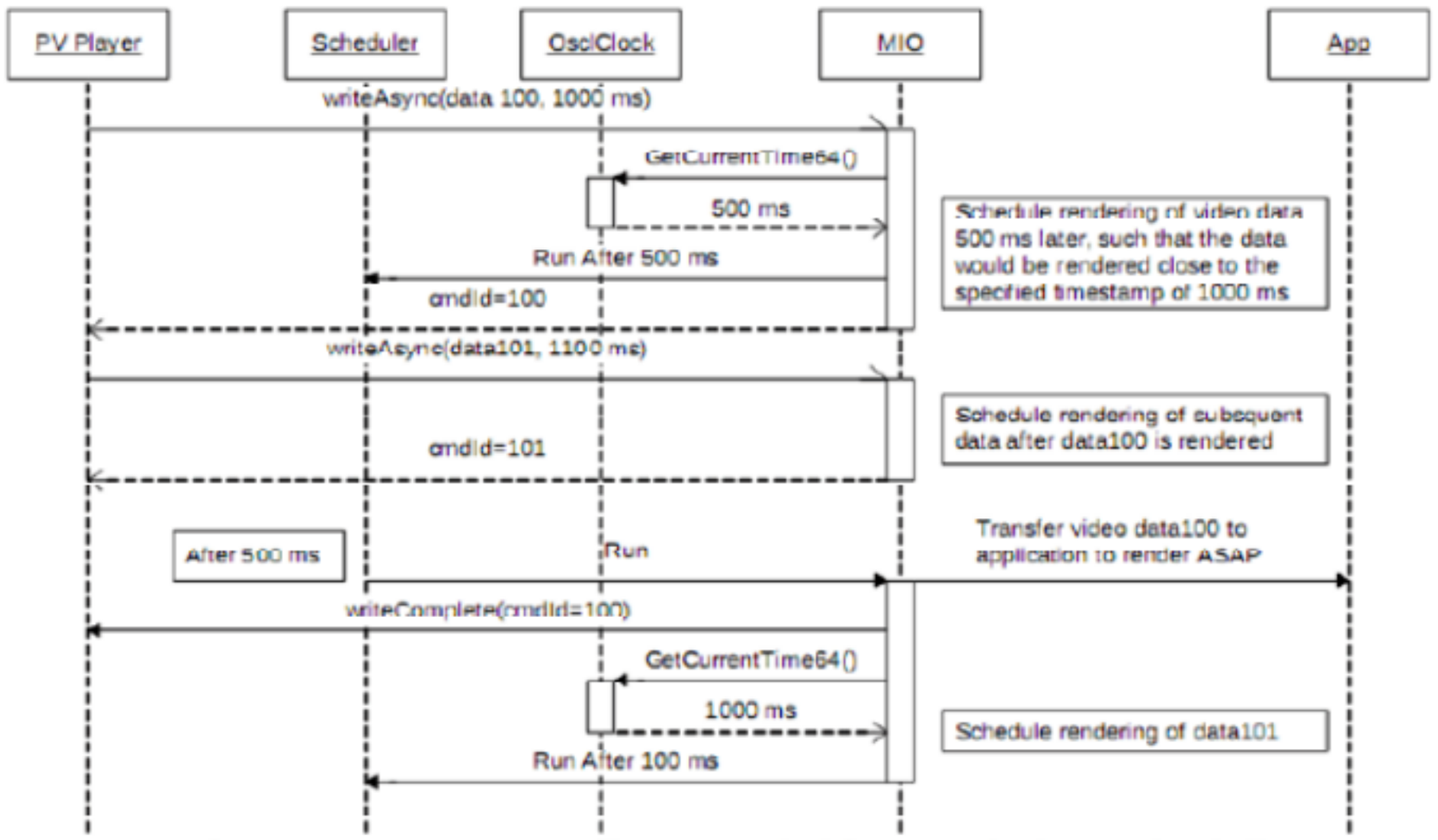


Figure 32: Sequence diagram of video rendering scheduling.

这个图讲得非常清楚，我再啰嗦几句，它的意思就是说，每一个数据包在发送到 MIO 的时候，都是有一个时间戳的，这个时间戳的意思就是说，你最好到这个时间才放出去，于是呢 ,MIO 就去查询当前的 clock的时间，发送时间是 500离时间戳上的 1000少 500ms，于是就起个 timer，告诉它， 500ms后叫醒我，于是就通过这种方式实现了同步，这里需要考虑的是，参考时钟是一直变大，而不会变小的，当用户重新调节播放进度的时候，实际上的数据包上的时间戳和参考时钟就会有一个差距，也就是位移，必须要在参考时钟上反应出来，否则就会不同步了。

6. 关于 component 的集成

因为 marvel之间实现了一套解码 code，所以必须要了解这个集成过程，它的主要代码在 pv_omxmastercore.cpp，omx_interface.h文件里面；

先说一下这个集成的原理：

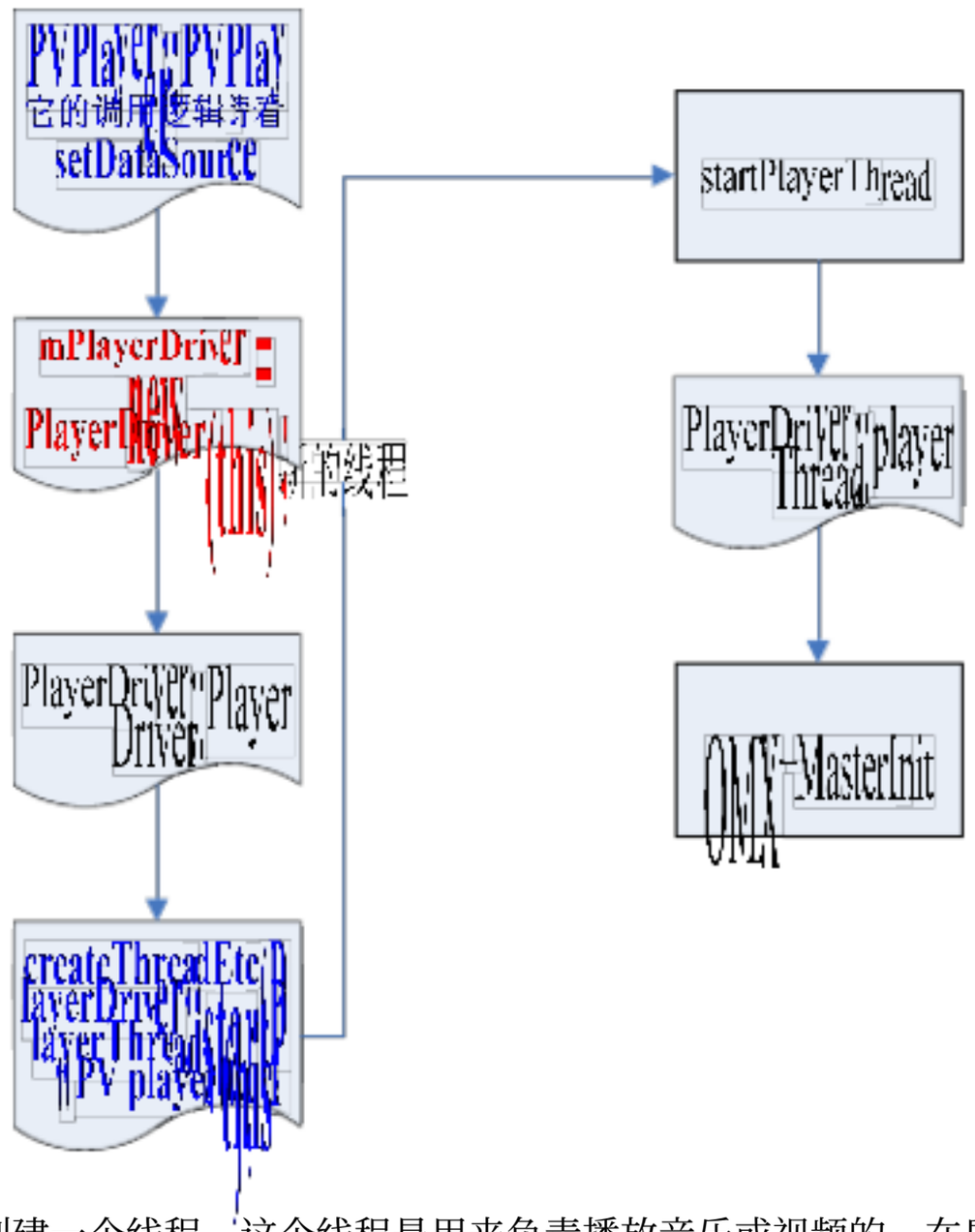
本质上， pv的架构对于解码器的控制都是通过一套以 OMX_ 开头的函数来控制的，比如 component 的枚举，是通过 OMX_ComponentNameEnum来实现的，比如初始化是通过 OMX_Init 来实现的；

所以这个集成的目的就是自己实现一套这样的函数，这样在 openmax的 AL 层使用这些函数来访问解码 component的时候就可以跳转到我们自己实现的解码器里面；也就是说，只要是提供一个库，实现了这些接口，并且想办法告诉 PV的框架，那么它就会把你的库装载进来，然后通过动态库的访问方法，比如 dlsym等就可以访问这个库的接口；当然你必须要遵守它定的规则，那就是实现 omx_interface.h里面所要求的那些函数指针；

OK，有了上面的了解，我们来看看，目前的实现逻辑；

6.1 接口库的加载时机

或许你还记得在前面讨论 setDataSource的时候（4.2节）会创建一个 PVPLayer，但是我们并没有对它的内部进行讨论，现在是时候了，看看下面的逻辑：



这里会创建一个线程，这个线程是用来负责播放音乐或视频的，在启动这个线程之前，就开始了对这个 codec component进行装载（如果还没有装载的话），它的装载是通过 OMX_MasterInit 来实现的，这是一个典型的 singleton模式，查询是否已经加载，没有就开始调用函数 _Try_OMX_MasterCreate来创建，它再调用 _OMX_MasterInit 来初始化，这里分成两步，

第一步是动态库的加载；

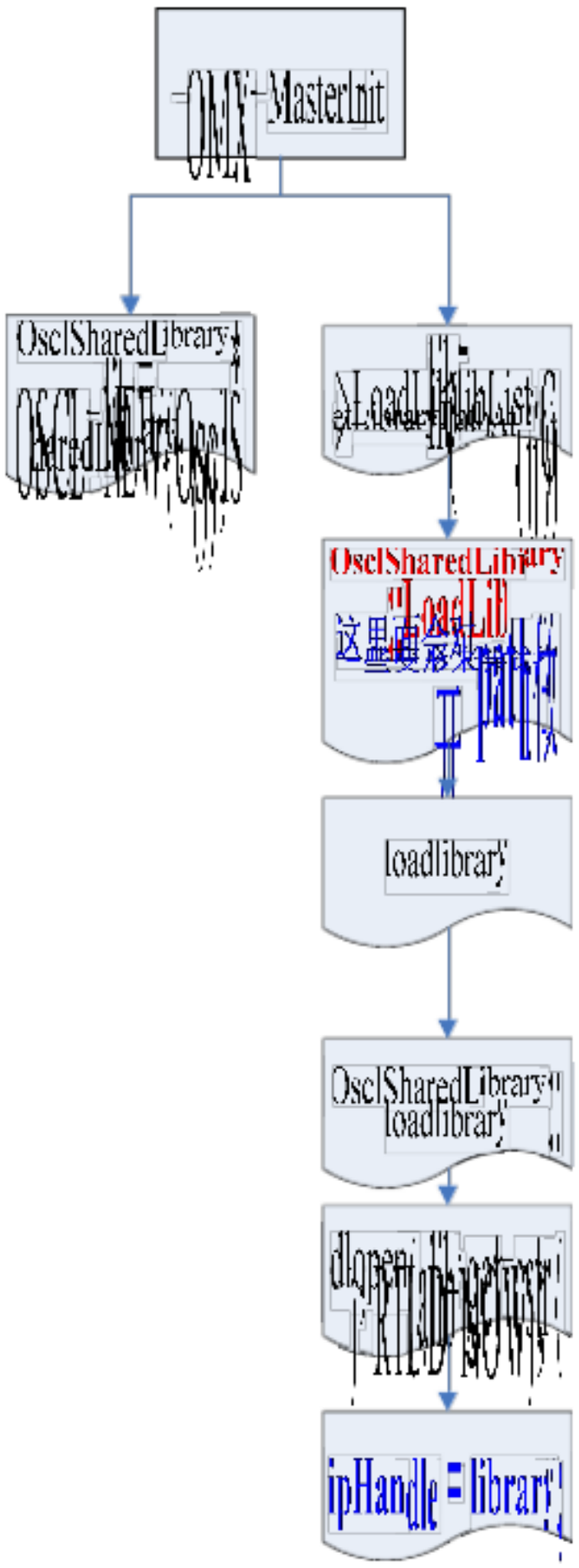
第二步是 omx_interface.h里面的函数指针的赋值过程；

6.1.1 动态库的加载

它首先在 /system/etc目录里面查找 .cfg的文件，如果找到，就从里面读出库的名字，比如对于 marvel 提供的 .cfg文件内如如下：

cat mrvl.cfg
(0x1d4769f0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),"libopencore_rtspreg.so"
(0x1d4769f0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),"libopencore_downloadreg.so"
(0x1d4769f0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),"libopencore_mp4localreg.so"
(0x6d3413a0,0xca0c,0x11dc,0x95,0xff,0x08,0x00,0x20,0x0c,0x9a,0x66),"libopencore_mp4localreg.so"
(0xa054369c,0x22c5,0x412e,0x19,0x17,0x87,0x4c,0x1a,0x19,0xd4,0x5f),"libomx_mrvl_sharedlibrary.so"

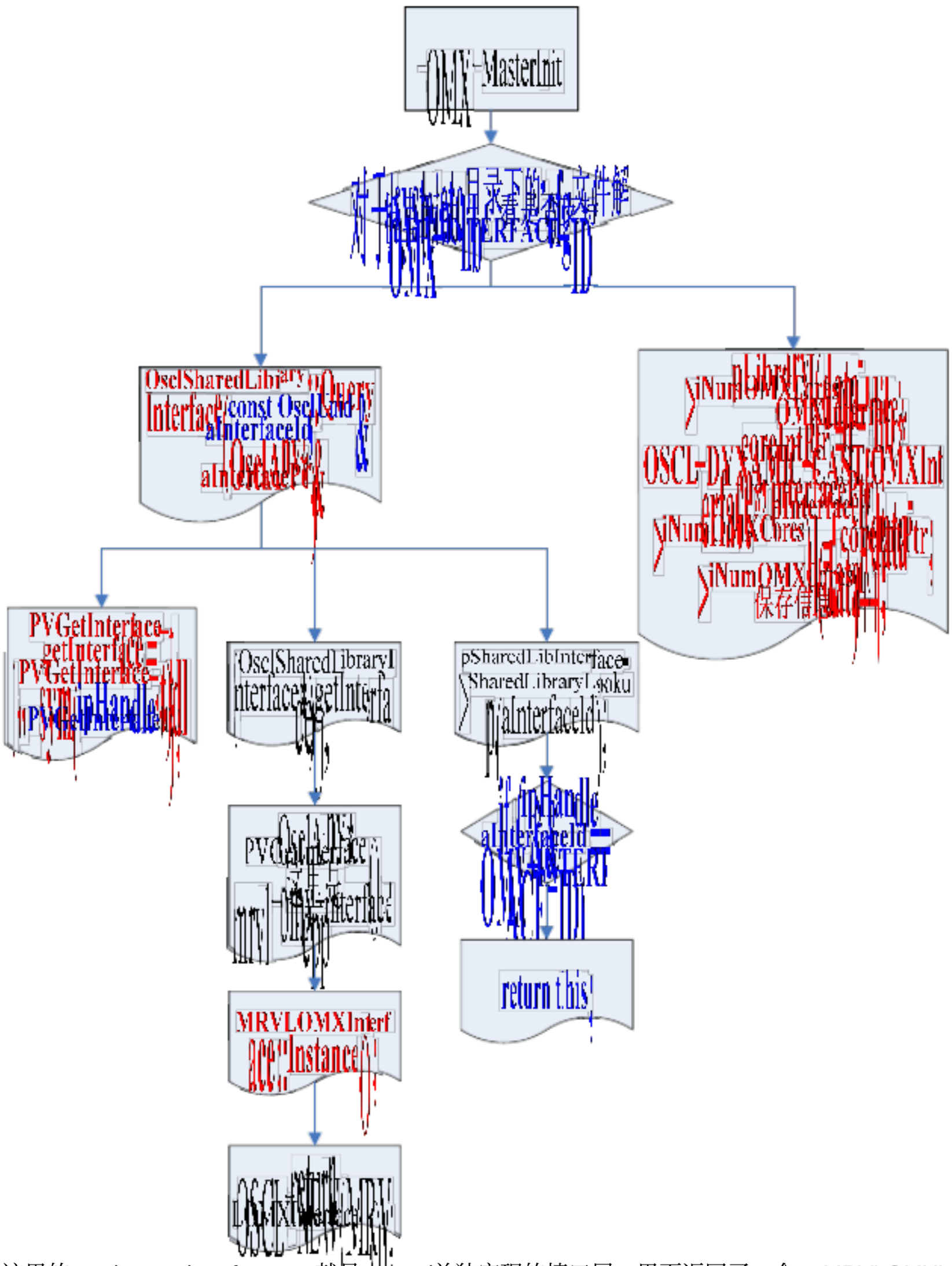
并不是所有的库都是 omx的接口库，它需要支持 OMX_INTERFACE_ID 接口，这在后面会将，先来看看这个装载过程：



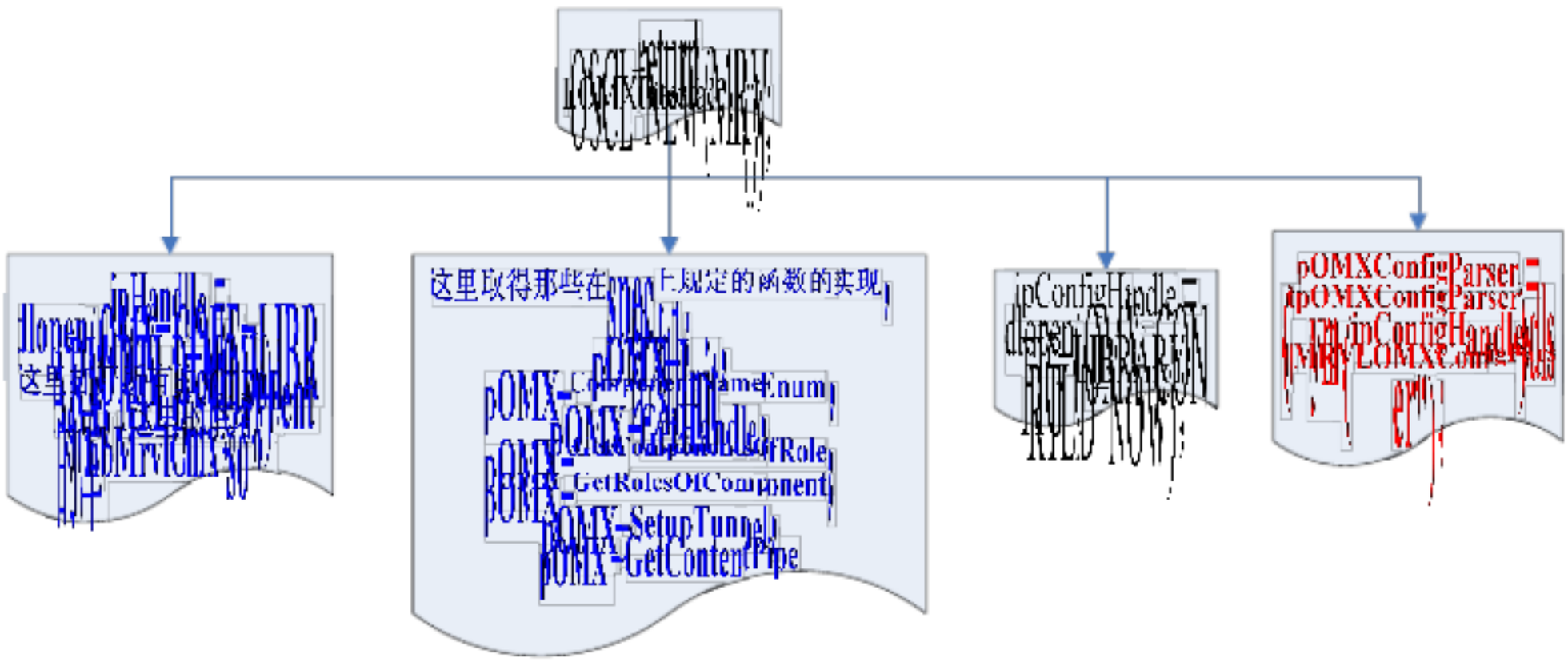
这里取得一个库的 handle供后面使用；

6.1.2 omx_interface 的实现

如下图所示：



这里的 mrvl_omx_interface.cpp就是 marvel单独实现的接口层，里面返回了一个 MRVLOMXInterface 车实例，取得这个实例以后再通过它提供的 SharedLibraryLookup函数判断这个 instance是否支持 OMX_INTERFACE_ID ，答案当然是了，于是返回了 this指针，这里重要的是 MRVLOMXInterface 都做了些什么，请看下面：



这里就为 omx_interface.h里面的接口函数赋值了，可以看到这些函数的指针来自于另外两个库叫做“ libMrvlOmx.so 和 ”libMrvlOmxConfig.so “，

对于 libMrvlOmx.so 的生成是在 vendor/marvell/generic/ipplib/omx/il 里面的 Mdroid.mk 这里面可以看到生成这个库所需要的文件，分成支持 mved还是不支持 mved，分别包含不同的静态库，这些静态库才是真正实现解码的地方；

有了这步的赋值，以后所以的访问都可以通过标准的接口了；

6.1.3 注册表的填充

取得了接口实例以后，还需要对这个 singleton所拥有的注册表（ data->iMasterRegistry ）进行填充，逻辑如下：

A） 首先对所有支持 omx_interface的实例进行遍历；

B） 对每个实例调用 OMX_Init 进行初始化；

C） 调用 GetpOMX_ComponentNameEnum来取得 component的名字，这个名字是根据一个索引来取得的，如下：

```
(*(pInterface[jj]->GetpOMX_ComponentNameEnum()))(
                        ComponentName,
                        PV_OMX_MAX_COMPONENT_NAME_LENGTH,
                       index);
```

D） 对上一步取得的名字调用 GetpOMX_GetRolesOfComponent取得它所支持的 role的个数，根据这个个数分配空间；

E） 再次调用 GetpOMX_GetRolesOfComponent传入上一步分配的空间以取得具体的 role，也就是支持哪些音频，或者视频，具体的类型等，应该都是字符串类型的，也就是一个字符串数组；

F） 对于每一个 component的每一个 role都登记在案，如下：

```
strncpy((OMX_STRING)pOMXMasterRegistry[master_index].CompName, ComponentName,
PV_OMX_MAX_COMPONENT_NAME_LENGTH);
                        strncpy((OMX_STRING)pOMXMasterRegistry[master_index].CompRole,
(OMX_STRING)ComponentRoles[role],
PV_OMX_MAX_COMPONENT_NAME_LENGTH);
                        pOMXMasterRegistry[master_index].OMXCoreIndex = jj;
                        pOMXMasterRegistry[master_index].CompIndex = component_index;
                        master_index++;
```

第一句记录的是 component的名字；

第二句记录的是 component的role ；

第三句记录的是 omx_interface 的索引，也就是最外层的 for 循环的值；

第四句记录的是 component的索引，它会一直递增，注意它和 master_index 是不一样的，因为

同一个 component 可能有好几个 role ；
第五句递增 master_index 这个值描述的是注册表里面的项数，其中以 role 以及 component 联合作为 primarykey ；
OK，等大循环结束所以信息也都记录下来了，以供后面的查询；
具体的要取得某个 component 的时候，只需要传入 component 的名字，调用 OMX_GetHandle 就可以取得对应的 handle 了，它的原型如下：

```
OMX_API OMX_ERRORTYPE OMX_APIENTRY OMX_GetHandle(
  OMX_OUT OMX_HANDLETYPE *          pHandle ,
  OMX_IN OMX_STRING       cComponentName,
  OMX_IN OMX_PTR        pAppData,
  OMX_IN OMX_CALLBACKTYPE *   pCallBacks
 )
```

有了这个 pHandle，就有了一切的控制权！！！

7. 集成总结

对于集成，我们需要做两件事；

1, marvel 提供的 IPP 写的 component 必须能注册进 opencore，这一步就是第 5 章描述的，应该 marvel 已经帮我们做好了；
2, overlay2 的集成，需要考虑，如何把 fb2 的 surface 整合进去，这样的话，在 playerdriver 设置 videosurface 的时候就可以设置 fb2 的 surface 了，调用到 MIO 层的时候，就可以使用了，这步目前 marvel 还没有实现，需要我们自己做，具体就是要实现 libhardware/modules/overlay 里面的 overlay.cpp，不过从现在的 patch 可以看出 marvel 的思路是通过 surfaceflinger 来判断出是否是 fb2 在最上面，然后在 mio 层直接控制 fb2，压根不想通过 android 提供的标准的 overlay.cpp 的方式实现视频的播放，而 omap 的实现却是通过 overlay.cpp 的方式做的；

8. 未分析

1, 关于 track list 的建立没有详细分析；
2, Parser 的工作原理没有分析；
3, 其它，没有考虑到的；

另外，文档多是以图的形式描述，也许有的人很不喜欢，抱歉！

9. reference

【1】, openmax_il_spec_1_1_2.pdf
【2】, openmax_call_sequences.pdf
【3】, pvplayer_developers_guide.pdf
【4】, mio_developers_guide.pdf
【5】, http://hi.baidu.com/wylhistory/blog/item/584fde248b69fd20d40742b2.html
【6】, http://hi.baidu.com/wylhistory/blog/item/b9f41cefc133233dacafd5c4.html
【7】, http://hi.baidu.com/wylhistory/blog/item/117c6b31bc8a52a35fdf0e34.html

作者： wylhistory
联系方式： wylhistory@gmail.com