

**OpenMAXTM Integration Layer Application Programming Interface Specification**

Version 1.1.2

Copyright © 2008 The Khronos Group Inc.

September 1, 2008

Document version 1.1.2.0

中文版由ZER0000（gayhub：<https://github.com/ZER0000> ）创建，遵循GPLv2协议

本文链接为：<https://github.com/ZER0000/OpenMAX-IL-API-Specification->

中文版本：V1.0 创建时间：2018年6月1日

第一章

# 2 OpenMAX IL 介绍与架构

文档的这一部分描述了OpenMAX IL的特性与架构。

## 2.1 OpenMAX IL 简介

OpenMAX IL层是一系列的软件接口，用于在系统中提供一个基于软件组件的访问层。软件接口的作用是为组件提供不同的初始化和命令方法，并提供一个具有标准化的命令集和组件构建/销毁方法的软件层。

### 2.1.1 架构概述

假想一个多媒体系统，它需要实现4个多媒体处理功能，标记为F1，F2，F3和F4。这些功能中每一个的实现都可能来自于不同的供应商，或者由组织内部不同的团体研发。每个功能对于安装和卸载都有不同的要求，每个功能又可能需要使用不同的方法来进行配置和数据传输。OpenMAX IL API提供了将这些功能（单独或以逻辑组的方式）封装到组件中的方法。该API中包含一个标准协议，它能够使来自不同供应商的组件相互交换数据并可互换使用。

OpenMAX IL API与一个称为IL 客户端的更高层次的实体进行交互，该实体通常可能为图形多媒体框架的部分功能、OpenMAX AL或一个应用程序。IL客户端与OMX的核心进行交互，该核心称为Core。IL客户端通过使用OpenMAX IL Core来进行组件的加载和卸载操作，设置两个OpenMAX IL组件间的直接通信以及访问组件的方法。

IL客户端通常通过IL Core来与组件进行通信。在大多数情况下，这种通信就是通过调用IL Core提供的某一个宏，这些宏最后直接转换为对应组件的方法。例外情况是IL客户端会调用可以实际工作的Core函数，这些函数包括组件的创建和销毁，已安装组件的查询，已安装组件所支持的role，两个组件的隧道连接等等。

组件体现了多媒体的处理功能。虽然本规范中明确定义了OpenMAX IL Core的功能，但组件的提供者定义组件的功能。组件根据它们导出的参数结构（音频，视频，图像和其他（例如用于同步的时间数据））对四种类型的数据进行操作。

OpenMAX IL组件通过其句柄提供出一组对其功能进行访问的标准函数。 这些函数允许客户端获取和设置组件，对端口进行参数配置，获取并设置组件的状态，向组件发送命令，接收事件通知，分配缓冲区，与组件的单个端口建立通信，以及与组件的两个端口建立通信。

每个OpenMAX IL组件都应至少有一个端口来声明OpenMAX IL一致性。 虽然供应商可能提供不带端口的OpenMAX IL兼容组件，但大部分一致性测试依赖于至少一个一致性端口。OpenMAX IL中定义的四种类型的端口对应于端口可以传输的数据类型：音频，视频，图像数据端口以及其他端口。 根据端口是否消耗或产生缓冲区，每个端口又被定义为输入端口或输出端口。

在包含四个多媒体处理功能F1，F2，F3和F4的系统中，系统实施者可以为每个功能提供标准的OpenMAX IL接口。 实施者只需要选择一些功能的组合，功能的划分基于端口。 图2-1显示了OpenMAX IL实施者对于这些功能的一些可能分区。

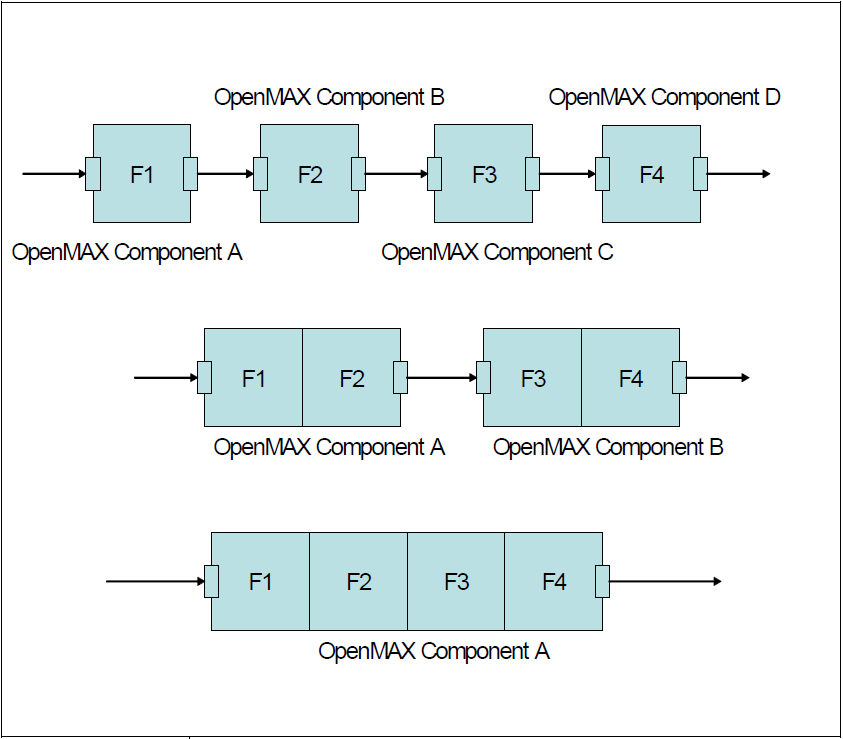


图2-1. OpenMAX IL实现的对于功能的一些可能分区

### 2.1.2 核心词汇

本节介绍OpenMAX IL API中常使用的首字母缩略词和术语。

#### 2.1.2.1 关键术语

表2-1列出了描述OpenMAX IL API中使用的关键术语

表2-1：关键术语

|  |  |
| --- | --- |
| **关键字** | **含义** |
| Accelerated component  加速组件 | 一种OpenMAX IL组件，它包含一个功能并且部分运行在加速器上。 |
| Accelerator  加速器 | 设计出来的用于加速某些功能的硬件，这个硬件也可以被称为加速硬件。需要注意的是，加速器实际上可能是运行在另一个处理器上的软件，根本不是硬件。 |
| Buffer Supplier  缓冲区提供者 | 传入一个端口的缓冲区的实际“拥有者” |
| Container  容器 | 用于封装基本数据流和相关元数据的格式（例如3gp文件格式）。 |
| Content Pipe  内容管道 | 访问（读取或写入）OpenMAX IL外部某些内容的一种抽象方法。外部的内容可以表现为一个文件或者可以利用系统文件I/O功能的管道，但抽象不限于这些特定类型的内容或内容访问。 |
| Component Group  组件组 | 一组功能上相互依赖的组件。 如果组中的某一个组件无法操作，那么该组中的所有组件都无法运行。 |
| Component Suspension  组件挂起 | 一个组件在缺少关键资源时，暂时挂起其拥有的其他资源，以便在所需资源再次可用时，该组件可以从挂起点恢复。 |
| Dynamic resources  动态资源 | 任何一个组件资源是在该组件从初始状态转换到Idle状态时进行分配的。动态资源分配是不推荐的，只有在分配参数（例如，内部缓冲区的大小或数量）在组件资源分配的优选时间内未知时，才会选择动态资源分配。 |
| Host processor  主处理器 | 多核系统中控制媒体加速，并且通常运行高级操作系统的那个处理器。 |
| IL client  IL 客户端 | 调用OMX Core或者组件方法的软件层。IL客户端可能是位于GUI应用程序下面的软件层，如GStreamer，或者可能位于GUI层下面的几层。在本文档中，应用程序是指任何调用OpenMAX IL方法的软件。 |
| Main memory  主存储器/主内存 | 通常是主处理器和加速器共享的外部存储器。 |
| OpenMAX IL component  OpenMAX IL 组件 | OpenMAX IL 组件的目的是包含目标系统中所需的功能，OpenMAX IL wrapper为被封装的函数提供标准接口。 |
| OpenMAX IL core  OpenMAX IL 内核 | 特定于平台的代码，并且具有找到和将OpenMAX IL组件加载到主内存中的功能。当应用程序指出不再需要一个OpenMAX IL组件时，OpenMAX IL core还需要负责从内存卸载组件。  通常来讲，在OpenMAX IL Core将组件加载到内存后，Core将不参与应用程序与组件之间的通信。 |
| Resource manager  资源管理器 | 管理系统中硬件资源的软件实体。 |
| Static resources  静态资源 | 作为组件进入Idle状态的先决条件分配的组件资源，大多数组件资源都属于这个类别。 |
| Synchronization  同步 | 门控一个组件与另一个组件操作的机制。 |
| Tunnels/Tunneling  隧道/隧道化 | 建立在两个OpenMAX IL组件之间直接进行标准数据通信的路径。 |

### 2.1.3 系统组件

图2-2描述了OpenMAX IL的各种通信类型。每个组件可以有任意数量的端口用于数据通信。带有单个输出端口的组件称为源组件。具有单个输入端口的组件被称为接收器组件。完全在主处理器上运行的组件称为主机组件。运行在松耦合加速器上的组件称为加速器组件。OpenMAX IL可以直接与应用程序集成，也可以与支持异构实现的多媒体框架组件集成。

这里描述了三种类型的通信。非隧道通信是指在IL客户端和组件之间交换数据缓冲区的机制。隧道通信是指组件间以标准方式直接交换数据缓冲区的机制。专有通信是指用于两个组件之间的直接数据通信，并可以在收到隧道请求时用其替代，只要这两个组件都能够允许这么做。

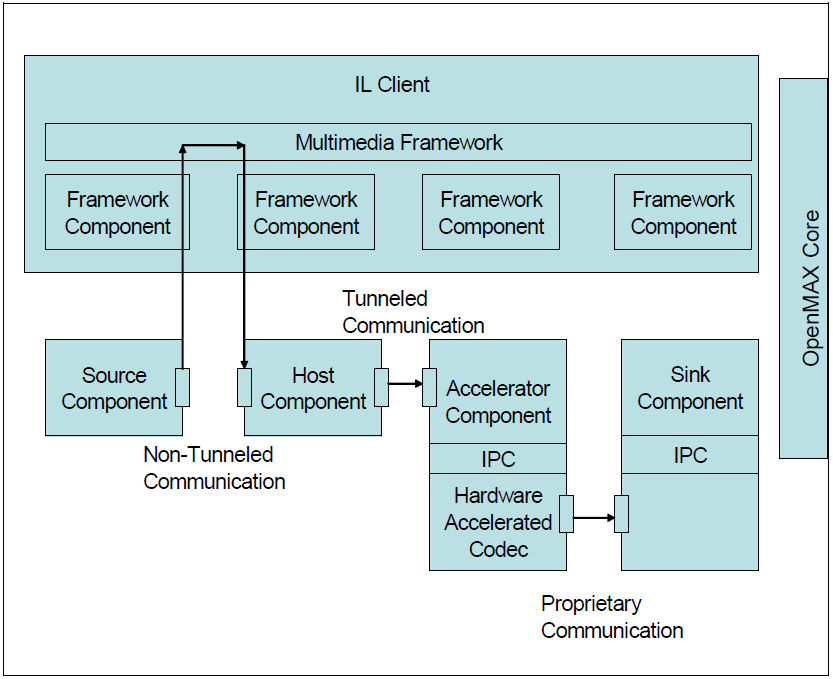


图2-2. OpenMAX IL API 系统组件

#### 2.1.3.1 配置组件

OpenMAX IL组件按照功能可以区分为两组配置组件：基本配置组件和交互配置组件。

基本配置组件应支持非隧道通信。 基本配置组件可支持专有通信。 基本配置组件不支持隧道通信。

交互配置组件是基本配置组件的超集。交互配置组件应支持非隧道通信和隧道通信。一个交互配置组件可以支持专有通信。

交互配置组件和基本配置组件的主要区别在于该组件是否支持隧道通信。基本配置组件的存在旨在简化OpenMAX IL实施者的实现难度,因为基本配置组件不需要实现隧道通信。

表2-2：每个配置组件支持的通信类型

|  |  |  |
| --- | --- | --- |
| **通信类型** | **基本配置** | **交互配置** |
| 非隧道化通信 | 是 | 是 |
| 隧道化通信 | 否 | 是 |
| 专有通信 | 是 | 是 |

### 2.1.4 组件状态

每个OpenMAX IL组件都可以经历一系列状态转换，如图2-3所示，每个组件的状态首先被认为Unloaded的。组件应通过调用OpenMAX IL Core的一个函数来加载，其他所有状态转换可以通过直接与组件通信来实现。

当一个组件在进行状态转换时由于操作了无效数据，就可能会进入Invalid状态。例如，如果回调函数的指针未设置为有效位置，组件可能会超时并向IL客户端发出错误警报。IL客户端应在检测到无效状态时停止，反初始化，卸载和重新加载组件。如图2-3中描述的，Invalid状态可从任何状态进入，但退出Invalid状态的唯一方法是卸载并重新加载组件。

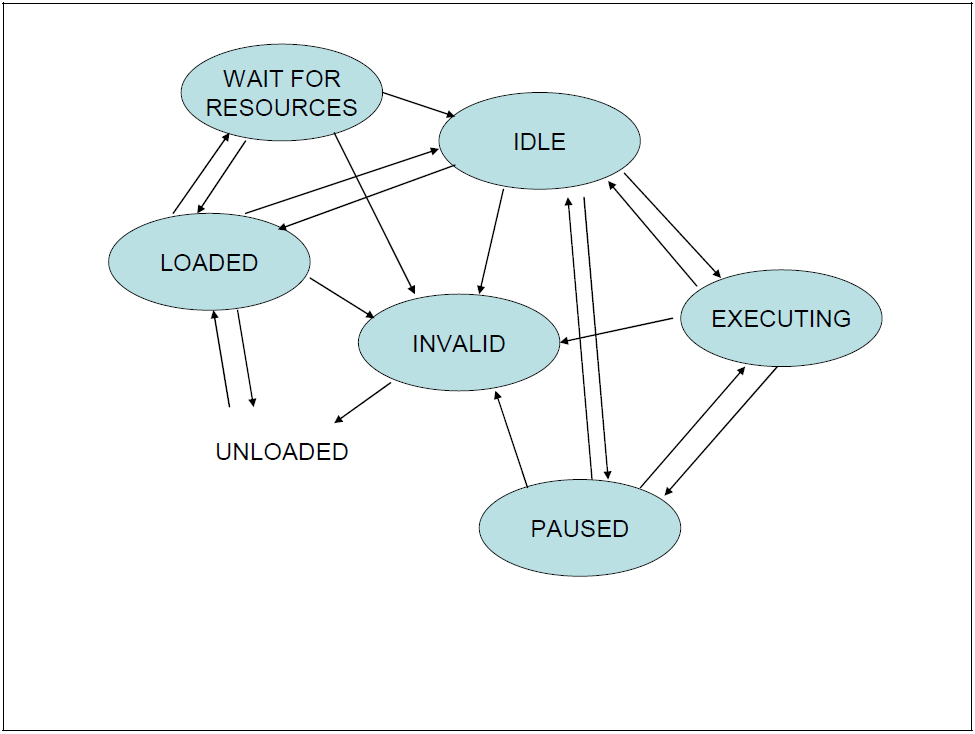


图 2-3. 组件状态

通常情况下，组件在IDLE状态时应具有所有可操作的资源。但是，例外情况是，在转换为IDLE状态时，资源分配的参数尚未知道。例如，解码视频的组件在数据流被检查前，是不知道需要多少参考帧的，然而组件在转换为IDLE状态之前无法检查流。在这种情况下，组件可能会推迟分配资源，直到知道分配参数为止。如果动态分配失败，组件应挂起它自己。因此，我们经常区分这两种资源，例如，向IDLE过渡前“先期”分配的资源为静态分配的资源，稍后通过调用而获得的资源称为动态分配的资源。

转换到IDLE状态时可能会失败，因为此状态需要分配所有可操作的静态资源。当从LOADED到IDLE的状态转换失败时，IL客户端可能会再次尝试或选择将该组件置于WAIT FOR RESOURCES状态。进入WAIT FOR RESOURCE状态后，组件向特定于供应商的资源管理器注册，以便于在资源可用时向其发出警报，该组件将随后转换到IDLE状态。IL客户端可以在除INVALID以外的所有其他状态下发送控制命令。

IDLE状态表示组件具有其所需的全部静态资源，但不处理数据。EXECUTING状态表示组件正在等待接收buffers来处理数据，并将按照第3章的规定来进行必要的回调。PAUSED状态保持组件与buffers执行过程的上下文，而不处理数据或交换buffers。从PAUSED状态转换到EXECUTING状态，组件从停止的地方恢复执行对buffer的处理。从EXECUTING或PAUSED转换到IDLE将会导致处理buffers过程的上下文丢失，需要重新引入流。从IDLE转换到LOADED会导致操作资源（如通讯缓冲区）丢失。

### 2.1.5 组件架构

图2-4描述了组件的体系架构。请注意，每个组件只有一个入口（通过组件的句柄而引出的一系列标准函数数组），但可能有多个回调函数取决于组件的端口数。每个组件都将调用指定的IL客户端事件处理程序，每个端口也会对指定的外部函数进行调用（或回调）。指向buffer头部的指针队列也与每个端口相关联,这些buffer头指向实际的buffer。命令函数也有一个命令队列。所有参数或配置调用都在特定的索引上执行，并包含与该参数或配置相关的结构体，如图2-4所示。

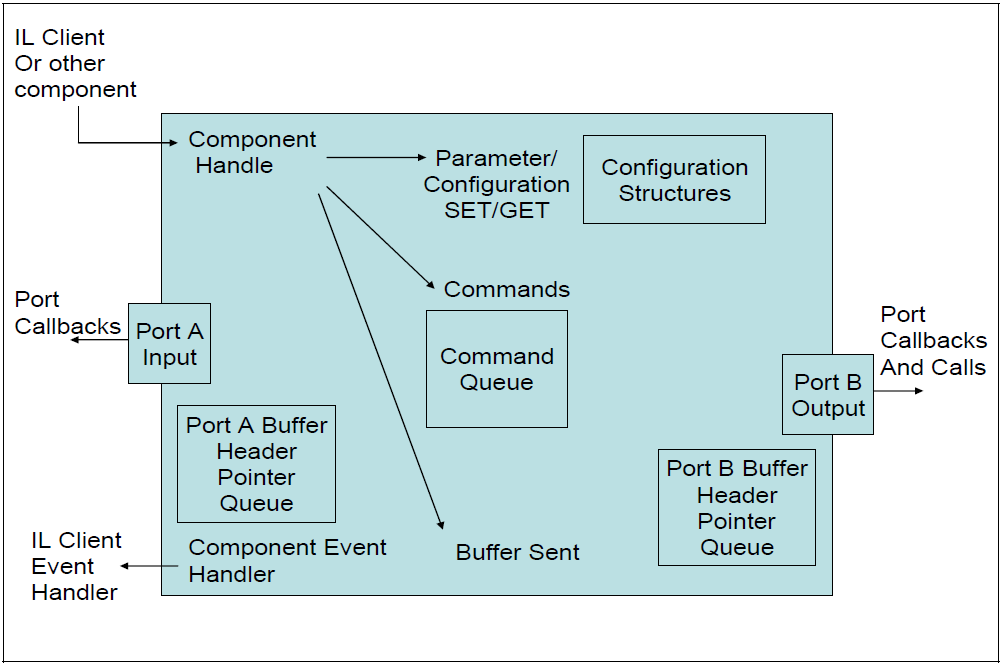


图 2-4. OpenMAX IL API 组件架构

一个端口应该支持对IL客户端的回调，并且当它为交互配置组件的一部分时，应该支持与其他组件上的端口通信。

### 2.1.6 沟通行为

一旦从OpenMAX IL Core接收到组件的句柄，就完成了对组件的配置。组件的数据通信调用是非阻塞的，并且当配置了端口数量，每个端口配置为特定的数据格式，该组件处于适当的状态时，数据通信就开始使能。数据通信特定于组件的端口。输入端口总是通过OMX\_EmptyThisBuffer函数从IL客户端调用（有关更多信息，请参见第3.2.2.17节），输出端口总是通过OMX\_FillThisBuffer从IL客户端调用（有关更多信息，请参见第3.2.2.18节）。在in-context实现中，将在返回之前调用EmptyBufferDone或FillBufferDone回调函数。图2-5描述了在in-context与out-of-context实现的预期行为。请注意，IL客户端不应该对返回/回调序列进行假设，以启用in-context和out-of-context OpenMAX IL组件的异构集成。

（注： 当OpenMAX组件与应用程序处于同一个进程/线程中时，称为in-context，反之，当它们不在同一个进程/线程中时，称为out-of-context）

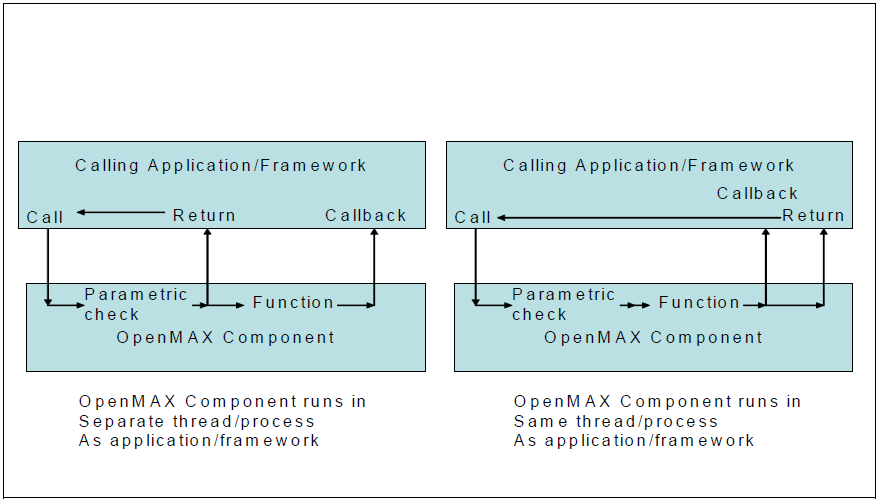


图 2-5. Out-of-Context VS In-Context 操作

与组件的数据通信始终指向特定的组件端口。每个端口都有一个组件规定的分配或使用buffer的最少数量。每个端口通过缓冲区头部来把所有缓冲区关联起来，缓冲区头部引用缓冲区中的数据并提供与缓冲区内容相关的元数据。每个端口应能够分配自己的缓冲区或使用预先分配的缓冲区；通常会选择其中一种最有效率的方式。

### 2.1.7 隧道缓冲区分配

本节介绍隧道化组件的缓冲区分配。对于给定的隧道，只有一个端口提供缓冲区并将这些缓冲区传递给其他端口。通常，隧道的buffer供应端口也会分配缓冲区。但是，在正确的情况下，隧道组件可能会选择再使用其他端口的buffer，以避免内存拷贝并优化内存使用。这个可选的方法，称为内存共享，在第10节 -实现缓冲区共享中有详细描述。

图2-6说明了与隧道缓冲区分配相关的概念。

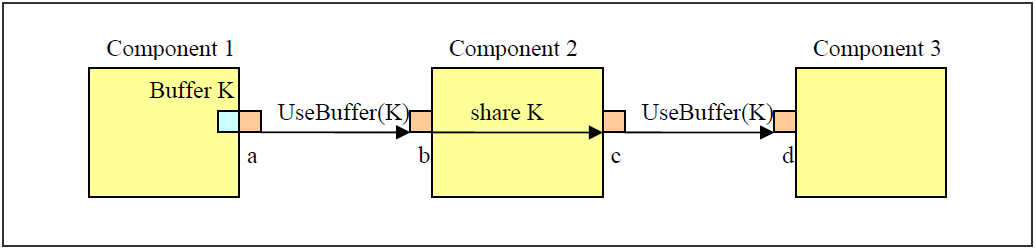


图2-6 缓冲区分配和共享关系示例

在这些处于隧道中的端口中，调用UseBuffer的来向其邻居端口传递buffer的端口被称为供应端口。供应端口不一定分配缓冲区，它可能会从同一个组件的另一个端口上再使用缓冲区。图2-6中的端口a和c说明了供应端口。

接收来自其邻居UseBuffer调用的端口称为非供应端口。 端口b和d图2-6显示了非供应商端口。

端口的*隧道端口*是与之共享隧道的端口。例如，图2-6中的端口b是端口a的隧道端口。同样，端口a是端口b的隧道端口。

*分配器端口*也会为其自己分配缓冲区。 图2-6中的端口a是唯一的分配器端口。

*共享端口*是再使用同一组件上另一个端口缓冲区的端口。 例如，图2-6中的端口c是共享端口。

*隧道组件*是使用至少一个隧道的组件。

端口所需的一组缓冲区信息包括缓冲区数量和每个缓冲区所需的大小。如果需要多组缓冲区的话，缓冲区的最大数量和大小，是这些组中的最大值。一个端口通过OMX\_GetParameter函数来从它的隧道端口上检索缓冲区要求，检索到的数据保存在OMX\_PARAM\_PORTDEFINITIONTYPE结构体中。需要注意的是，一个端口可能会从共享其缓冲区的端口来确定其缓冲区要求，而不使用OMX\_GetParameter调用，因为它们都包含在同一个组件中。

无论组件是否共享缓冲区，它都有义务遵守以下外部语义：

• 在供应商端口上提供缓冲区。

• 准确地传达其端口上的缓冲区要求。

• 使用OMX\_EmptyThisBuffer调用将缓冲区从输出端口传递到输入端口。

• 使用OMX\_FillThisBuffer调用将缓冲区从输入端口返回到输出端口。

#### 2.1.7.1 IL客户端组件设置

要设置隧道组件，IL客户端应按以下顺序操作：

1. 加载所有隧道组件，并在这些组件上设置隧道。

2. 命令所有隧道组件从LOADED状态转换到IDLE状态。

需要注意的是，如果某些组件在共享缓冲区时IL客户端不以这种方式运行，则由于组件之间可能存在依赖关系，因此隧道组件可能永远不会转换到IDLE状态。

#### 2.1.7.2组件从LOADED状态转换到IDLE状态

当组件从LOADED状态转换到IDLE状态时，非共享组件的每个供应商端口执行以下操作：

1. 通过OMX\_GetParameter调用确定隧道端口的缓冲区要求。

2. 按照自己要求和隧道端口要求的最大值分配缓冲区。

3. 在其隧道端口上调用OMX\_UseBuffer。

### 2.1.8 端口重新连接

端口重新连接可使已经隧道化的组件替换为另一个隧道组件，而不必拆除周围的组件。在图2-7中，组件B1将被替换为组件B2。为此，组件A的输出端口和组件B的输入端口应首先使用端口禁用命令禁用，一旦所有分配的缓冲区都返回到其合法所有者并释放后，组件A的输出端口就可以连接到组件B2了。组件B1的输出端口和组件C的输入端口应同样给予端口禁用命令，在所有分配的缓冲区已经返回到其所有者并释放后，组件C输入端口可以连接到组件B2输出端口。然后所有的端口都可以被赋予enable命令。有关端口禁用和启用的更多信息，请参见第3.4.4节“端口禁用和启用”。

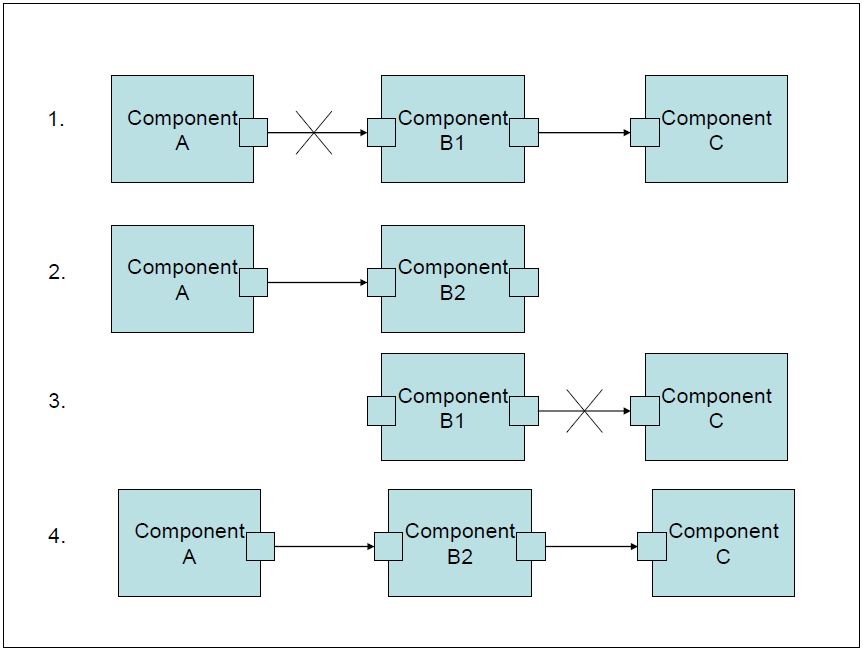


图 2-7. 端口重新连接

在某些情况下，如音频，可能需要将一个组件重新连接到另一个组件，然后淡入一个新组件的数据，同时淡化原始组件的数据。图2-8说明了这将如何工作。在步骤1中，组件A将数据发送到组件B1，然后组件B1将数据发送到组件C。组件A和C都有一个禁用的额外端口。在步骤2中，IL客户端首先在组件A和B2之间建立隧道，然后在B2和C之间建立隧道，然后启用两个隧道中的所有端口。假定这些是音频组件，组件C可能能够以各种增益混合来自组件B1和B2的数据。在步骤3中，组件A和组件C连接到组件B1的端口被禁用，组件B1的资源可能会被取消分配。

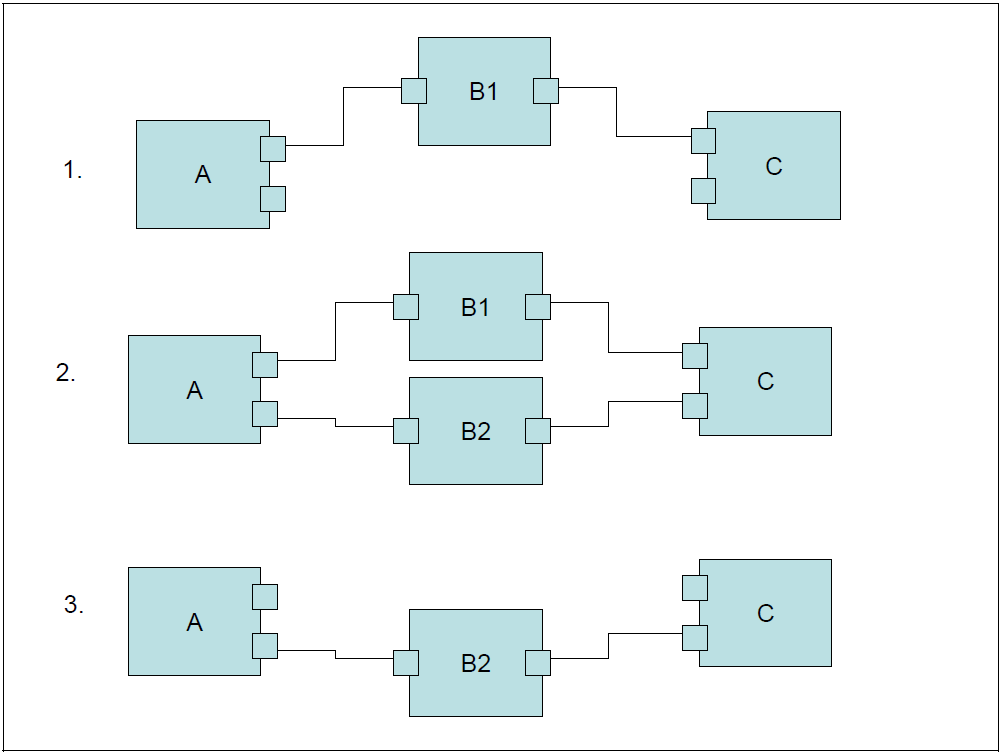


图 2-8. 重新连接组件

### 2.1.9 队列和Flush

命令队列使组件能够刷新尚未处理的缓冲区，并在使用非隧道通信时将这些缓冲区返回给IL客户端，或者在使用隧道通信时将这些缓冲区返回给隧道输出端口。例如，假定组件的输出端口正在使用由IL客户端分配的缓冲区。 在此示例中，客户端在发送flush命令之前向组件发送五个缓冲区。在处理flush命令后，组件返回尚未处理的缓冲区并触发其事件处理程序来通知IL客户端。在处理flush命令之前，已经处理了两个缓冲区，该组件返回未填充的剩余三个缓冲区并生成一个事件。IL客户端在反初始化组件之前，应该等待事件。

### 2.1.10 标记缓冲区

当遇到被标记的缓冲区时，IL客户端也可以触发事件产生。一个缓冲区可以在其buffer header中被标记，该标记buffer在内部通过一系列OpenMAX IL组件从输入缓冲区传输到输出缓冲区。该标记使组件能够在遇到标记的缓冲区时向IL客户端发送事件。 图2-9描述了这是如何工作的。

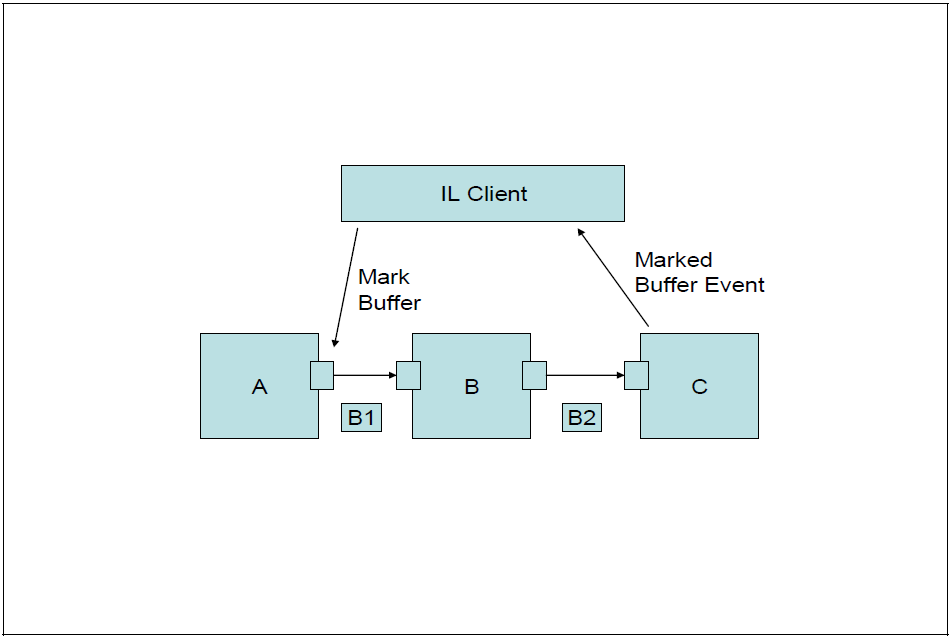


图 2-9. 标记缓冲区

IL客户端发送一个命令来标记缓冲区。从组件的输出端口发送的下一个缓冲区标记为B1。组件B处理B1缓冲区并将结果与标记一起提供给缓冲区B2。当组件C通过其输入端口接收标记的缓冲区B2时，处理该缓冲区，当它处理完后，触发其事件处理程序。

### 2.1.11事件和回调函数

组件可以向IL客户端发送六种事件：

• 错误事件：是一个枚举值并可能随时发生

• 命令完成通知事件：成功执行命令后会触发命令完成通知事件。

• 标记缓冲区事件：在组件检测到标记缓冲区时触发。

• 端口设置更改通知事件：在组件更改端口设置时，会生成端口设置更改通知事件。

• 缓冲标志事件：遇到流结束时触发缓冲标志事件。

• 资源获取事件：组件获取到其等待的资源时会生成资源获取事件。

端口在缓冲区可用时执行缓冲区处理回调函数或指示需要缓冲区。

### 2.1.12 缓冲区负载

端口配置确定和定义数据在组件端口上传输的格式，但并未定义数据在缓冲区中的存放方式。

通常有三种情况描述了数据是如何填充在缓冲区中的，每种情况都有其优劣势。

在所有情况下，缓冲区中有效数据的范围和位置由buffer header中的pBuffer，nOffset和nFilledLen参数定义。pBuffer参数指向缓冲区的开始，nOffset参数指示从缓冲区开始到有效数据开始之间的字节数，nFilledLen参数指定缓冲区中有效数据的连续字节数。 因此，缓冲区中的有效数据位于pBuffer + nOffset到pBuffer + nOffset + nFilledLen的范围内。

以下情况代表在解码或编码时传入或传出组件的缓冲区中的压缩数据。在所有情况下，缓冲区仅为数据提供传输机制，对内容没有特别的要求。 内容需求由端口配置参数定义。

缓冲区的阴影部分表示数据，白色部分表示没有数据。

案例1：每个缓冲区全部或部分填充。 在包含压缩数据帧的缓冲区的情况下，帧由f1至fn表示。

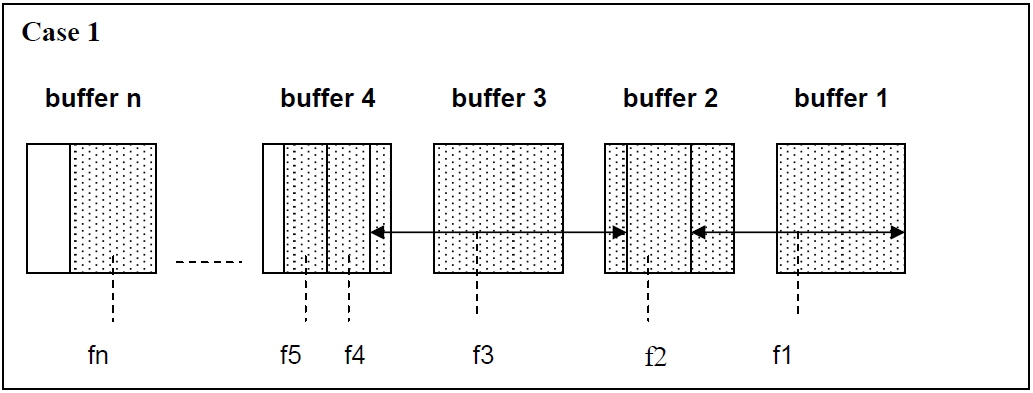


图2-10：案例1 - 每个缓冲区全部或部分填充

案例1为解码播放提供了一个好的案例。缓冲区可以容纳多个帧，可以减少缓冲一定数量的解码所需的事务数量。 然而，这种情况可能需要解码器在解码帧时解析数据，它还可能需要解码器组件具有一个以帧为单位建立的缓冲区，在该帧缓冲区中放置解析的数据或保存将在下一个缓冲区才能解析完成的部分帧。

案例2：每个缓冲区只填充完整的压缩数据帧。

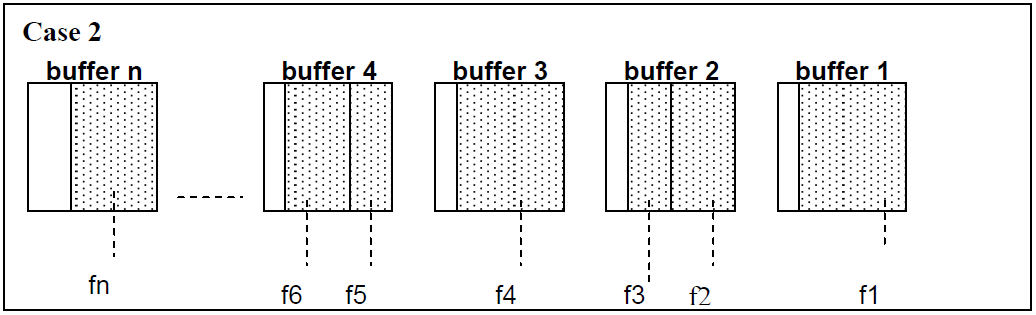


图2-11：案例2 - 每个缓冲区中只填充完整的数据帧

案例2与案例1不同，因为它首先需要解析压缩数据，只有完整的帧才被放入缓冲区。案例2也可能需要解码器组件解析数据以进行解码,这种情况可能不需要额外的工作缓冲区来解析案例1中所需的帧。

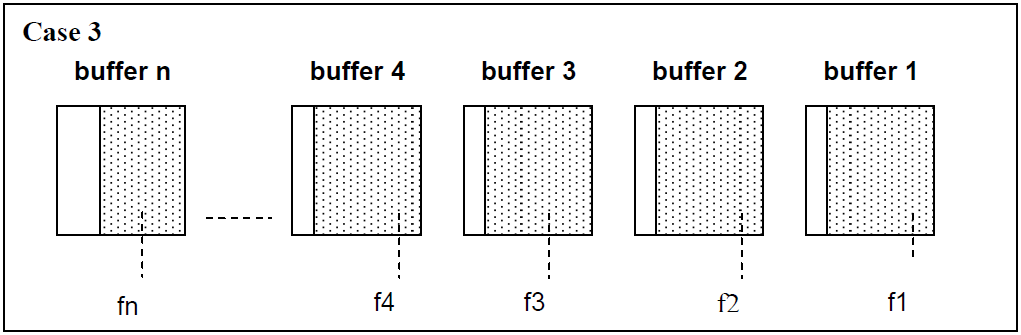
案例3：每个缓冲区只填充一帧压缩数据。

图2-12：案例3 - 每个缓冲区只填充一帧压缩数据

案例3中的好处是解码组件不必解析数据，而源组件需要解析。但是，这种方法会造成数据传输的瓶颈，数据传输被限制为每次传输一帧。根据实现情况，每帧只处理一个事务可能比从缓冲区中解析帧会对性能造成更大的影响。

解码器或编码器组件至少需要支持案例1。根据定义，如果编解码器组件可以支持案例1，则它也可以支持案例2和3，但是是在压缩格式允许字节对齐帧边界的情况下。例如，配置一个自适应多速率（AMR）编解码器在RTP载荷格式，带宽效率模式下时，案例2或3可能就没有意义。因为这种格式是非字节对齐的帧。

当用压缩数据填充缓冲区用以输入到解码器或来自编码器的输出时，如果帧不是字节对齐的话，可能会出现非完整帧填充受限的问题。为解决这个问题，就需要在格式规范以外添加额外的数据,而这些填充数据需要删除。 这需要知道哪些是标准规范之外的填充位。同样，如果此填充不能保持符合端口配置的标准规范，则完整的帧不能始终放置在缓冲区中。在任何一种情况下，都需要具体了解这种情况如何处理，并且可组件之间可能有所不同。

为了实现互操作性，缓冲区中提供的内容不应被假定或要求为任何数量的完整帧，尽管至少有一个完整的数据单元将在未压缩数据格式的缓冲区中提供。压缩数据格式不会限制每个缓冲区中传递的内容数量。

### 2.1.13 缓冲区标志和时间戳

缓冲区标志将某些属性（例如数据流的结束标志EOS）与包含在缓冲区中的数据相关联。缓冲区时间戳将微秒级的呈现时间与用于定时渲染该数据的缓冲区中的数据相关联。一旦时间戳与缓冲区相关联，除了时钟组件外，其他组件不应该改变用于速率控制或同步的时间戳。

缓冲区元数据（比如：标志和时间戳）存放于缓冲区中的[第一个]新逻辑单元。考虑到缓冲区中存在多个逻辑单元，元数据将存放于缓冲区中首先出现的逻辑单元。 [缓冲区中的后续逻辑单元没有明确的标志或时间戳，如果每个逻辑单元都需要显式标志和时间戳，则应在每个缓冲区中包含一个或更少的逻辑单元]。 除非另有说明（例如，在标志定义中），接收标有标志或时间戳的逻辑输入单元的组件应当将该元数据从输入逻辑单元复制到所有逻辑输出单元。

### 2.1.14 同步

通过在时钟组件上使用同步（sync）端口来启用同步。这些端口和时钟组件在“other”域中定义，并使用和数据端口相同的协议和调用进行操作。时钟组件保持着一个媒体时钟，该媒体时钟基于音频和视频的参考时钟来跟踪媒体流中的位置。时钟组件通过同步端口将包含时间信息（表示媒体时间更新并包含媒体时钟的当前位置，缩放和状态）的缓冲区传输到客户端组件。客户端组件可以请求时钟组件在与媒体时钟相匹配时发送该时间戳来将操作（例如，视频帧的呈现）盖上时间戳。在这种情况下，客户端组件在通过其同步端口接收到请求的实现时执行该操作（盖上时间戳）。图2-13显示了组件配置示例中的时间和数据缓冲区的流程。

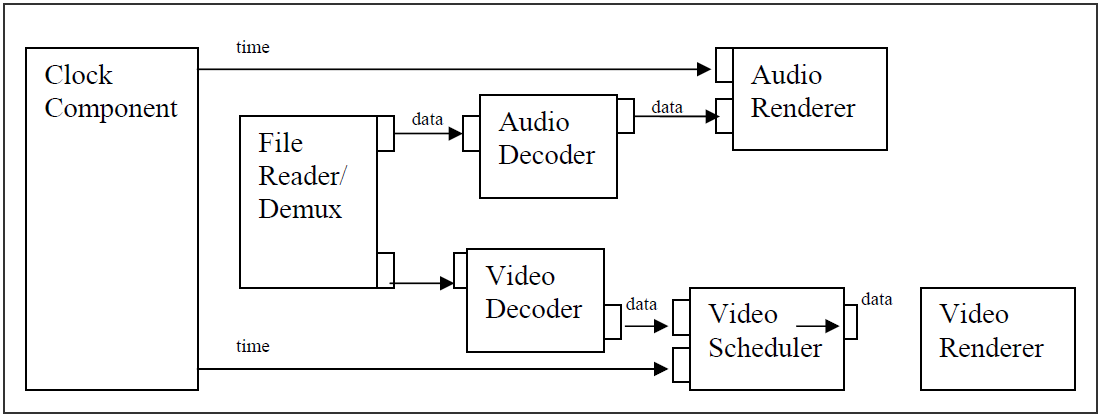


图2-13.时间流和数据缓冲区

### 2.1.15 速率控制

时钟组件还通过公开一组用于控制其媒体时钟的配置来实现速率控制。IL客户端可以通过改变媒体时钟的比例因子（有效地改变媒体时钟前进的速度和方向）来实现播放，快进，快退，暂停和慢动作特技模式。IL客户端也可以通过使用这些配置来更改媒体时钟的状态，从而启动和停止时钟。时钟组件通过在所有同步端口上发送带有新比例/状态的媒体时间，使其所有客户端组件都能够意识到媒体时钟比例和状态的变化。虽然组件可能不会因为缩放比例更改而更改缓冲区的时间戳，但组件可能会相应地更改其处理方式。例如，音频组件可能会在特技模式期间缩放和调整音频，或者完全停止输出。

### 2.1.16 组件的注册

如何将组件注册到Core里面，通常是由Core特定的。

但是，如果Core支持与组件的静态链接，那么它将会支持第3节中描述的标准编译时组件的注册机制。因此，供应商可以提供适用于静态链接的所有组件;这是通过将组件信息放置到链接组件与核心的数据结构中来实现的。

一个组件可以使用这种机制静态注册，但其代码的大部分是动态加载的。

组件提供一个接口来检索它支持的所有标准组件角色。核心可以利用这个接口向IL客户端公开与角色相关的信息。

### 2.1.17 资源管理

本节讨论资源管理在OpenMAX IL API中的作用。

#### 2.1.17.1 资源管理的必要性

当组件由于缺乏资源而不能进入空闲状态时，IL客户端不知道缺乏的资源是什么或者哪些组件正在使用该资源。因此，IL客户端也就无法解决资源冲突问题，这些情况下就需要IL资源管理。

OpenMAX IL的目标之一是由IL层对其上层的硬件独立性。硬件独立性的目标可以通过指定资源管理的以下要求来实现：

• IL客户端（通常是软件平台的一部分的多媒体插件）不需要知道IL实现的细节或IL组件正在使用的资源。

• 在资源冲突的情况下，IL客户端应该能够依靠IL的实现和硬件平台做出一致性行为。

• IL客户端不应直接与硬件供应商特定的资源管理器进行交互，原因有两个：

o 这种方法违反了硬件独立性的目标。

o 此方法为IL客户端增加了大量重复性工作，同时也会对IL客户端在多个硬件平台上的复用性产生影响。

虽然资源管理在OpenMAX IL API版本1.1中未得到充分解决，但资源管理的“钩子”遍布行为规则，组件优先级和资源管理相关组件状态等等。这些“钩子”为OpenMAX IL API的后续版本的全面资源管理奠定了基础。

在继续进行之前，先来看看资源管理和策略的定义：

• *资源管理*负责管理组件对有限资源的访问。资源管理器将知道特定资源有多少可用，哪些组件正在使用该资源，以及组件正在使用多少资源。资源管理者将根据资源冲突和可用性策略来决定哪些组件应该被抢占或恢复。

• *策略*负责管理组件链或流。 策略会根据资源，系统配置和其他因素在内的信息确定流是否可以运行。

#### 2.1.17.2 架构示例

图2-15显示了一个基于OpenMAX IL系统的高级体系架构图。这个例子中，在应用程序和IL层之间存在一个带有策略管理器的多媒体框架。该系统还具有多个硬件平台，这些平台被不同的OpenMAX IL组件使用，并由多个硬件供应商特定的资源管理器进行管理，但是这个系统对单一的集中式资源管理器来说也是一样的。

本示例体系结构用作以下关于组件优先级，行为规则和特定于硬件的资源管理器讨论的背景。然而，需要注意的是，该讨论适用于任何基于OpenMAX IL的架构。

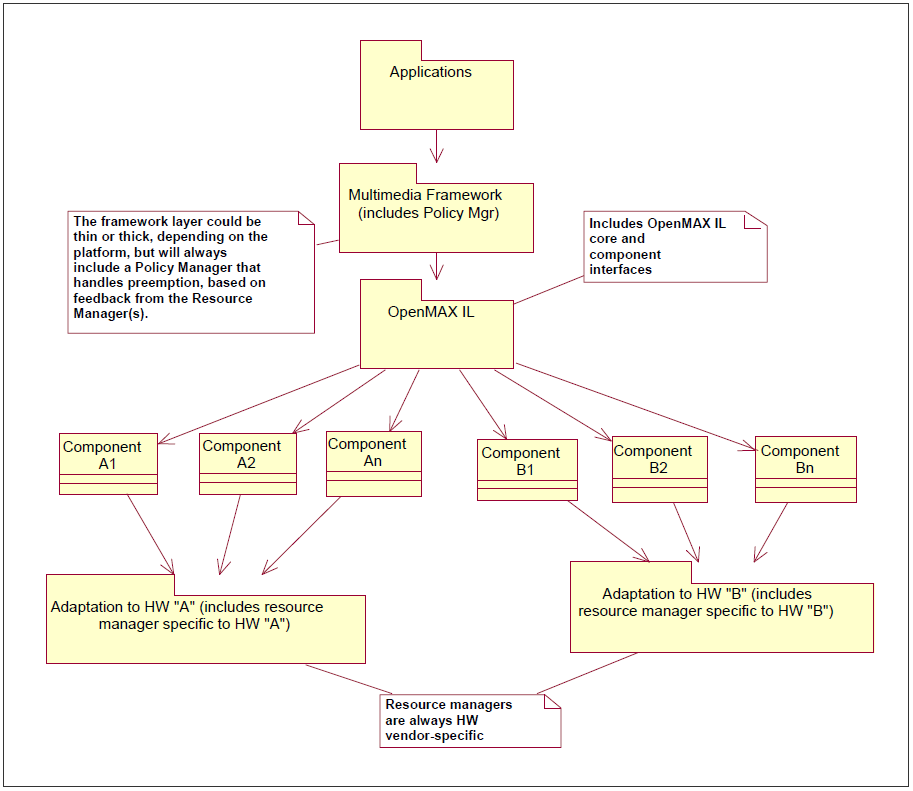


图 2-14. 示例架构

为了确保在资源冲突情况下组件仍会有一致性的行为，需要对组件优先级和行为规则制定一套通用的定义。

#### 2.1.17.3 组件优先级

每个IL组件都有一个IL客户端设置的优先级值（一个OMX\_U32类型的整数）。

采用降序排列的优先级，0表示最高优先级。下面的平局规则也适用：当比较具有相同优先级的组件时，最近获得资源的组件比拥有更长时间资源的组件具有更高的优先级。IL组件也可以由IL客户端分配组优先级，同一组ID的组件都具有相同的组优先级。

#### 2.1.17.4 行为规则

以下行为在IL层上定义：

• OMX\_ErrorInsufficientResources错误只会在组件尝试进入IDLE状态时资源不足，或资源充足却被低优先级组件提前占用从而无法释放足够资源的情况下发生。

• 组件在尝试进入空闲状态时不知道发生抢占，所需的资源需要由较低优先级的组件来释放。

• 当某个组件具有需要被抢占的资源时，它会在从Executing或Paused状态转换到Idle状态时向IL客户端发送OMX\_ErrorResourcesPreempted错误。一旦释放资源，该组件将会在从Idle状态移动到Loaded状态时，向IL客户端发送OMX\_ErrorResourcesLost错误。

• 如果IL客户端想要知道与组件关联的流何时可以恢复或启动，则IL客户端应在资源可用时请求通知。这通过将组件置于OMX\_StateWaitForResources状态来实现。 当资源可用时，组件自动进入Idle状态。当客户端收到组件处于Idle状态的通知时，它可以尝试将该链中的其余组件也置于Idle状态。这种自动移动到Idle状态可确保在多个IL客户端正在等待相同资源的情况下，只要资源可用，IL客户端就可以恢复或启动流。如果组件只是自动移动到Loaded状态，那么另一个IL客户端可能会首先获取该资源。

这些行为规则旨在涵盖IL客户端和IL组件之间的交互。

#### 2.1.17.5 特定于硬件供应商的资源管理器

要实现行为规则，硬件供应商特定的资源管理器可能存在并执行以下功能：

• 实施和管理等待队列。

• 跟踪可用资源。

• 跟踪具有资源和使用哪些资源的每个组件。

• 通知一个组件或多个组件，当有较高优先级的组件请求该资源时，它们需要放弃其资源。

• 在资源可用时通知等待资源的最高优先级组件。

组件与硬件供应商特定的资源管理器之间的实际交互是特定于供应商的，并且超出了本文的范围。 第3节披露了与优先级和资源管理相关的参数结构和用例的更多细节。

#### 2.1.17.6 组件挂起

当组件缺乏足够的资源来处理数据时，它可能会选择暂停自己作为启用更优化的动态资源管理的手段。 组件暂停解决了两个用例：

1. 组件失去了重要资源，而该资源缺失本质上是暂时的。

2. 必要资源的动态分配失败

在没有挂起功能的情况下，组件对抢占和资源丢失的唯一反应是通过转换到Idle状态然后到Loaded状态来进行反初始化。 这种反初始化会导致数据流的状态丢失，因为缓冲区必须返回到其分配器。挂起允许组件保持其状态，以便在延迟一段时间后从暂停点恢复。

当组件处于Idle或Pause状态时，挂起是组件的一个属性。特别是，当一个组件在丢失一个或多个资源从而会阻止其处理数据时，就会“挂起”。这意味着组件不能被挂起并同时处于Executing状态（因为“executing”意味着组件的数据是可用的，并在处理或输出数据）。因此，一个组件在通常在持有一些资源但不处理数据时可以挂起，也就是说当组件处于Idle状态或Pause状态时。

组件挂起不需要新的组件状态，只是添加了一个新的组件启动的状态，即从Executing到Pause的转换，执行组件在挂起时对自身执行的转换。IL客户端可以执行挂起组件上的任何正常状态转换，但存在以下例外情况：客户端可能不会将挂起的组件转换为执行状态。任何尝试这样做都会失败并返回OMX\_ErrorComponentSuspended错误。

### 2.1.18 内容管道

IL组件可以利用内容管道来从源或目的地（例如，本地文件，远程文件，广播等）同步拉入或推出内容（例如文件流）。内容管道是通过实现内容管道结构中定义的数据访问抽象接口来提供内容访问的对象。

内容管道接口包括用于常规内容操作的功能，包括：

• 打开，关闭和创建content

• seek content中的特定位置

• 获取content中的位置

• 从当前位置读取数据

• 将数据写入当前位置

该内容管道接口还包括适应内容管道实现的功能，比如将数据异步流入或从远程位置流出。在这种情况下，管道可能不会立即准备提供数据（在读取的情况下）或接受数据（在写入的情况下）。此外，这些管道可以维护他们自己的数据缓存。 这些功能支持：

•检查管道的可用字节（传入或传出）以验证管道客户端可能会执行的后续读取或写入操作。

•通过管道提供的数据缓冲区读或写数据，以避免管道缓冲区和客户端缓冲区之间不必要的复制。

组件通过OMX\_GetContentPipe函数从IL Core获取管道，从而能够使用内容管道（例如容器分离器或复用器）。 或者，IL客户端可以通过OMX\_IndexParamCustomContentPipe配置提供定制的内容管道（例如，如果客户端实现内容管道本身）。 IL客户端通过OMX\_IndexParamContentURI参数将目标内容指定为URI。

### 2.1.19 文件解析

OpenMAX IL 1.1 定义了标准容器格式分离器以及便于组件中的文件分析机制。包括：

• 用于指示组件是否成功检测到并支持所给的数据流格式。

• 用于检查并选择每个组件输出端口上可用的流（当存在多个流时）。

• 用于IL客户端遍历，提取和过滤组件从数据流中捕获的元数据。

### 2.1.20 视频解码器错误的映射

视频解码器组件具有通知IL客户端在解码流时遇到的任何macroblock（MB）错误的能力。客户端可以通过专用参数随时查询它遇到的MB错误的映射。

此功能的一个用途是视频电话，其中一端的视频终端为远程视频终端生成码流。码流在传输过程中可能会损坏，导致远程终端接收并解码时发生MB错误。应用程序可以在解码终端提取MB错误图，并将其发送给编码终端，从而允许其在随后的编码帧中用内部MB error刷新宏块。

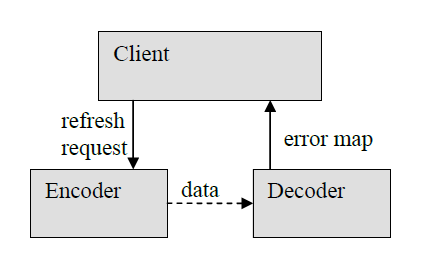


图2-15. 错误的映射示例

### 2.1.21 缓冲区有效负载附加信息

根据缓冲区负载类型和组件需求，可能会出现需要将附加信息添加到缓冲区的末端以在下一个组件内进一步处理缓冲区有效负载内容。

举例来说，视频解块算法需要MB级量化信息以便对视频内容进行解块处理。附加缓冲区有效负载信息应通过缓冲区头部结构中的“extra data”缓冲区标志来识别，这在3.1.2.7节 - OMX\_BUFFERHEADERTYPE中有描述。

这个附加的缓冲区负载信息应用于缓冲区中的第一个新的逻辑单元。 因此，如果在缓冲区中存在多个逻辑单元，则“extra data”标志存放于其起始边界首先出现在缓冲区中的逻辑单元。缓冲区中的后续逻辑单元没有明确的“extra data”。如果每个逻辑单元都需要有明确的“extra data”，则每个缓冲区中应包含一个或更少的逻辑单元。

#### 2.1.21.1 缓冲区数据格式

当存在额外数据时，数据属性（如类型和大小）由对应的数据结构标识，紧跟在缓冲区有效负载之后并位于实际数据之前。多种类型的额外数据可作为一系列块对（支持数据结构和实际数据）附加到常规有效负载的末尾，要终止这个额外数据部分的列表，应在缓冲区中包含额外的数据结构，用以指示这是终止。更多详细信息，请参见第4.2.33节。

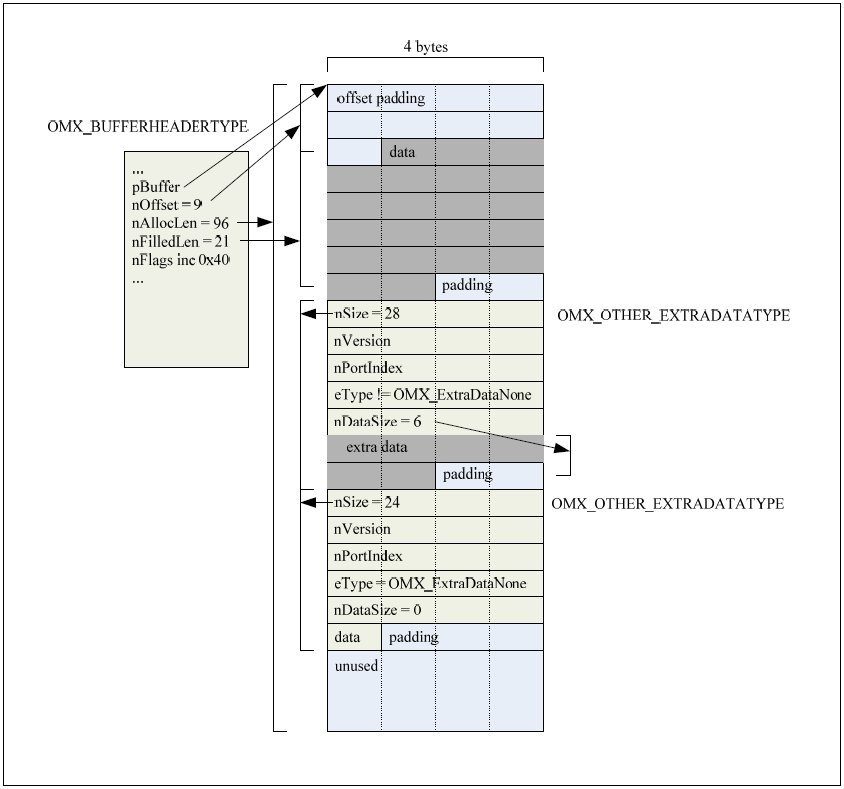


图2-16. 缓冲区额外数据的格式

## 2.2 字节序

OpenMAX IL API数据结构的字节序应服从运行IL客户端的平台的字节序，包括IL客户端使用的接口和组件之间的接口（例如，专门在两个隧道组件之间执行的函数）。OpenMAX IL负责实现其字节序转换;任何此类转换对于IL客户端和与IL客户端使用相同字节序的组件都是透明的。

# 3 OpenMAX Integration Layer Control API

OpenMAX IL API允许IL客户端控制audio, video和 image域中的多媒体组件,还包含“other”域提供的额外功能，例如音视频（A/V）同步。OpenMAX IL API的用户通常是一个多媒体框架。 在本文的其余部分，OpenMAX IL API的用户将被称为IL Client。

OpenMAX IL API是在一组头文件中定义的，即：

• OMX\_Types.h：OpenMAX IL中使用的数据类型

• OMX\_Core.h：OpenMAX IL核心API

• OMX\_Component.h：OpenMAX IL组件API

• OMX\_Audio.h：OpenMAX IL音频域数据结构

• OMX\_IVCommon.h：图像和视频域通用的OpenMAX IL结构

• OMX\_Video.h：OpenMAX IL视频域数据结构

• OMX\_Image.h：OpenMAX IL图像域数据结构

• OMX\_Other.h：OpenMAX IL其他域数据结构（包括A/V同步）

• OMX\_Index.h：所有OpenMAX IL定义的数据结构索引

• OMX\_ContentPipe.h：内容管道定义

本节介绍如何配置OpenMAX IL内核和OpenMAX IL组件以进行操作。

首先介绍OpenMAX IL数据类型,接下来，描述OpenMAXIL Core的方法。组件实现的方法在3.2.3节中讨论。最后，3.4节描述了一些有意义的操作的调用序列，包括组件初始化，正常数据流，数据隧道设置和存在数据隧道的数据流，序列图旨在描述IL客户端，IL Core和OpenMAX IL组件之间的动态相互作用。当记录函数时，以下约定适用于函数参数：

• <param\_name> [in]指定一个输入参数，该参数由调用函数的程序设置并在函数的实现中读取。

•<param\_name> [out]指定一个输出参数，该参数在函数的实现中设置并返回给调用者。当函数返回时，调用者可以读取参数的新值，通常通过引用传递过来。

• <param\_name> [inout]指定一个输入/输出参数。函数的调用者可以设置该值，而函数的实现可以在返回给函数调用者之前修改该参数。

这些参数分类也可以在OpenMAX IL头文件中找到，其中定义了空的宏OMX\_IN，OMX\_OUT和OMX\_INOUT。OMX\_IN对应于函数参数<param\_name> [in]。OMX\_OUT对应于函数参数<param\_name> [out]，而OMX\_INOUT对应于函数参数<param\_name> [inout]。

（举例：#define OMX\_IN ）

## 3.1 OpenMAX IL 类型

### 3.1.1 枚举

OMX\_Core.h中定义了五个32位整数枚举：

• OMX\_ERRORTYPE 是各个函数的返回值（请参阅第3.1.1.3节）。

• OMX\_COMMANDTYPE 包含了IL客户端可能发送给OpenMAX IL组件的命令（请参阅第3.1.1.1节）。

• OMX\_EVENTTYPE 包含了可在OpenMAX IL组件内部生成的事件，并通过回调函数传递给IL客户端（请参阅第3.1.1.4节）。

• OMX\_BUFFERSUPPLIERTYPE 包含了在隧道端口下的所有buffer供应者的可能性。这个枚举类型的使用描述可以在3.1.1.5节中找到。

• OMX\_STATETYPE 在3.1.1.2节中描述。

#### 3.1.1.1 OMX\_COMMANDTYPE

表3-1表示了IL客户端可能发送给OpenMAX IL组件的命令。由于命令是非阻塞的，当命令完成时，OpenMAX IL组件会通过回调函数发送命令完成事件，回调是在专门的结构中定义的; 见3.1.2.8节。

表 3-1: OpenMAX IL 命令

|  |  |
| --- | --- |
| **Field Name 字段名称** | **Description 描述** |
| OMX\_CommandStateSet | 改变组件的状态 |
| OMX\_CommandFlush | 冲刷组件端口上的缓冲区队列 |
| OMX\_CommandPortDisable | 禁用组件上的某一个端口 |
| OMX\_CommandPortEnable | 启用组件上的某一个端口 |
| OMX\_CommandMarkBuffer | 标记缓冲区并指定哪个组件将引发收到的事件标记 |

表3-2描述了用于每个命令的参数。

表3-2：命令语法

|  |  |  |
| --- | --- | --- |
| **Command code** | **nParam** | **pCmdData** |
| OMX\_CommandStateSet | OMX\_STATETYPE – 将要转换的状态 | NULL |
| OMX\_CommandFlush | OMX\_U32 – 目标端口的 ID | NULL |
| OMX\_CommandPortDisable | OMX\_U32 – 目标端口的 ID | NULL |
| OMX\_CommandPortEnable | OMX\_U32 – 目标端口的 ID | NULL |
| OMX\_CommandMarkBuffer | OMX\_U32 – 目标端口的 ID | OMX\_MARKTYPE\* - 标记的数据和目标组件 |

#### 3.1.1.2 OMX\_STATETYPE

图3-1说明了由于IL客户端调用OMX\_SendCommand（OMX\_StateSet，<state>）而导致的状态转换，其中组件的新状态作为一个参数传递。由“<”和“>”括起来的名称并不是由IL客户端发送的命令触发的，而是由内部组件一系列事件引起的。

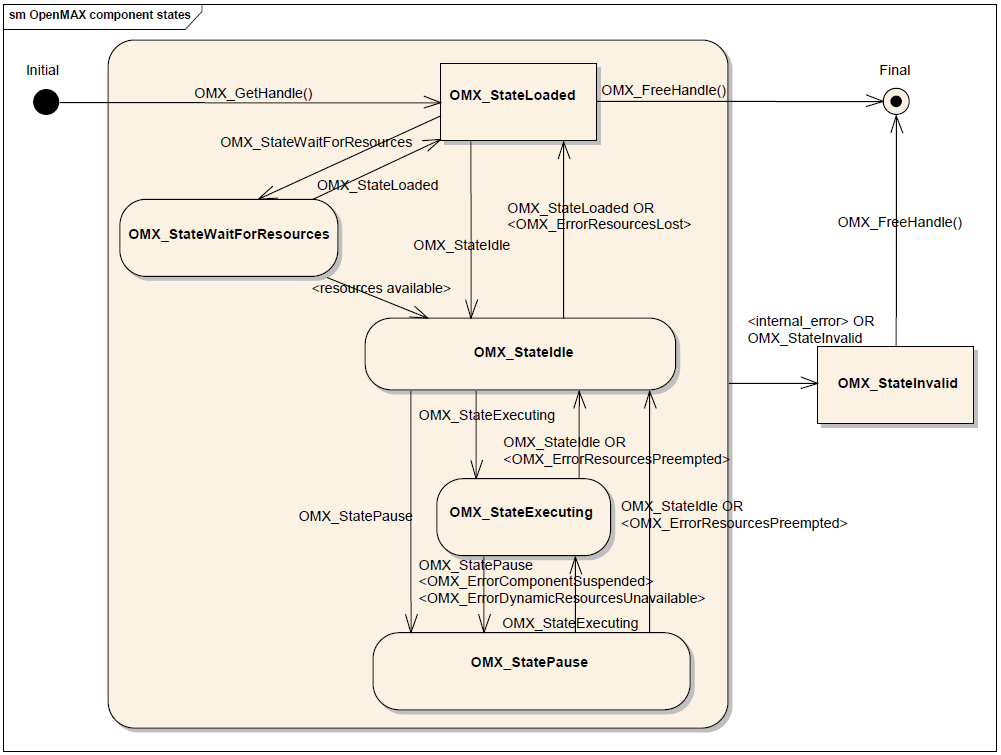


图3-1. OpenMAX IL组件状态转换

本节介绍组件的状态。IL客户端使用OMX\_SendCommand函数发送OMX\_CommandStateSet命令来使组件改变状态。

表3-3 表示OpenMAX IL组件的状态

表 3-3: OpenMAX IL 组件状态

|  |  |  |  |
| --- | --- | --- | --- |
| Field Name | 描述 | 静态资源分配 | Location of buffer |
| OMX\_StateInvalid | 组件损坏或遇到无法恢复的错误。 | Unknown | Unknown |
| OMX\_StateLoaded | 组件已被加载但没有分配资源。 | No | Not available |
| OMX\_StateIdle | 组件具有所有资源，但尚未传输任何缓冲区或尚未开始处理数据。 | Yes | Supplier only |
| OMX\_StateExecuting | 组件正在传输缓冲区并正在处理数据（如果数据可用）。 | Yes | Supplier or non-supplier |
| OMX\_StatePause | 组件数据处理已暂停，但可能会从暂停点恢复。 | Yes | Supplier or non-supplier |
| OMX\_StateWaitFor Resources | 组件正在等待资源变为可用。 | No | Not available |

##### 3.1.1.2.1 OMX\_StateLoaded

在通过OMX\_GetHandle创建并在分配给其资源之前，组件处于OMX\_StateLoaded状态。 在此状态下，IL客户端可以通过OMX\_SetParameter修改组件的参数，使用OMX\_SetupTunnel在组件端口上设置数据通道，或者将组件转换为OMX\_StateIdle状态或OMX\_StateWaitForResources状态。

例如，如果组件在试图转换到OMX\_StateIdle状态时未能获取其所有静态资源，IL客户端可能会选择将当前处于OMX\_StateLoaded状态的组件转换为OMX\_StateWaitForResources状态。

###### 3.1.1.2.1.1 OMX\_StateLoaded to OMX\_StateIdle

如果IL客户端请求把组件从OMX\_StateLoaded转换到OMX\_StateIdle状态，那么在完成转换之前，组件应获取其所有静态资源，包括为所有已启用的端口分配缓冲区。该组件不会为任何禁用的端口分配缓冲区。此外，在转换完成之前，缓冲区供应者（没有隧道连接时总是IL客户端）应确保非供应者拥有其所有缓冲区。

对于连接到IL客户端的端口，IL客户端可以自己分配缓冲区，然后通过端口上的OMX\_UseBuffer调用将它们传递给端口，或者可以指示端口通过OMX\_AllocateBuffer执行分配。对于每个端口，IL客户端应专门使用OMX\_UseBuffer或OMX\_AllocateBuffer。

当一个端口进行隧道传输时，供应端口可以自己分配缓冲区，或者如果端口实现缓冲区共享，则从同一个组件的端口再使用缓冲区。隧道的非供应者端口通过OMX\_UseBuffer来从供应者端口获取缓冲区。

端口使用的缓冲区数目在其端口定义中指定（参见OMX\_IndexParamPortDefinition），默认值为最小值（在相同结构中指定），但供应者可以在调用OMX\_UseBuffer和OMX\_AllocateBuffer之前通过OMX\_SetParameter函数修改。

##### 3.1.1.2.2 OMX\_StateIdle

在OMX\_StateIdle状态下，组件已准备好使用，这意味着所有必需的静态资源都已正确分配。但是，供应者保持着所有的缓冲区，不进行缓冲区交换或处理。 因此，如果该状态是从OMX\_StateExecuting或OMX\_StatePause状态进入的，那么该组件应该将它正在处理的所有缓冲区返回给它们各自的供应者。IL客户端可以将组件转换为除OMX\_StateInvalid和OMX\_StateWaitForResources状态以外的任何其他状态。

###### 3.1.1.2.2.1 OMX\_StateIdle to OMX\_StateLoaded

在从OMX\_StateIdle到OMX\_StateLoaded的转换中，缓冲区的供应者应该调用OMX\_FreeBuffer，而使非供应者端口释放缓冲区。如果供应者已经分配了缓冲区，则应在调用OMX\_FreeBuffer之前自己释放缓冲区。如果非供应者端口分配了缓冲区，它将在收到OMX\_FreeBuffer后释放缓冲区。此外，非供应者端口在收到OMX\_FreeBuffer时应该始终释放buffer header。当所有缓冲区都从组件中移除后，状态转换完成;该组件通过回调事件通知告诉OMX\_SendCommand已完成。

###### 3.1.1.2.2.2 OMX\_StateIdle to OMX\_StateExecuting

当组件被挂起时，不允许此类转换。如果IL客户端请求组件从OMX\_StateIdle到OMX\_StateExecuting的状态转换，并且组件未挂起，则组件应开始传输和处理数据。如果客户端在组件挂起时请求这个转换，该调用将失败并返回OMX\_ErrorComponentSuspended错误。对于与IL客户端通信的端口，IL客户端将通过OMX\_EmptyThisBuffer和OMX\_FillThisBuffer启动缓冲区传输。在隧道端口中，是供应者的输入端口应通过OMX\_FillThisBuffer将空缓冲区传输到隧道输出端口。

##### 3.1.1.2.3 OMX\_StateExecuting

在这种状态下，OpenMAX IL组件正在传输和处理数据; 组件处于该状态时不可以被挂起。组件应接受其输入端口上的OMX\_EmptyThisBuffer的调用和输出端口上的OMX\_FillThisBuffer的调用。任何与IL客户端进行通信的端口都会调用EmptyBufferDone和FillBufferDone回调函数，分别返回一个空的或满的缓冲区给IL客户端。任何隧道端口都应在其相应的隧道端口上调用OMX\_FillThisBuffer或OMX\_EmptyThisBuffer，以分别返回空或满缓冲区。IL客户端可以将处于OMX\_StateExecuting状态的组件转换为OMX\_StateIdle状态或OMX\_StatePause状态。

###### 3.1.1.2.3.1 OMX\_StateExecuting to OMX\_StateIdle

如果IL客户端请求从OMX\_StateExecuting到OMX\_StateIdle的状态转换，那么在完成转换之前，组件应将所有缓冲区返回给其各自的供应者并接收属于其供应者端口的所有缓冲区。与IL客户端通信的任何端口都应通过EmptyBufferDone和FillBufferDone回调函数返回它所持有的缓冲区，这些回调函数分别用于输入和输出端口。任何非供应者端口应分别使用OMX\_EmptyThisBuffer或OMX\_FillThisBuffer将其所持有的所有缓冲区返回到正在进行隧道传输的输入端口或输出端口。同样，任何供应者隧道端口都应该等待其所有缓冲区从其隧道端口返回。

###### 3.1.1.2.3.2 OMX\_StateExecuting to OMX\_StatePause

组件在以下三种情况之一发生从OMX\_StateExecuting状态到OMX\_StatePause状态的转换：

•客户端明确请求转换时；

•组件丢失执行所需的资源时；如果稍后可以重新获取资源，则会从资源丢失的地方恢复。在这种情况下，组件应自动执行转换并发出OMX\_ErrorResourcesSuspended错误事件。

•组件尝试分配动态资源失败时；在这种情况下，组件应自动执行转换并发出OMX\_ErrorDynamicResourcesUnavailable错误事件。

##### 3.1.1.2.4 OMX\_StatePause

在这种状态下，OpenMAX IL组件不会传输或处理数据，但缓冲区不一定会返回给其供应者。从OMX\_StatePause状态，可以直接转换到OMX\_StateExecuting继续执行而不丢弃数据。但是，如果客户端在组件挂起时请求此转换，那么组件将失败返回OMX\_ErrorResourcesSuspended错误。该组件仍然可以在其输入端接受数据缓冲区，但这些缓冲区只会加入队列而不会进一步处理。IL客户端可以将处于OMX\_StatePause状态的组件转换为OMX\_StateIdle或OMX\_StateExecuting。在从OMX\_StatePause到OMX\_StateIdle的转换中，组件应以与3.1.1.2.3.1节中描述的OMX\_StateExecuting-to-OMX\_StateIdle转换相同的方式将所有缓冲区返回给它们各自的供应者。

##### 3.1.1.2.5 OMX\_StateWaitForResources

在此状态下，组件正在等待一个或多个所需资源来使其变为可用。这种状态与资源管理有关。假设在平台上存在一个或多个特定于硬件的资源管理器来处理可用资源。OpenMAX IL组件和资源管理器之间的交互不在本规范的范围之内。

如果处于OMX\_StateLoaded状态的组件由于缓冲区之外的资源不足而无法进入OMX\_StateIdle状态，如果IL客户端希望在需要的资源可用时收到通知，那么IL客户端可能会将组件置于OMX\_StateWaitForResources状态。IL客户端可以通过将组件从OMX\_StateWaitForResources状态转换到OMX\_StateLoaded状态来命令组件停止等待资源。 如果OMX\_StateWaitForResources状态中的某个组件获取了它正在等待的所有资源，它将启动转换到OMX\_StateIdle状态。

###### 3.1.1.2.5.1 OMX\_StateWaitForResources to OMX\_StateIdle

当组件启动从OMX\_StateWaitForResources状态到OMX\_StateIdle状态的转换时，它应通过OMX\_EventResourcesAcquired事件将此转换告诉给IL客户端。当IL客户端收到OMX\_EventResourcesAcquired事件时，它将以从OMX\_StateLoaded过渡到OMX\_StateIdle的方式调用OMX\_UseBuffer和OMX\_AllocateBuffer。同样，组件在获取其所有静态资源（包括缓冲区）之前，无法完成向OMX\_StateIdle的转换。

##### 3.1.1.2.6 OMX\_StateInvalid

在此状态下，组件遭受内部损坏或无法恢复的错误。 当它检测到这种情况时，组件会自行转换为OMX\_StateInvalid，并通过生成OMX\_ErrorInvalidState的OMX\_EventError事件来通知IL客户端。 当IL客户端收到OMX\_EventError指示转换到OMX\_StateInvalid时，它应释放与该组件关联的所有资源，并最终调用OMX\_FreeHandle释放与该组件关联的句柄。

处于OMX\_StateInvalid状态的组件应该对其进行的每次调用都失败，并返回除OMX\_GetState，OMX\_FreeBuffer或OMX\_ComponentDeinit之外的OMX\_ErrorInvalidState错误消息。IL客户端也可以通过发送OMX\_SendCommand明确地将组件转换到OMX\_StateInvalid状态，组件可以在任何状态和OMX\_StateInvalid状态之间转换。

typedef struct OMX\_COMPONENTREGISTERTYPE

{

const char \* pName;

OMX\_COMPONENTINITTYPE pInitialize;

} OMX\_COMPONENTREGISTERTYPE;

typedef OMX\_ERRORTYPE (\* OMX\_COMPONENTINITTYPE)(OMX\_IN OMX\_HANDLETYPE hComponent);

第四章