

**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。

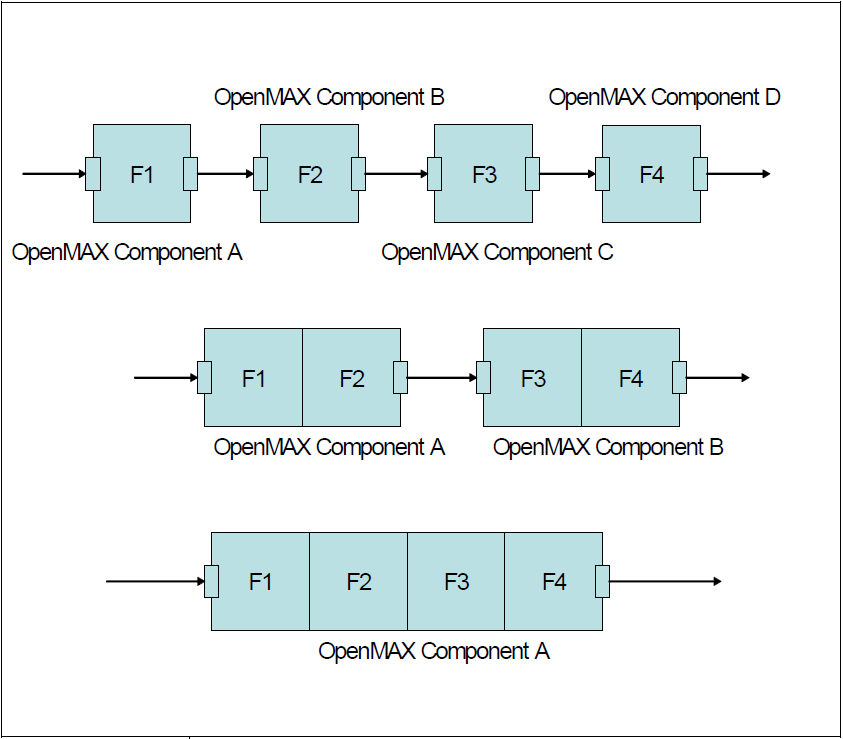


图2-1. OpenMAX IL实现的对于功能的一些可能分区

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。

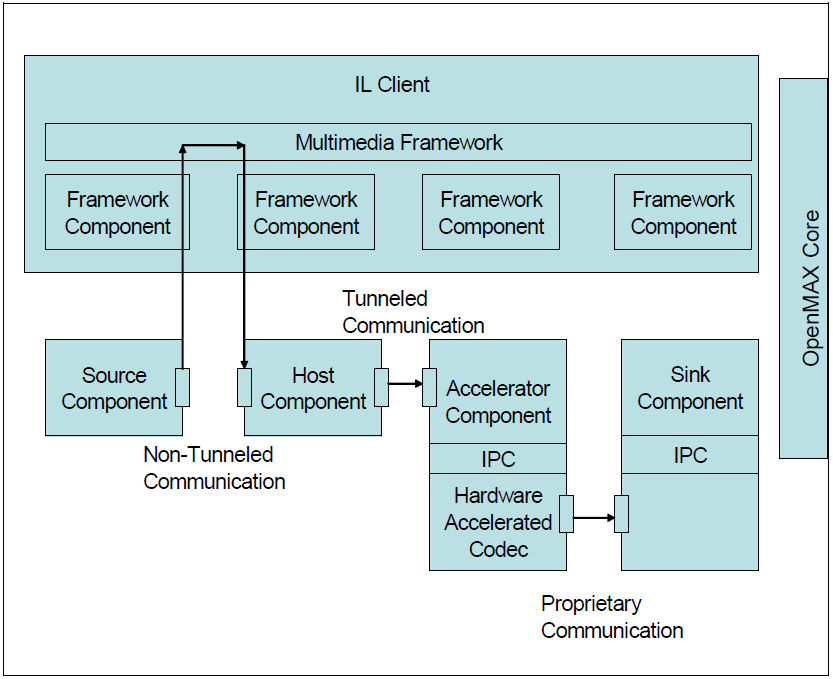


图2-2. OpenMAX IL API 系统组件

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。

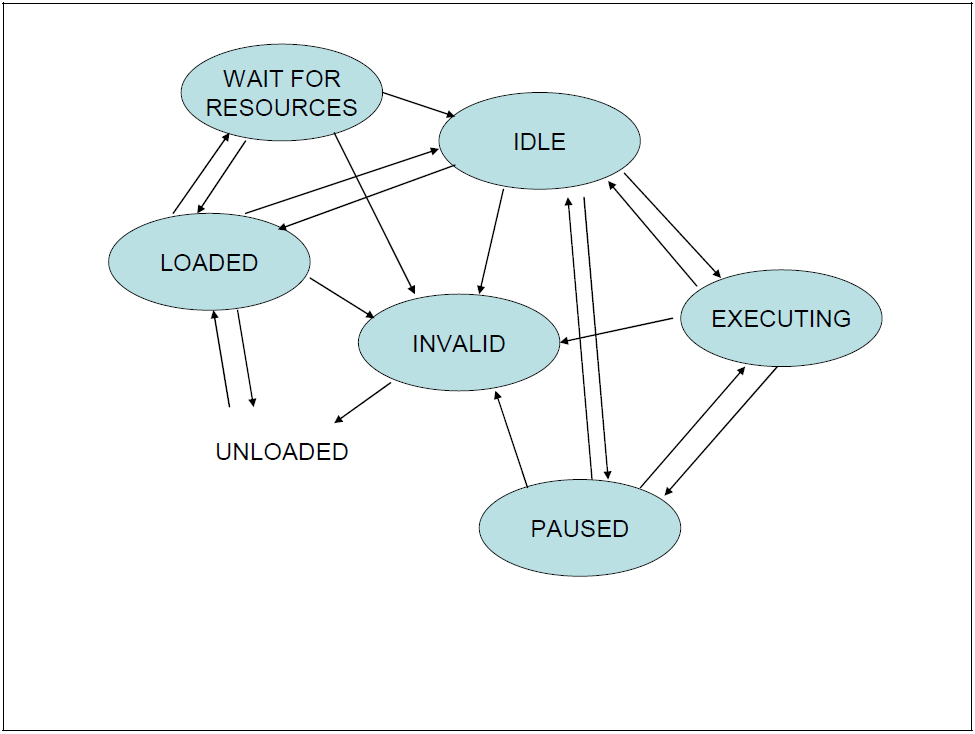


图 2-3. 组件状态

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。

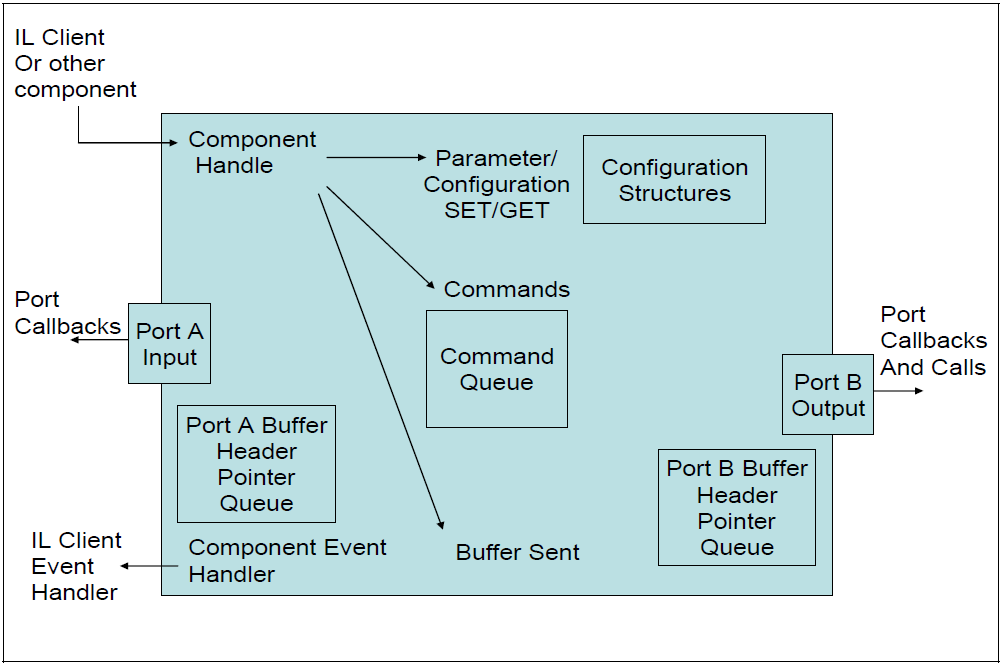


图 2-4. OpenMAX IL API 组件架构

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）

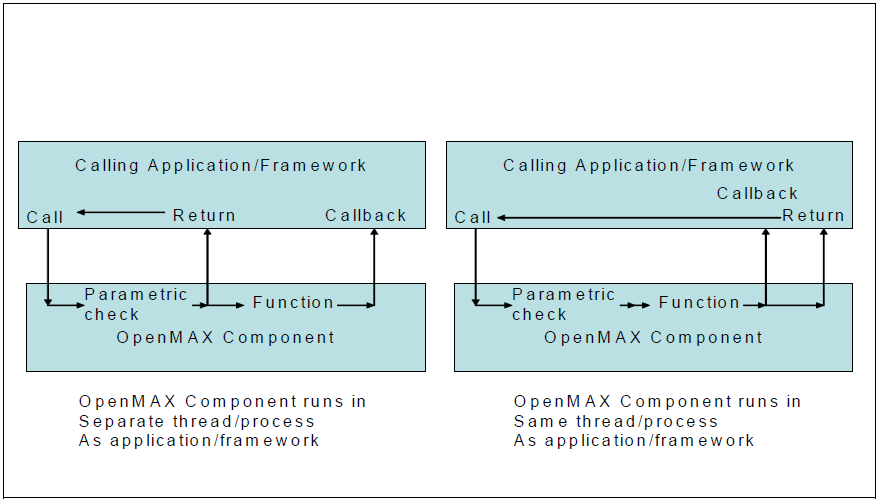


图 2-5. Out-of-Context VS In-Context 操作

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。

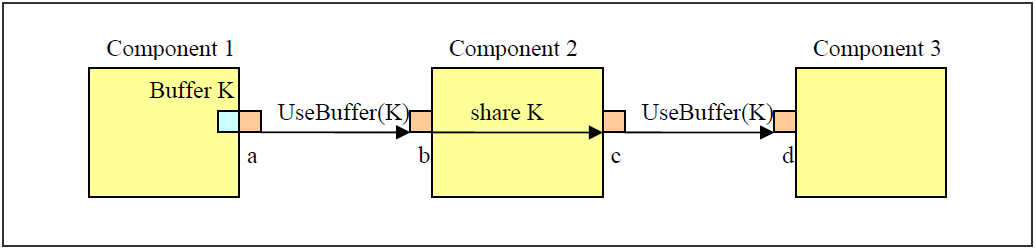


图2-6 缓冲区分配和共享关系示例

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”。

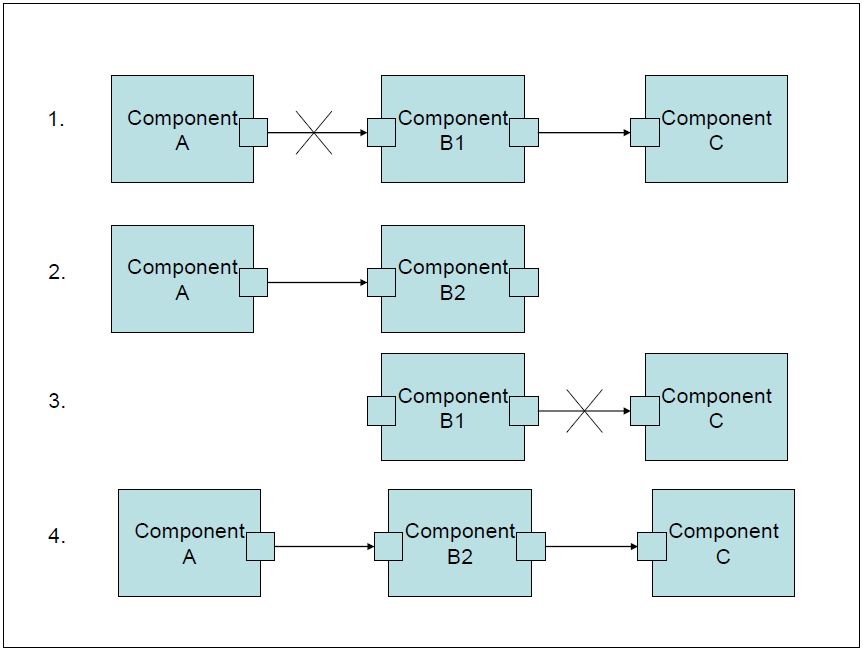


图 2-7. 端口重新连接

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。

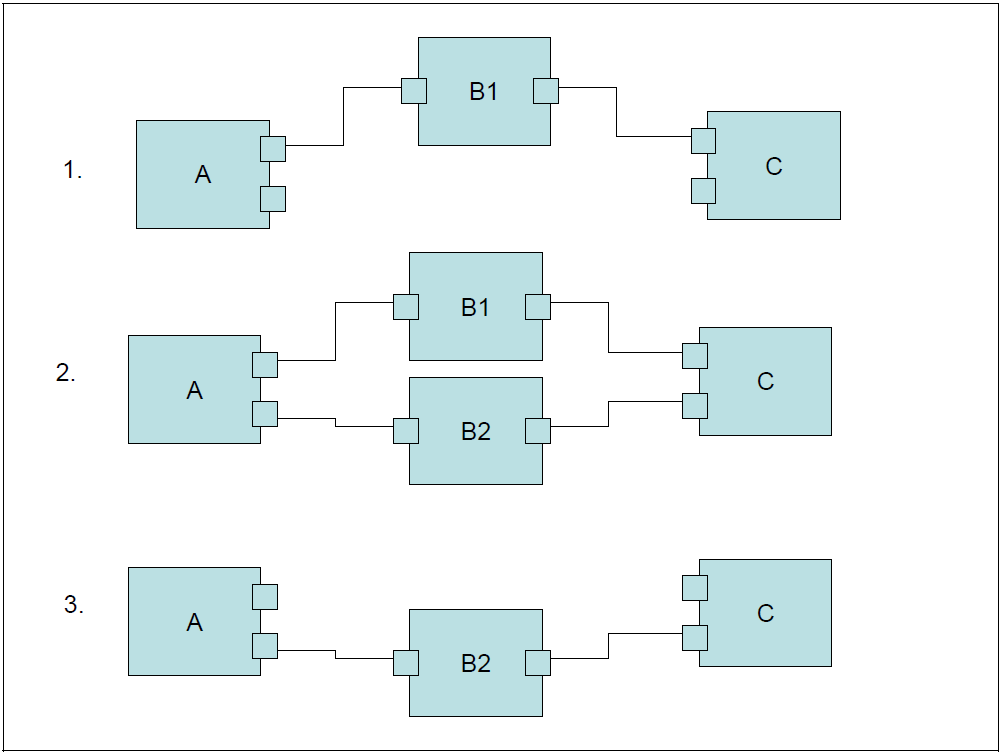


图 2-8. 重新连接组件

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。

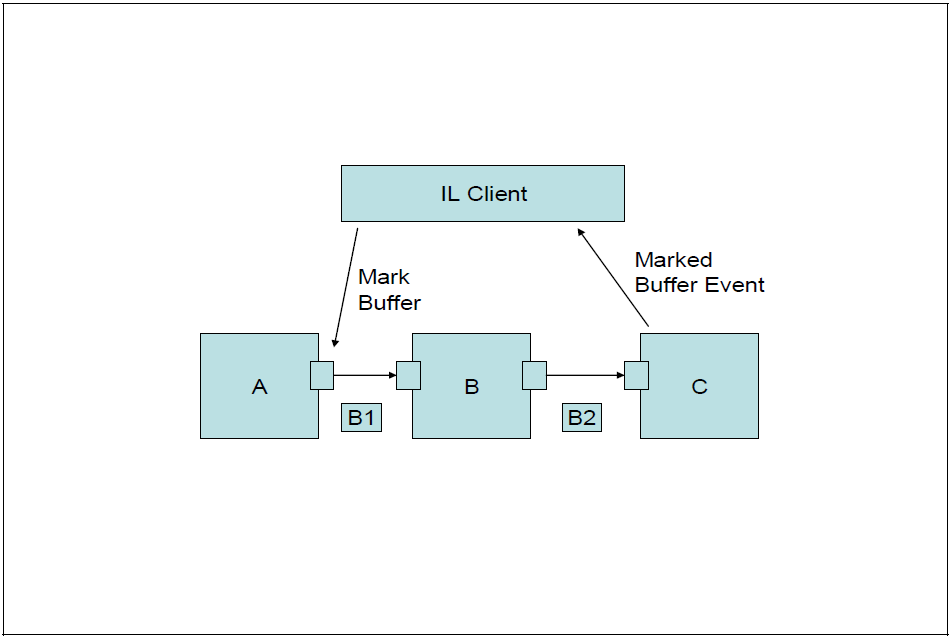


图 2-9. 标记缓冲区

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。

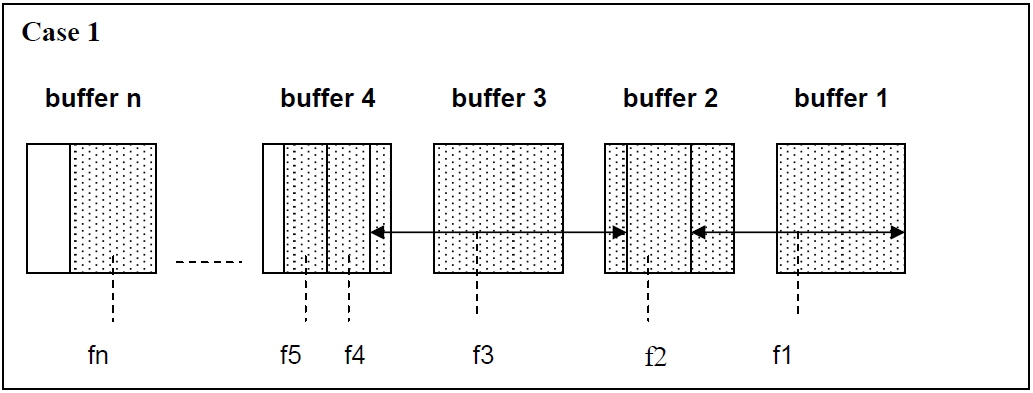


图2-10：案例1 - 每个缓冲区全部或部分填充

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。

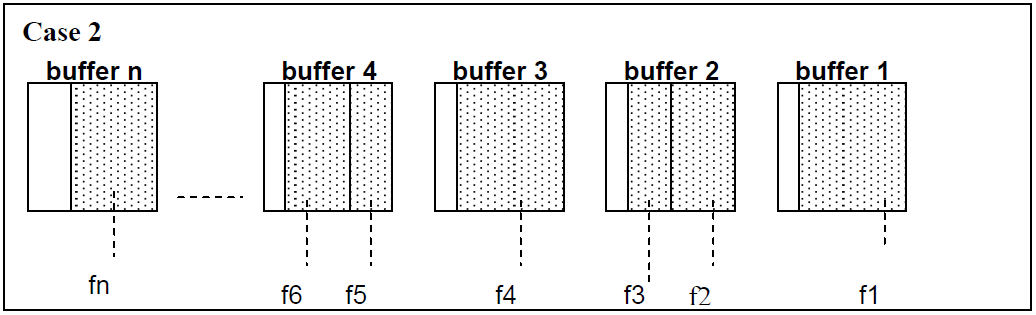


图2-11：案例2 - 每个缓冲区中只填充完整的数据帧

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。

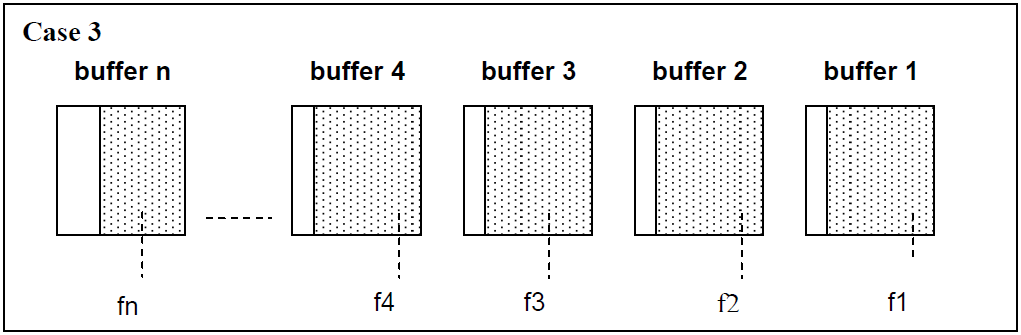
案例3：每个缓冲区只填充一帧压缩数据。

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。

第三章

第四章