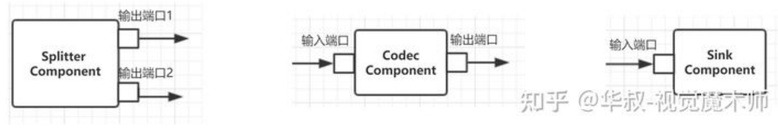
OpenMax (OMX) 开发入门 —— OMX IL层

华叔-视觉魔术师
横跨 音视频编解码、图形图像算法、Android嵌入式的大叔

一、端口和组件



1. 组件是独立的一个处理模块，可以有内部独立的线程（但并不一定）处理数据。 通常组件类型有：splitter组件、hot组件、sink组件、clock组件等。

2. 组件之间通过端口进行数据通信，每个组件至少要有一个端口，根据数据方向区分：有输入端口和 输出端口 两种方向。任意一个端口只能是其中一种方向。

3. 端口根据通信的数据类型可以分为四种：Video_Port; Audio_Port; Image_Port; Other_Port; 其中除了音视频和图像数据，其他数据都是通过Other_Port来传递，比较典型的就是Clock全局时钟。

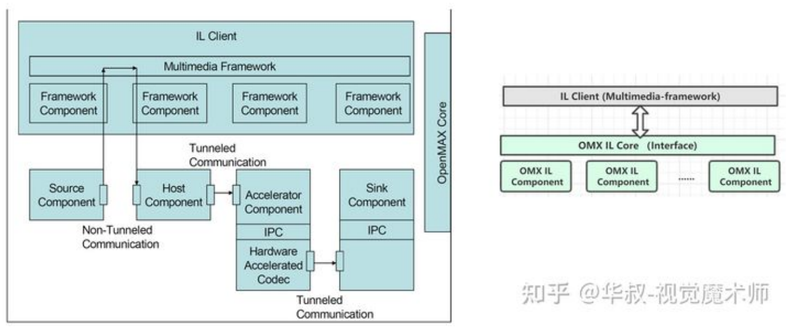
4. 每个组件都有自己独立的运行状态机，总共有6个状态。

5. 组件支持两种不同的Profile:

1）Base Profile: 仅支持 non-tunnel方式的数据通信，也可能支持Proprietary模式

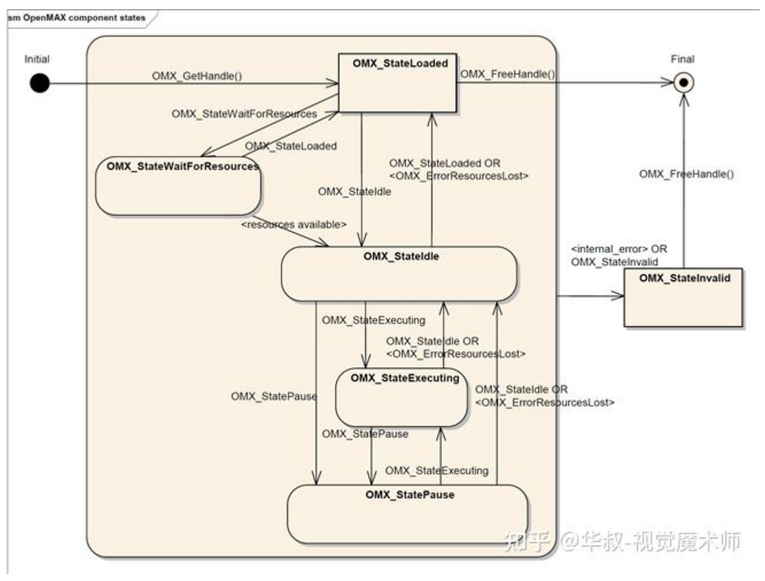
2）Interop Profile: 必须支持tunnel 和 non-tunnel两种数据通信方式，也可能支持Proprietary模式

二、内部架构



1. 组件之间数据通信，既可以依赖IL Client调用层，也可以不依赖IL Client，通过隧道方式内部进行传输。
2. OMX Core负责IL层所有组件库的初始化、单个组件查询、加载、卸载、组件之间Tunnel建立等，对于单个组件操作也是通过 IL Core层的接口进行。

三、状态机



1. **OMX_StateInvalid** : 组件运行中产生无法恢复的错误，不能再继续进行了，只能卸载组件

2. **OMX_StateLoaded** : 组件已经加载到系统中，但是还没有进行初始化

3. **OMX_StateWaitForResources**：组件正在等待资源，当资源到位后会切换成Idle状态

4. **OMX_StateIdle**：组件初始化完成，一切准备就绪

5. **OMX_StateExecuting**：组件正常运行，进行相关数据处理

6. **OMX_StatePause**：组件暂停运行

四、组件库函数

1. IL层支持的库函数主要有两组：一组是针对所有组件库的Core函数； 另外一组是针对单个组件的函数。
2. 大部分的函数操作都是**同步调用**，并且严格限制函数的执行时间（例如：OMX_SetConfig 限制在5ms内；OMX_UseBuffer 限制在20ms内）。
3. **OMX_FillThisBuffer**() 和 **OMX_EmptyThisBuffer**()是**异步调用**，组件在实际执行完成后通过 **EmptyBufferDone**() 和 **FillBufferDone**() 两个回调通知执行完成。
4. **OMX_SendCommand**() 函数也是异步调用，支持5种命令调用

1）OMX_CommandStateSet： 切换状态机

2）OMX_CommandFlush： 刷新缓冲区

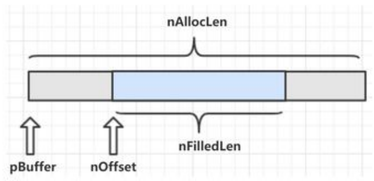
4）OMX_CommandPortEnable 启用某个端口

5）OMX_CommandMarkBuffer：标记缓冲区对象（如：仅解码）

这些命令执行完成后通过 EventHandler() 回调来通知完成

五、缓冲区对象

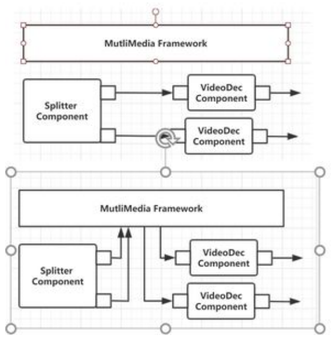
```
typedef struct OMX_BUFFERHEADERTYPE
{
    OMX_U32 nSize;              /**< size of the structure in bytes */
    OMX_VERSIONTYPE nVersion;   /**< OMX specification version information */
    OMX_U8* pBuffer;            /**< Pointer to actual block of memory
                                     that is acting as the buffer */
    OMX_U32 nAllocLen;          /**< size of the buffer allocated, in bytes */
    OMX_U32 nFilledLen;         /**< number of bytes currently in the
                                     buffer */
    OMX_U32 nOffset;            /**< start offset of valid data in bytes from
                                     the start of the buffer */
    OMX_PTR pAppPrivate;        /**< pointer to any data the application
                                     wants to associate with this buffer */
    OMX_PTR pPlatformPrivate;   /**< pointer to any data the platform
                                     wants to associate with this buffer */
    OMX_PTR pInputPortPrivate;  /**< pointer to any data the input port
                                     wants to associate with this buffer */
    OMX_PTR pOutputPortPrivate; /**< pointer to any data the output port
                                     wants to associate with this buffer */
    OMX_HANDLETYPE hMarkTargetComponent; /**< The component that will generate a
                                              mark event upon processing this buffer. */
    OMX_PTR pMarkData;          /**< Application specific data associated with
                                     the mark sent on a mark event to disambiguate
                                     this mark from others. */
    OMX_U32 nTickCount;         /**< Optional entry that the component and
                                     application can update with a tick count
                                     when they access the component.  This
                                     value should be in microseconds.  Since
                                     this is a value relative to an arbitrary
                                     starting point, this value cannot be used
                                     to determine absolute time.  This is an
                                     optional entry and not all components
                                     will update it.*/
    OMX_TICKS nTimeStamp;       /**< Timestamp corresponding to the sample
                                     starting at the first logical sample
                                     boundary in the buffer. Timestamps of
                                     successive samples within the buffer may
                                     be inferred by adding the duration of the
                                     of the preceding buffer to the timestamp
                                     of the preceding buffer.*/
    OMX_U32     nFlags;         /**< buffer specific flags */
    OMX_U32 nOutputPortIndex;   /**< The index of the output port (if any) using
                                     this buffer */
    OMX_U32 nInputPortIndex;    /**< The index of the input port (if any) using
                                     this buffer */
} OMX_BUFFERHEADERTYPE;
```



1. pBuffer:　分配的缓冲区缓冲区开始地址
2. nAllocLen: 分配的缓冲区的整个长度
3. nOffset:　当前有效数据的偏移位置
4. nFilledLen: 当前有效数据的长度

所以当前有效数据范围:
(pBuffer+nOffset) ~ (pBuffer+nOffset+nFilledLen)

六、组件间通信方式

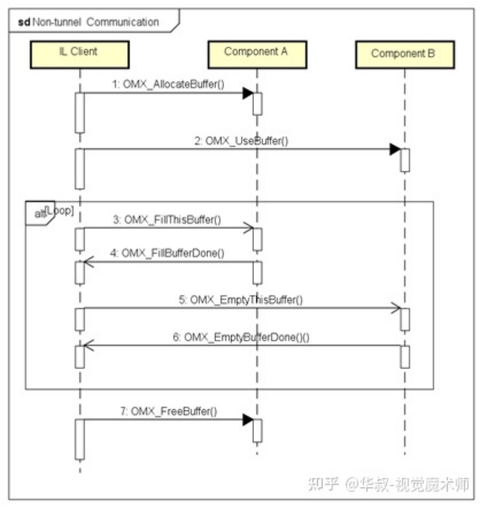


1. **Tunnel模式**
　组件之间直接通过端口来交换数据，不需要上面多媒体框架层来参与。在组件运行过程中，数据自动由前一个组件的输出端口传递给下一组件的输入端口中，这种方式可以简化框架层对于组件的使用。（组件通常有自己的内部线程）

2. **Non-tunnel模式**
　组件的端口之间不直接发生交互，由上面框架层来控制组件之间的数据交互。框架层获取前一个组件输出端口的缓冲区数据，然后送入下一个组件的输入端口，数据流的驱动由框架层来控制。

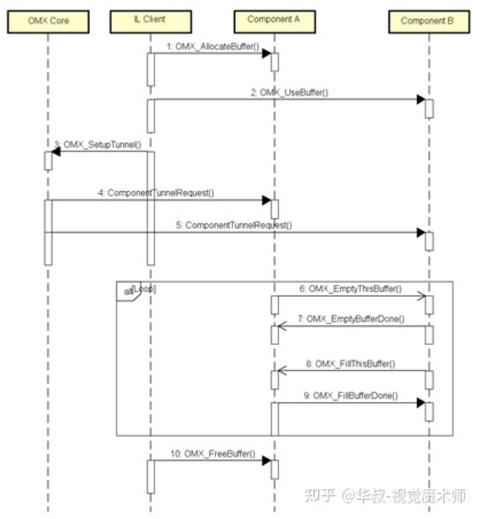
3. **Proprietary 模式**
　两个组件之间直接交换数据的专有方式

1. Non-tunnel 通信方式



1. IL Client通过OMX_AllocateBuffer() 在组件A的输出端口上创建缓冲区对象，这个缓冲区对象直接返回给 IL Client。
2. IL Client再将这个缓冲区对象，通过 OMX_UseBuffer()指定给组件B的输入端口使用这个缓冲区对象。
3. 在循环的数据处理过程中，IL Client调用OMX_FillThisBuffer() 命令组件A将处理好要输出的数据填入这个缓冲区中，注意：这个调用是异步的，这个调用返回后，缓冲区数据可能还没有填充好。
4. 组件A内部先进行数据处理，处理完成后将输出数据填入缓冲区，然后通过OMX_FillBufferDone()回调函数通知IL Client数据已经准备好。
5. IL Client接收到回调后，调用OMX_EmptyThisBuffer()命令组件B来取缓冲区中的数据，注意：这个调用也是异步的，这个调用返回后，缓冲区数据可能还没有取走。
6. 组件B内部根据优先处理顺序，将缓冲区中的数据全部取走后，然后通过OMX_EmptyBufferDone()回调函数通知IL Client缓冲区已经清空，可以继续使用。 如此反复来传递缓冲区对象，使得两个组件通过一个缓冲区来进行通信。
7. 最后在组件A上释放这个缓冲区对象

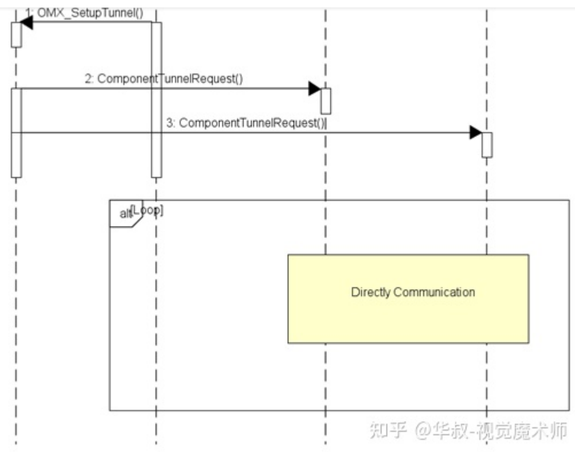
2. Tunnel 通信方式



1. IL Client通过OMX_AllocateBuffer() 在组件A的输出端口上创建缓冲区对象，这个缓冲区对象直接返回给 IL Client。
2. IL Client再将这个缓冲区对象，通过 OMX_UseBuffer()指定给组件B的输入端口使用这个缓冲区对象。
3. IL Client调用 OMX Core的 OMX_SetupTunnel(hCompA, nOutPortIdx, hCompB, nInPortIdx) 函数来将两个组件的输入输出端口建立隧道通信方式。注：这个函数内部实现会调用两个组件的内部函数ComponentTunnelRequest()来传递组件本身的信息。
4. **Push数据方式**：组件A数据准备完毕后，直接调用组件B上的 OMX_EmptyThisBuffer()方法让组件B取数据，组件B获取完数据后，将OMX_EmptyBufferDone()通知直接回调给组件A，通知组件A这个缓冲区已经清空，可以继续使用了。
5. **Pull数据方式**：组件B需要数据的时候，直接调用组件A上的 OMX_FillThisBuffer()方法让组件A填充数据，组件A填充完成后，将OMX_FillBufferDone()通知直接回调给组件B，通知组件B这个缓冲区上数据可以使用了。

与 Non-tunnel方式主要的差异就是：建立隧道后，组件之间的数据通信不需要IL Client参与了，两个组件内部直接进行。 通常支持Tunnel通信方式的组件都有内部线程，方便数据同步处理

3. Proprietary 通信方式



1. 组件之间有类似DMA之类的直接通信机制，不需要外部提供缓冲区和控制缓冲区对象传递。
2. IL Client 只需要调用 OMX Core的 OMX_SetupTunnel(hCompA, nOutPortIdx, hCompB, nInPortIdx) 函数来将两个组件的输入输出端口建立关联即可。
3. 通常这种情况下的组件都是由硬件来实现，直接通过硬件或者系统内部的共享缓冲区来访问数据
4. 通信时也不再有 OMX_FillBufferDone() 和 OMX_EmptyBufferDone() 的回调

文章系列目录:

华叔-视觉魔术师：OpenMax (OMX) 开发入门 —— 绪论