Outline

1. What is OpenMAX-IL?
2. OMXIL Entities – Core, Components and Ports
3. OMXIL Component Architecture
4. OMXIL Profiles – Base and Interop
5. OMXIL Component States

Outline

6. Buffers and Buffer Headers
7. Simple Core/Component Usage
8. Tunneling Setup
9. *Transitions*
   - *Non-Tunneled*
   - *Tunneled*

Disclaimer:

- For simplicity, we are not considering buffer sharing and port enable/disable
- For simplicity, we do not consider intricacies on any scenarios where states of components are not done togethre.

Index

OpenMAX-IL

- From Khronos

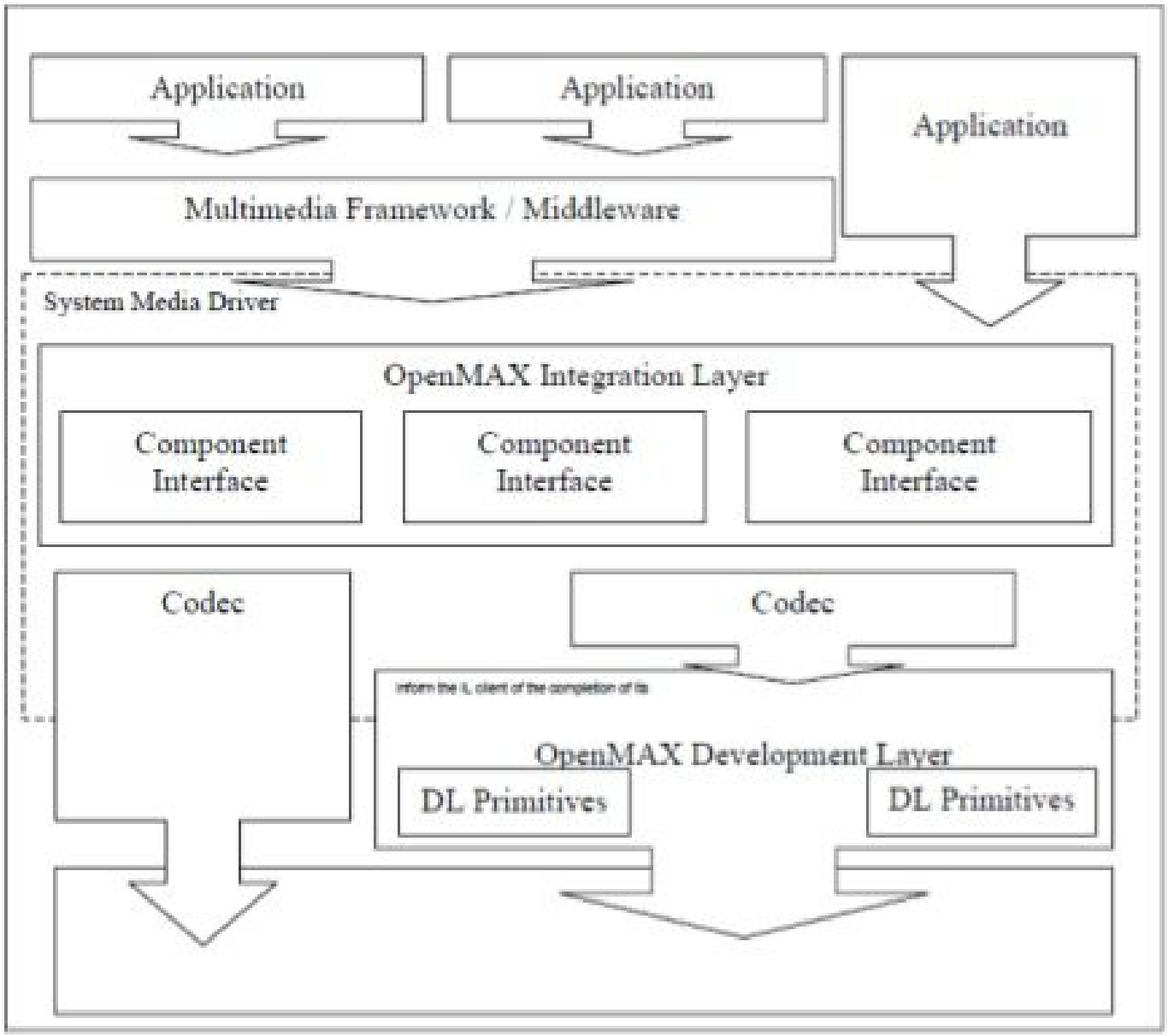
OpenMAX-IL

OpenMAX™ is a royalty-free, cross-platform API that provides comprehensive streaming media codec and application portability by enabling accelerated multimedia components to be developed, integrated and programmed across multiple operating systems and silicon platforms. The OpenMAX API will be shipped with processors to enable library and codec implementers to rapidly and effectively make use of the full acceleration potential of new silicon - regardless of the underlying hardware architecture.

OpenMAX-IL

- Is a cross-platform multimedia API
  - Not framework?
- C Based API
  - Very clean, concise
- Defines
  - data types [OMX_types.h]
  - APIs as functions and macros

Software Landscape



Need, Benefits

- Codecs, filters, sources, sinks etc have certain common characteristics, but have varied implementations.
  - State machine,
  - Command set, Parameters
  - Data exchange methodologies
  - API
- OpenMAX-IL standardizes such implementations with APIs (setup and communication), so that they are portable, benefiting – SV, ISV, OSV, OEMs

Index

OpenMAX-IL Entities

1. Core [OMX_Core.h]

2. Components [OMX_Component.h]
   a. Names, Roles
   b. Ports [OMX_Port.h]

3. {IL-Client}

Core

The OpenMAX-IL core is used *for and only for* -

- Dynamically loading and unloading components
- Facilitating component communication (tunneling)
- Queries about installed components and the roles they support
- Once loaded, the API allows the user to communicate directly with the component, which eliminates any overhead

Component

- An OpenMAX IL component is a software block which can produce / process / transform / consume media or related data
- Components can be sources, sinks, codecs, filters, splitters, mixers, or any other data operator
  - E.g., H264 Video decoder component
- Component could possibly represent a piece of hardware, a software codec, another processor, or a combination thereof
- OMXIL Component API and semantics are designed to be able to create (flexible) pipeline/graph of multiple components to realize varied combinations of media processing use cases.

Component

- Component has a name and may have 1 or more roles for discovery and selection based on capability
- Has input and output ports for media interconnection with other components and/or IL client
- Can be configured with parameters relevant to its operation
- Provides errors, asynchronous notifications, events to the IL client
- Has OMXIL defined state machine built-in for control by IL client

Component Name

- A component's unique identifier in a system.

- Naming convention -

`OMX.<vendor_name>.<vendor_specified_convention>`

  - For e.g., `OMX.STE.MP3Decoder.ProductXYZ`

- To load, IL client can enumerate and select a component by name.
  - Core API: `OMX_ComponentNameEnum`

Component Role

- Is the behavior of component acting according to a particular standard or vendor-specific component definition.
  - i.e., A role is a component's capability

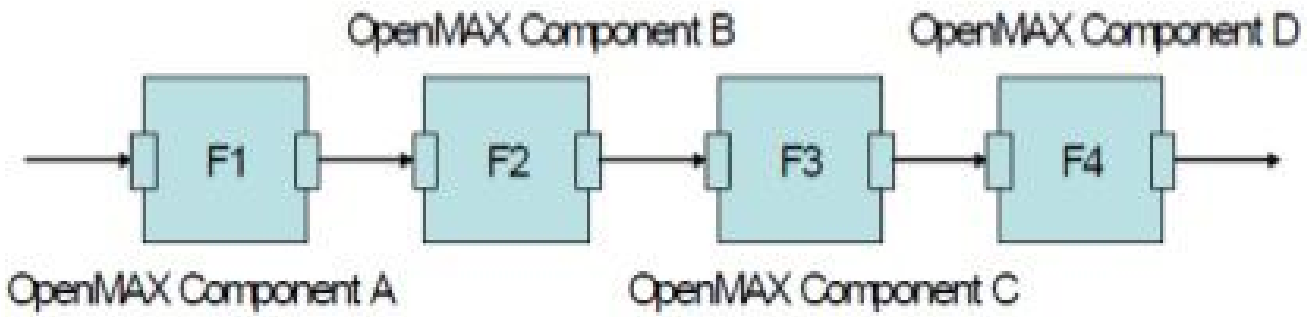
- A component named OMX.CompanyXYZ.MyAudioDecoder may support these roles
  - audio_decoder.mp3 (mp3 decode capability)
  - audio_decoder.aac (aac decode capability)
  - audio_decoder.amr (amr decode capability)

- IL client can choose a component to load by looking for a specific role using
  a. `OMX_ComponentOfRoleEnum`
  b. `Or, OMX_ComponentNameEnum + OMX_RoleOfComponent`

Ports

- Components have ports as a channel to communicate media data with other component or IL client.
- Types:
  - Input, Output
  - Image, Video, Audio and Other
  - E.g., Video input port

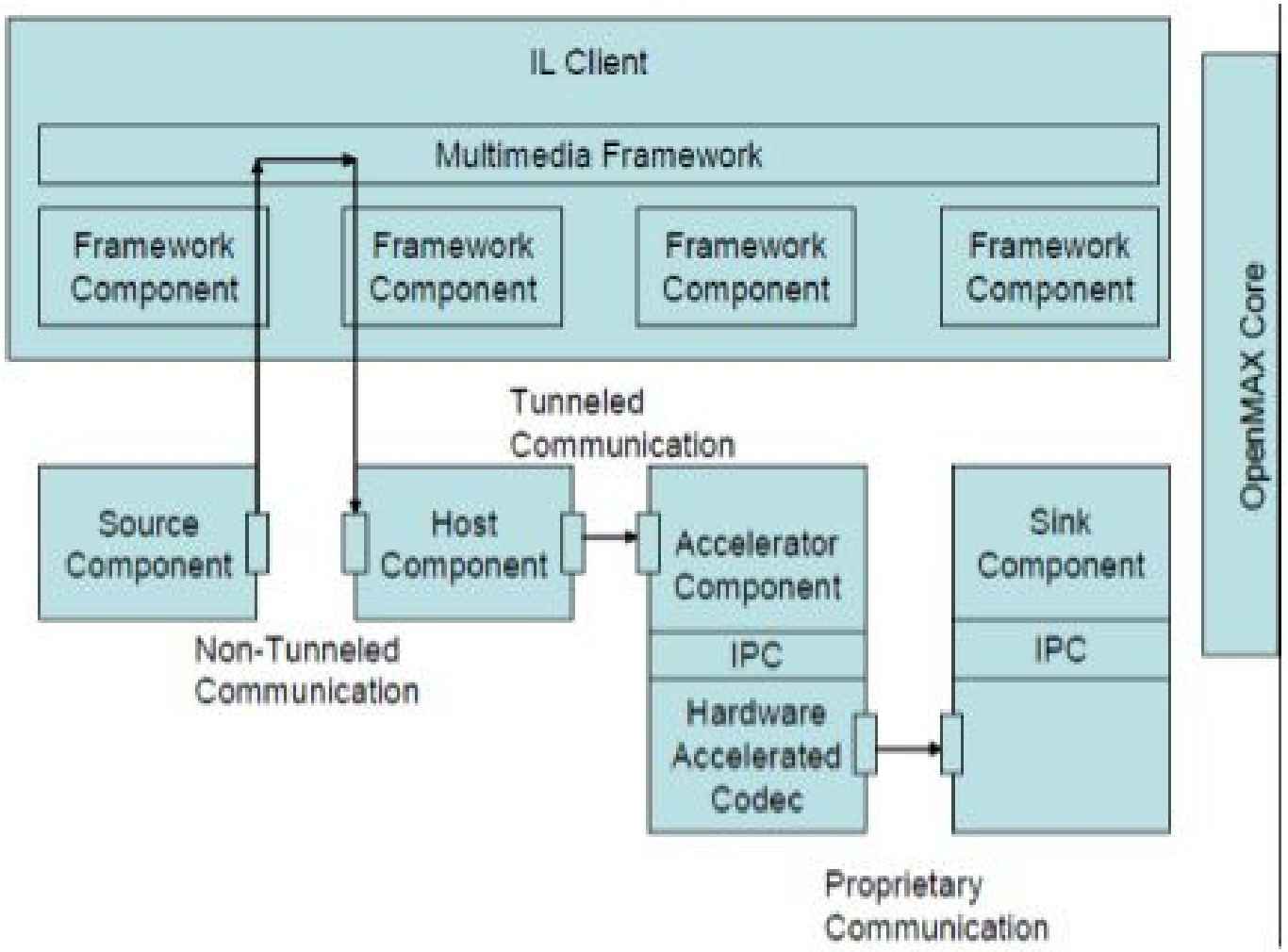
Ports


OpenMAX Component B
OpenMAX Component D
F1
F2
F3
F4
OpenMAX Component A
OpenMAX Component C

Port communication

- Non-tunneled
  - Data buffers are exchanged between client and component
- Tunneled
  - Data buffers are exchanged directly between components, in a way as defined by standard
- Proprietary
  - Data buffers are exchanged directly between components, via proprietary APIs

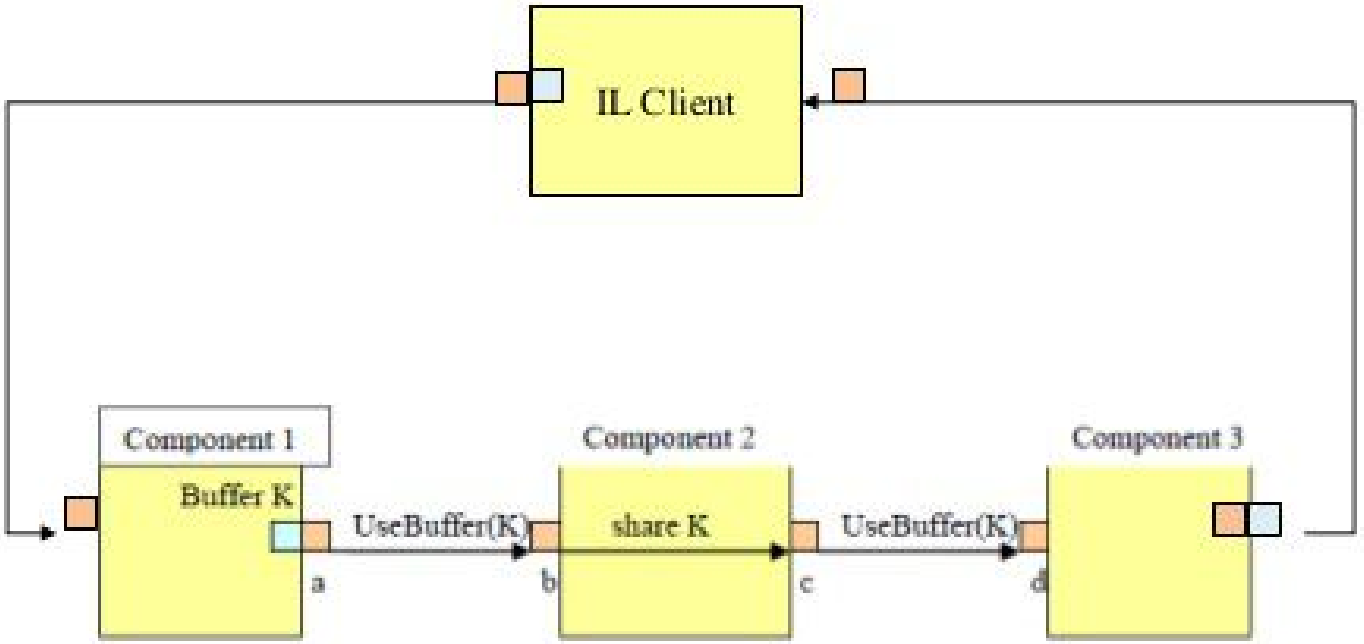
Port communication



Ports – some terminologies

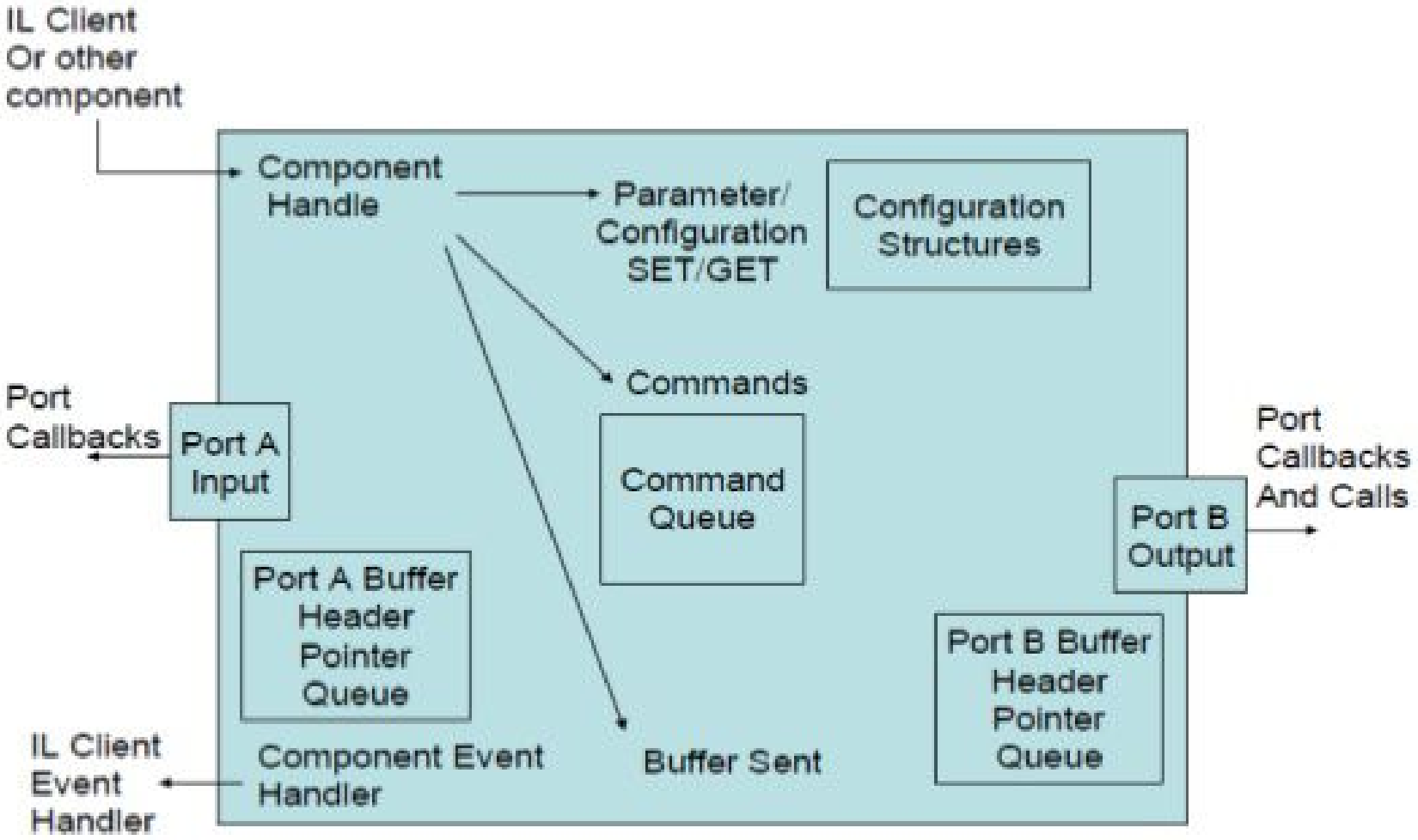
- A tunneling port is the port which it shares a tunnel with another component.
  - E.g., port b is the tunneling port to port a. Likewise, port a is the tunneling port to port b.
- An allocator port is a port that allocates its own buffers.
  - Port a is the only allocator port.
- Buffer supplier port is a port which supplies buffers to its tunneled port.
  - Buffers either allocated by itself (e.g., port a)
  - Or, buffers allocated by other components and shared (e.g., port c)
  - In non-tunneled comm, if IL client is supplying buffers, IL client is the buffer supplier.
- A sharing port is a port that re-uses buffers from another port on the same component.
  - E.g., port c
- A tunneling component is a component that uses at least one tunnel.

Ports – some terminologies



Index

OMXIL Component Architecture



Comp. Arch – Some new stuff

- Component handle
  - Opaque pointer to a component object in memory
  - Returned on creation of component by `OMX_GetHandle`
- Parameters/Configurations
  - Parameters are the static settings of a component. E.g., Number of buffers on a port etc
  - Configuration defines the dynamic settings of a component. For e.g., complexity of post processing etc
  - APIs: `OMX_GetConfig(index, valueStruct)`, `OMX_SetConfig`, `OMX_SetParameter`, `OMX_GetParameter`
  - It is mandatory to support some standard indices
- Commands
  - Allow IL client to control components
  - E.g., To move a component from one state to another `OMX_SendCommand(comp_handle, OMX_CommandStateSet…)`

Comp. Arch – Some new stuff

- Callbacks
  - Components are given a set of callback functions (at creation time) to provide asynchronous notifications/events to IL client.
  - Those are `EventHandler()`, `FillBufferDone()`, `EmptyBufferDone()`
- Event Handler
  - Component uses this to notify IL client of certain async events like completion of a command, Error, Port settings change etc.

Index

1. What is OpenMAX-IL?
2. OMXIL Entities – Core, Components and Ports
3. OMXIL Component Architecture
4. *OMXIL Component Profiles – Base and Interop*

Component profiles

- Base profile
  - *Shall* support non-tunneled communication
  - *Does not* support tunneled communication
  - *May* support proprietary communication
  - exists to reduce the adoption barrier for OpenMAX IL implementers by simplifying the implementation
- Interop profile
  - Is a superset of the base profile
  - *Shall* support non-tunneled and tunneled communication
  - *May* support proprietary communication

- Shall → Requirement. If not met, component is not conformant to OpenMAX IL
- Should → Not a requirement but recommended/good practice
- May → Optional requirement
- Will → Not a requirement

Types of Communication Supported Per Component Profile

Type of Communication	Base Profile Support	Interop Profile Support
Non-tunneled comm	Yes	Yes
Tunneled comm	No	Yes
Proprietary comm	Yes	Yes

NOTE:

An Interop profile compliant component may have both tunneled as well as non-tunneled communication

Index

1. What is OpenMAX-IL?
2. OMXIL Entities – Core, Components and Ports
3. OMXIL Component Architecture
4. OMXIL Component Profiles – Base and Interop
5. *OMXIL Component States*

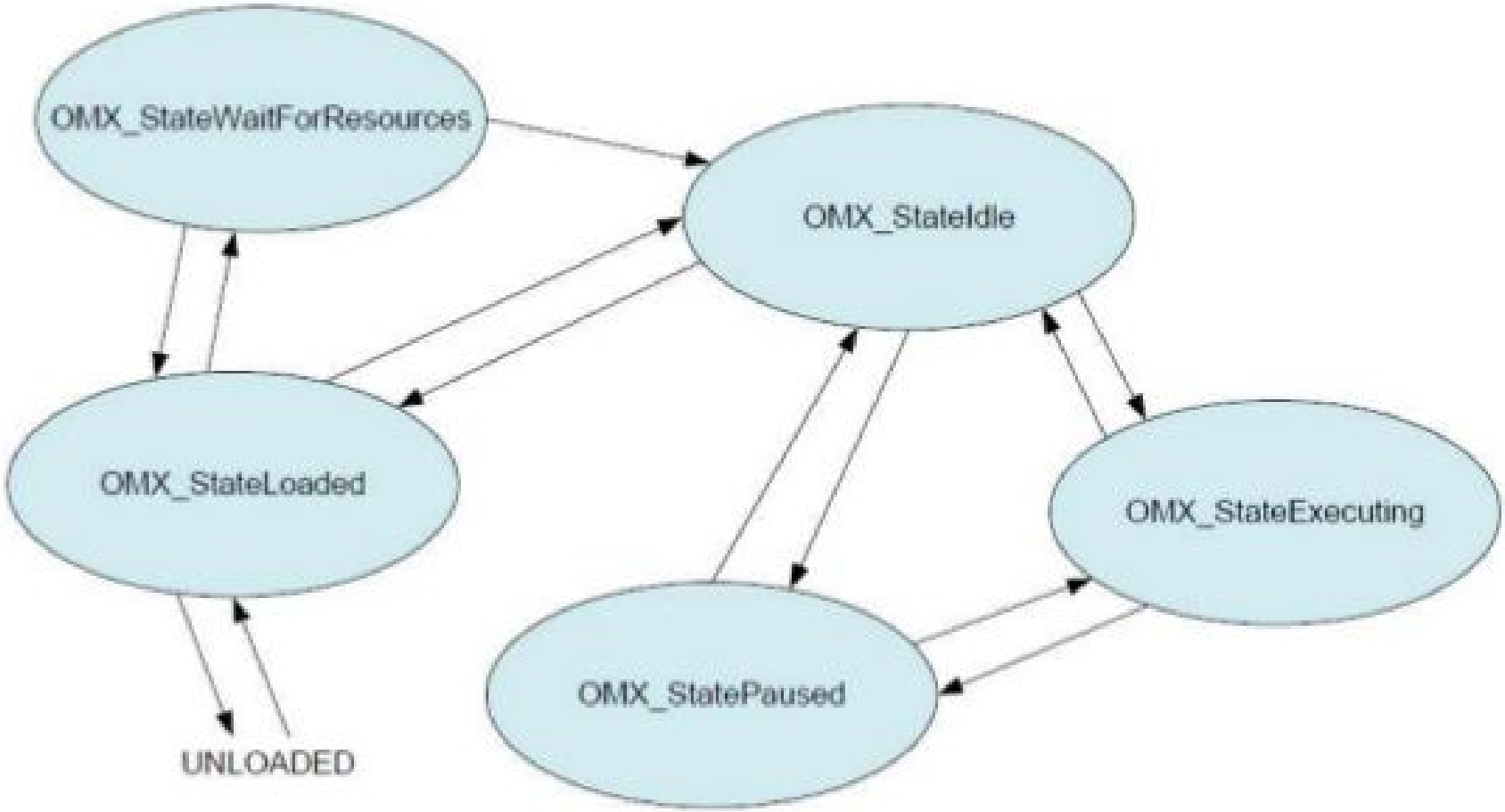
Component States

- OpenMAX-IL identifies the states that a component can be in at any point of time.
  1. Unloaded
  2. Loaded
  3. Idle
  4. Waiting for Resources
  5. Paused
  6. Executing

Component States

- IL client shall move components from one state to another.
  - A component can perform a state transition by itself only in cases of error, loss of resource etc
  - IL-client is notified of such change via `EventHandler`
- OMXIL defines a state transition graph
- OMXIL defines what the internal state (w.r.t buffers/communication/processing) of the component must be in each state
- OMXIL defines what a component must do while transitioning from one to another

State Transition Graph



Component States

- Every component is first considered to be unloaded.
- The component shall be loaded through a call to the OpenMAX IL core – `OMX_GetHandle()`.
- All other state transitions may then be achieved by communicating directly with the component.

Component States

- INVALID : unrecoverable 'dead' state. Can only be unloaded. Enters on event of fatal error.
- LOADED : component is in memory
- IDLE : Moved here from Idle. Has acquired static resources
- WAIT FOR RESOURCE – Intermediate between Loaded and Idle. Upon entering the WAIT FOR RESOURCE state, the component registers with a vendor-specific resource manager to alert it when resources have become available. The component will subsequently transition into the IDLE state.

Component States

- EXECUTING : state indicates data is processed and buffers are exchanged
- PAUSED : suspension of exchange and processing of buffers, with context preserved
- Transitioning from EXECUTING or PAUSED to IDLE will cause the context in which buffers were processed to be lost, which requires the start of a stream to be reintroduced.
- Transitioning from IDLE to LOADED will cause operational resources such as communication buffers to be lost.

Index

Buffers and Buffer headers

- Buffers are raw pieces of memory that contain media or related data

- Buffer headers are structures that along with the buffer pointer, contain metadata describing the data in buffers
  - Buffer allocated size, data size, data offset etc

- Between two entities at either ends of a communication path, one allocates the buffers, while the other allocates buffer header.
  - Between two tunneled ports or
  - Between IL client and the port it interfaces with

Buffers and Buffer headers

- Data communications with components is always directed to a specific component port.

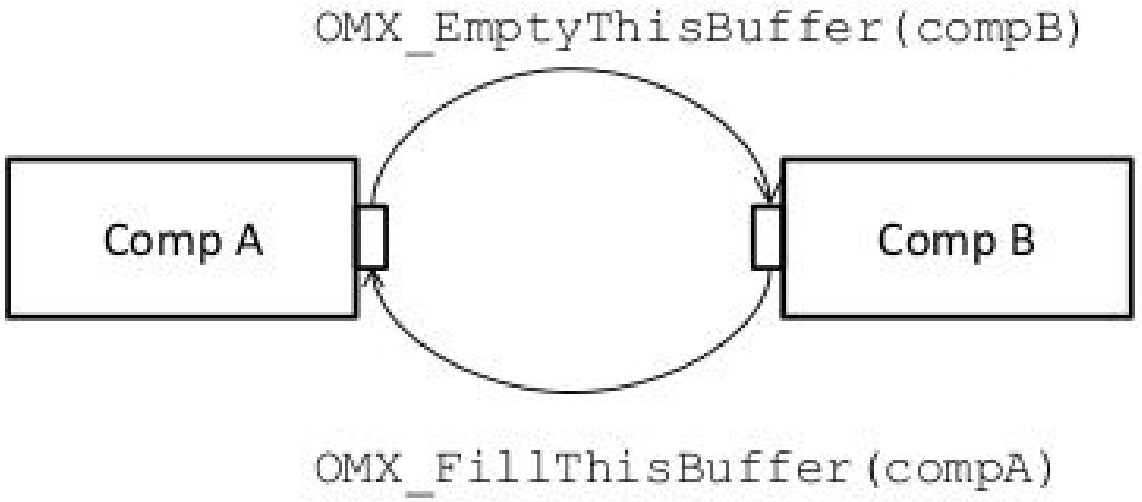
- Every component port ***shall*** be capable of allocating its own buffers or using pre-allocated buffers; one of these choices will usually be more efficient than the other.

Buffers and Buffer headers

Buffer communication APIs:

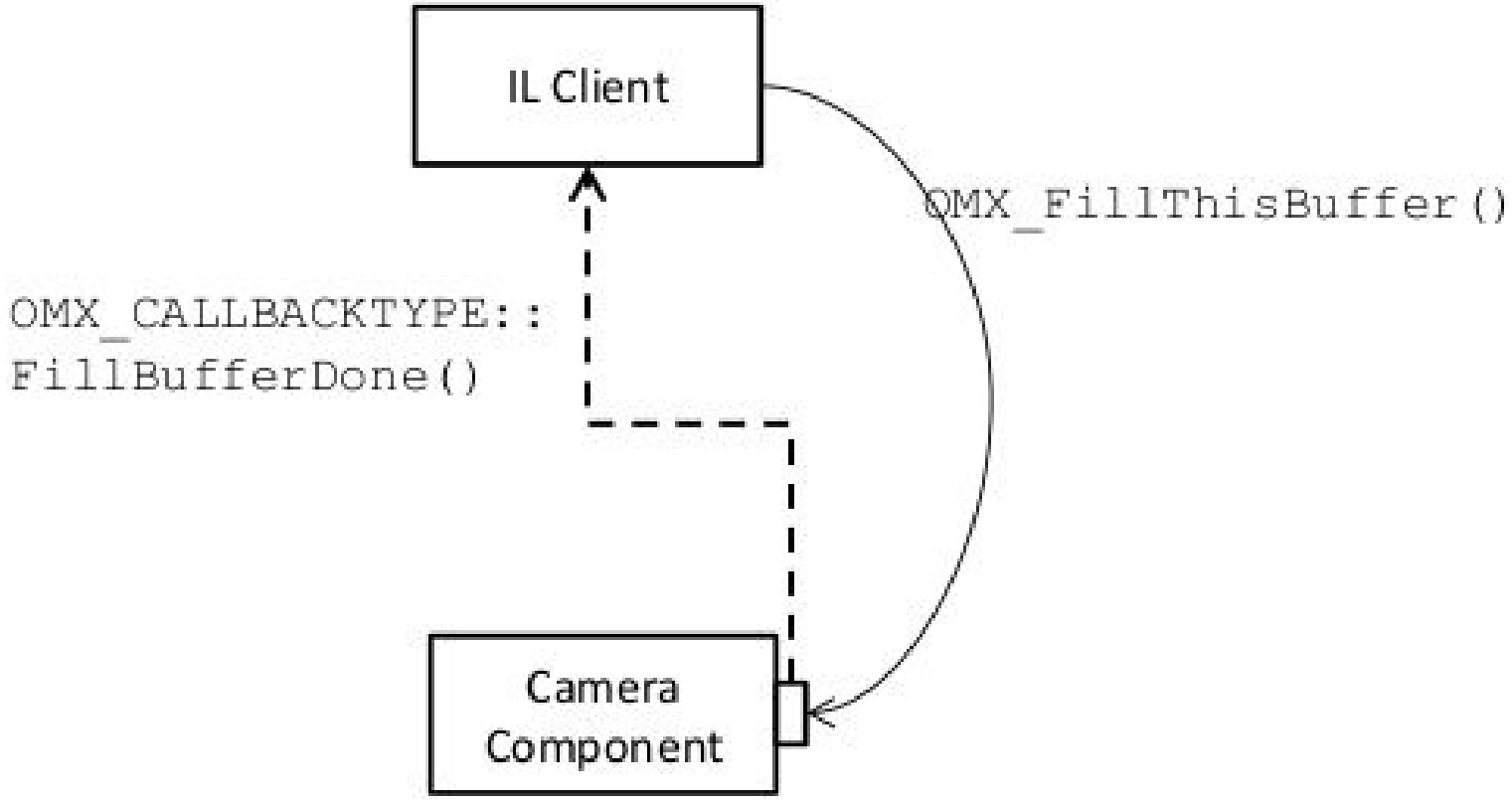
- `OMX_EmptyThisBuffer` is always called on input ports
  - Called by a tunneled component or IL client
- `OMX_FillThisBuffer` is always called with output ports
  - Called by a tunneled component or IL client
- `EmptyBufferDone` or `FillBufferDone` callbacks
  - Callback functions called by component (port) on IL client to notify completion of emptying or filling.
  - Used only in non-tunneled communication

Buffers and Buffer headers



Buffer exchange between tunneled ports

Buffers and Buffer headers



Buffer exchange in non-tunneled mode

Index

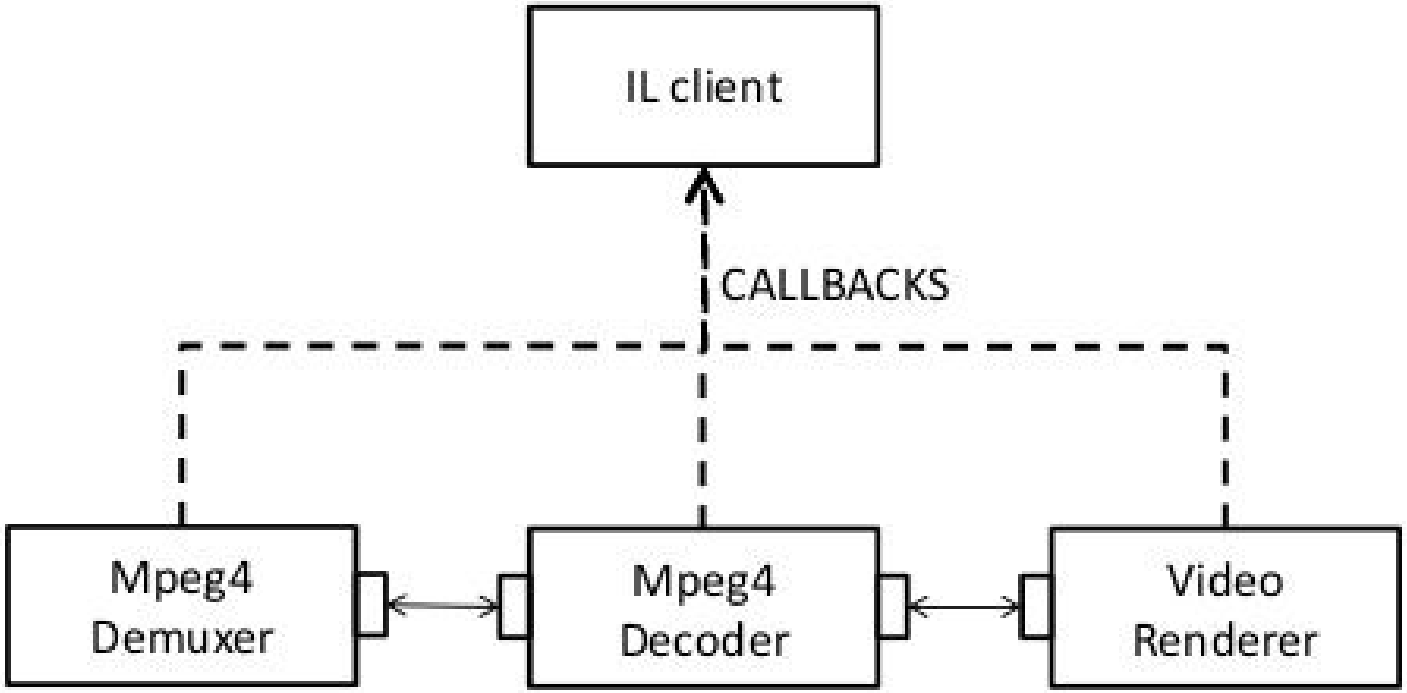
Simple Core/Component Usage

- OMXIL core is initialized with `OMX_Init()`
- Multiple components are chosen after enumeration and created with `OMX_GetHandle()`
  - Components are now in `OMX_Loaded` state.
- IL client sets relevant parameters and configurations of components in this state
- IL client sets up tunnels between two compatible ports where it deems possible using `OMX_SetupTunnel()`.
- IL client moves all components from Loaded → Idle state using `OMX_SendCommand(comp_handle, OMX_CommandStateSet…)`
  - A notification/error is received by IL client via `EventHandler` about state change

Simple Core/Component Usage

- IL client moves all components from Idle → Executing state
  - Buffer exchanges begin transferring media data 'downstream'
- For e.g., if the components are Mpeg4 demuxer, Mpeg4 decoder and Video renderer, in `OMX_Executing` state, video frames are rendered on to the screen
- IL client is notified of 'End of Stream' when playback is done.
- IL client moves all components to `OMX_Idle` state
- IL client then, moves all components to `OMX_Loaded` state
- IL client then, calls `OMX_Deinit()` upon which the components are deleted, and core is de-initialized.

Simple Core/Component Usage



Buffer exchange between tunneled ports

Index

Tunneling setup

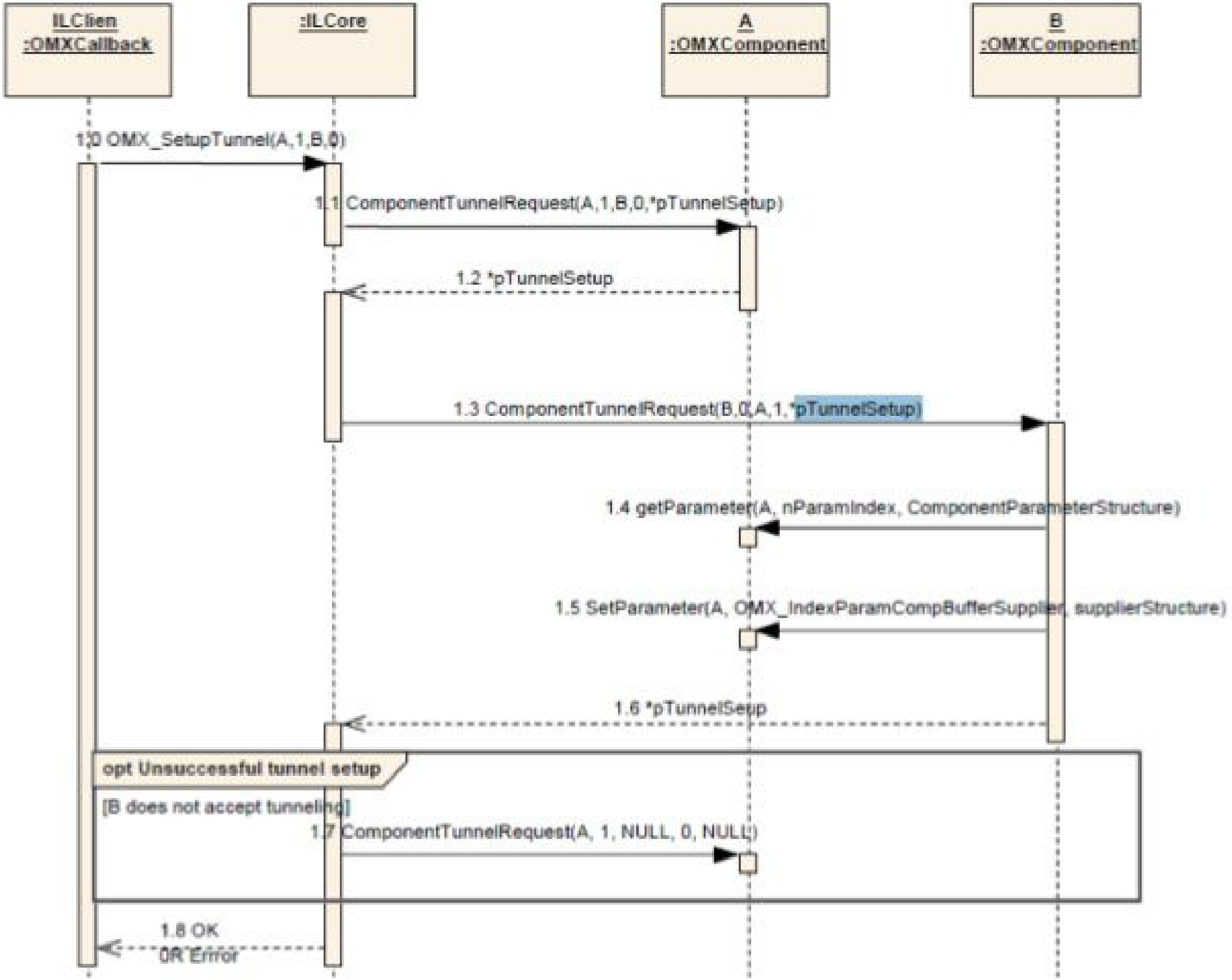
- IL client shall establish tunneling between two ports when two components are in `OMX_Loaded` state.
- Using `OMX_SetupTunnel(compA, outputPortNum, compB, inputPortNum)`
- For a given tunnel, exactly one port supplies the buffers. The component (or port) which supplies buffers is decided during tunnel setup.

Tunneling setup - Protocol

- The `OMX_SetupTunnel` impl first calls `CompA::ComponentTunnelRequest(setupPrefA)` – the component of output port. CompA fills `setupPrefA` and returns.

- `OMX_SetupTunnel` then calls `CompB::ComponentTunnelRequest(setupPrefA)`. – the component of input port .
  - Now, setupPrefA is read-only for compB's review.

- Comp B reviews `setupPrefA`, checks compatibility
  - If OK, decides which port would be the buffer supplier and sets the same on CompA with `OMX_SetParameter(OMX_IndexParamCompBufferSupplier, ..)`


ILClien
:OMXCallback
:ILCore
A
:OMXComponent
B
:OMXComponent
1.0 OMX_SetupTunnel(A,1,B,0)
1.1 ComponentTunnelRequest(A,1,B,0,*pTunnelSetup)
1.2 *pTunnelSetup
1.3 ComponentTunnelRequest(B,0,A,1,*pTunnelSetup)
1.4 getParameter(A, nParamIndex, ComponentParameterStructure)
1.5 SetParameter(A, OMX_IndexParamCompBufferSupplier, supplierStructure)
1.6 *pTunnelSetup
opt Unsuccessful tunnel setup
[B does not accept tunneling]
1.7 ComponentTunnelRequest(A, 1, NULL, 0, NULL)
1.8 OK
OR Error

… Buffer supplier Negotiation

- For tunneling, CompA may prefer buffer supplier to be
  Output port OR Input port OR Unspecified
  (this is part of `setupPrefA`)

- In any case, its up to the impl of compB to honor or disregard CompA's preference.
  - CompB may call CompA->SetParameter(ParamBufferSupplier, BsChoice)
  - CompB's choice may vary from one comp to the other.

- (The negotiation is not clearly specified and depends on the `OMX_SetupTunnel` implementation)
- After successful setup, IL client can override the buffer supplier

Tunneling – Buffer Allocation

- To set up tunneling components, the IL client should perform the following setup operations in this order:
  - Load all tunneling components and set up the tunnels on these components.
  - Command all tunneling components to transition from the loaded state to the idle state.

    Each supplier port of a non-sharing component does the following
    - Determine the buffer requirements of its tunneled port via an OMX_GetParameter call.
    - Allocate buffers according to the maximum of its own requirements and the requirements of the tunneled port.
    - Call OMX_UseBuffer on its tunneling port.

Index

Transitions (T|NT)

- Loaded
  1. [Tunnel setup]
- Loaded to Idle:
  1. Buffer and Buffer header allocation
- Idle to Executing
  1. Start buffer exchanges
- Executing to Idle
  1. Return buffers to their owners / IL client
- Idle to loaded
  1. Owners delete buffers
  2. Owners ask tunneled ports to free related data.

Index

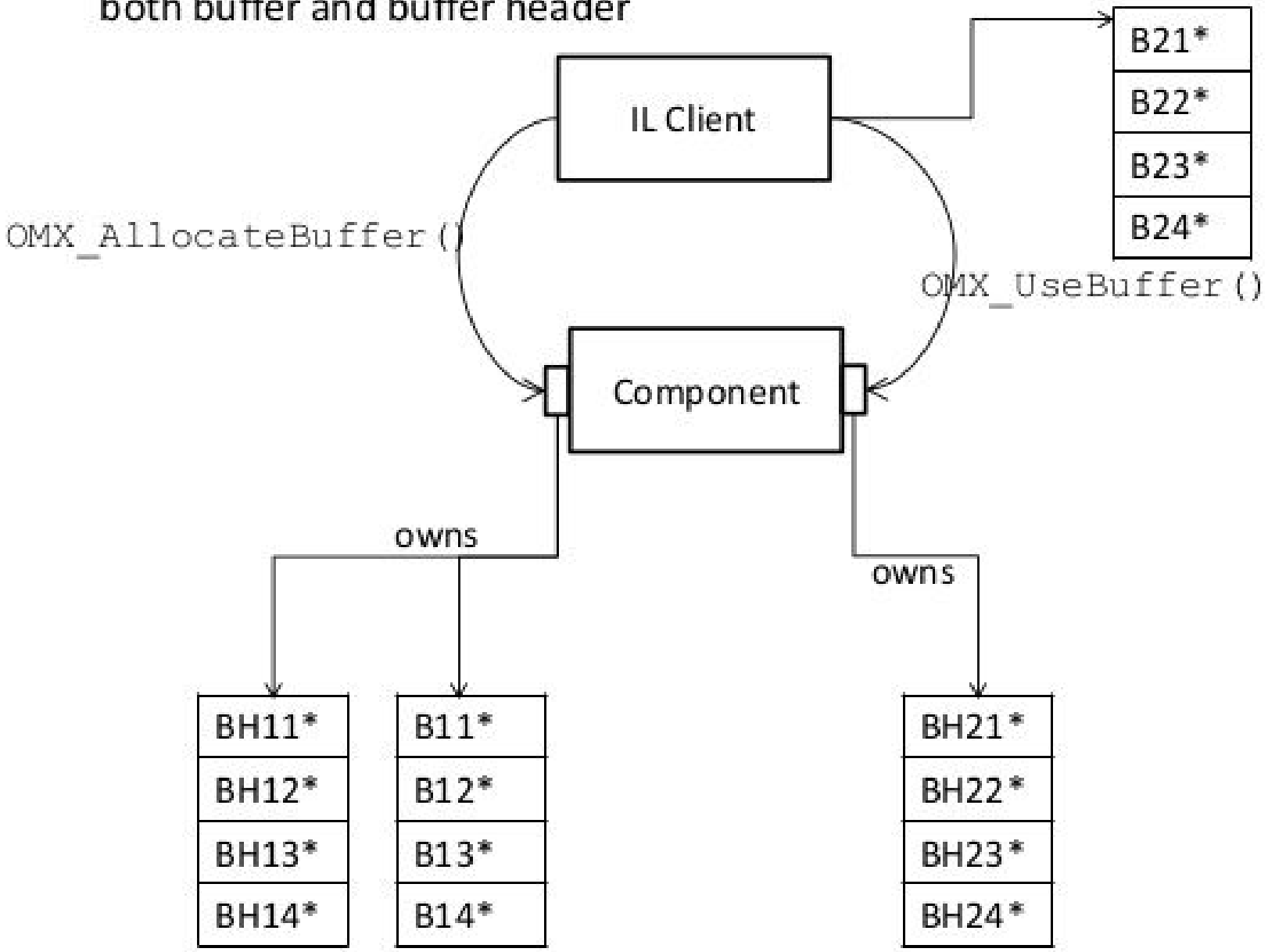
6. Buffers and Buffer Headers
7. Simple Core/Component Usage
8. Tunneling Setup
9. *Transitions*
   - *Non-Tunneled*

Non-tunneled - Loaded to Idle

- IL client asks component to move from loaded to idle
- Buffer supplier in non-tunneled mode is always the IL client
- IL client calls `OMX_AllocateBuffer` / `OMX_UseBuffer` on each port it interfaces
- Component thus has needed buffers on all ports and moves to Idle state.
- [Who allocates buffers is decided by IL client]

Non-tunneled - Loaded to Idle

- `OMX_AllocateBuffer` is only called in a non-tunneled mode. Allocates both buffer and buffer header

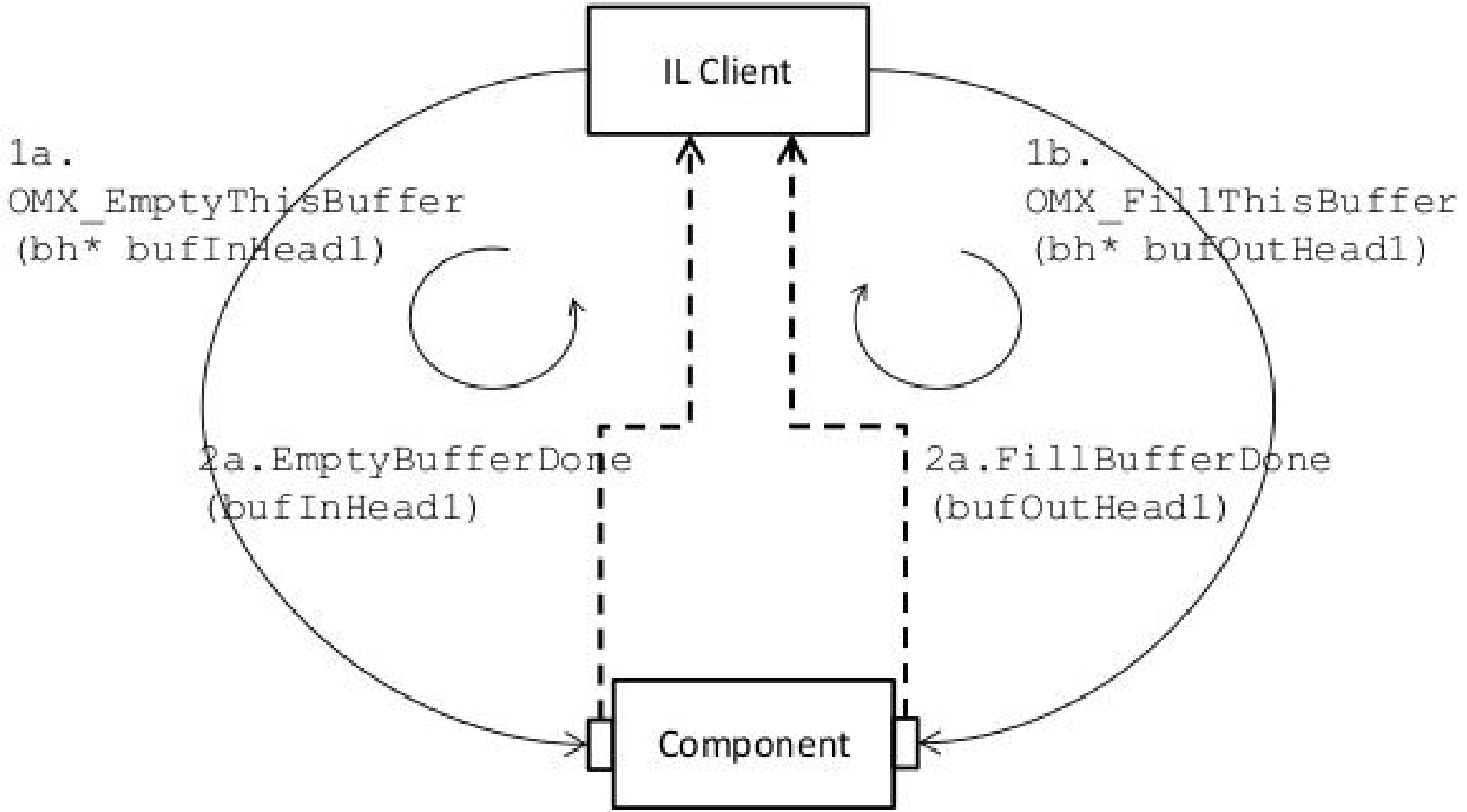


Non-tunneled – Idle to Executing

- Idle to Executing:
  - IL client asks the component to move from Idle to Executing state
  - Component transitions to Executing
- In Executing state…
  - IL client is entirely responsible for moving data buffers among components if data tunneling is not used
  - IL client calls OMX_EmptyThisBuffer on input port, OMX_FillThisBuffer on output port of the component.
  - Components provide feedback via FillBufferDone / EmptyBufferDone
  - IL client then calls OMX_EmptyThisBuffer/ OMX_FillThisBuffer to continue buffer exchange

Non-tunneled – Executing

- `OMX_AllocateBuffer` is only called in a non-tunneled mode. Allocates both buffer and buffer header



Non-tunneled – Executing to Idle

- IL client asks the component to move from Executing to Idle state
- Component returns buffers to IL client (buffer supplier) using FillBufferDone/EmptyBufferDone and moves to Idle state

Non-tunneled - Idle to Loaded

- IL client asks the component to move from Executing to Idle state
- Buffer supplier (IL client) calls `OMX_FreeBuffer` for each buffer on the non-supplier port
- Non-supplier port deletes buffer header. It also deletes buffer if the component had allocated it (via `OMX_AllocateBuffer`)
- Buffer supplier (IL client) deletes buffer before calling `OMX_FreeBuffer` if the buffer was allocated by the IL client.
- Component moves to Loaded state.

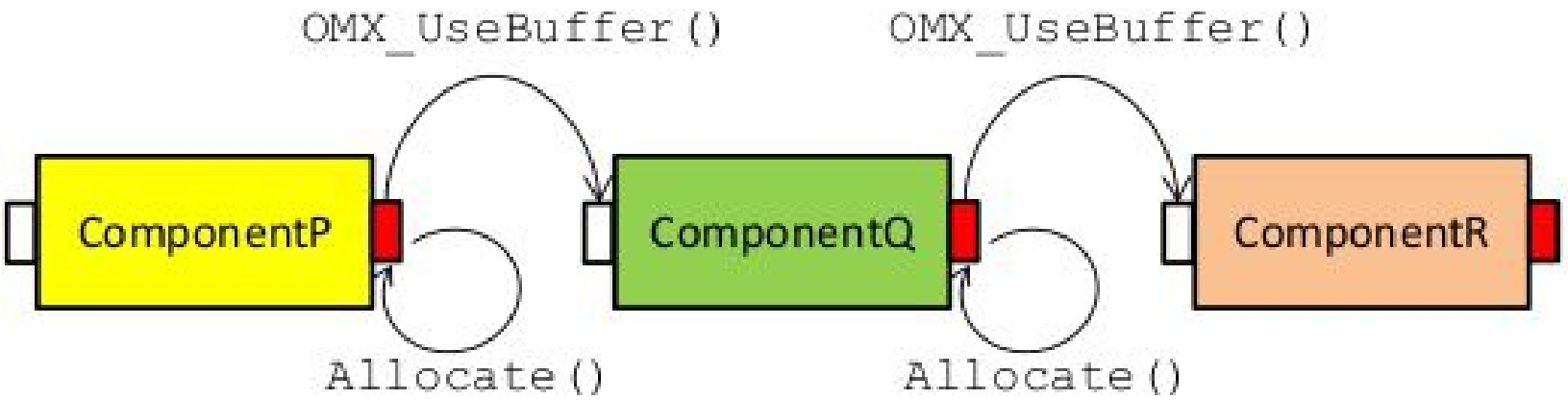
Index

Tunneled - Loaded to Idle

- IL client sets up tunnel between components → buffer supplier responsibilities are agreed.
- IL client asks each component to move from loaded to idle
- Component then allocates buffers for each buffer supplier port. Each such buffer is then forwarded to its tunneled port (say B) using `OMX_UseBuffer` call. B port then allocates buffer header for each buffer
- Each component allocated static resources – buffers and buffer headers. Then the component moves to Idle state.

Tunneled - Loaded to Idle

-*Allocate() is an internal function
- Red ports are allocator ports



BHp1*	BHq1*
BHp2*	BHq2*
BHp3*	BHq3*
BHp4*	BHq4*

	BHr1*
	BHr2*
	BHr3*
	BHr4*

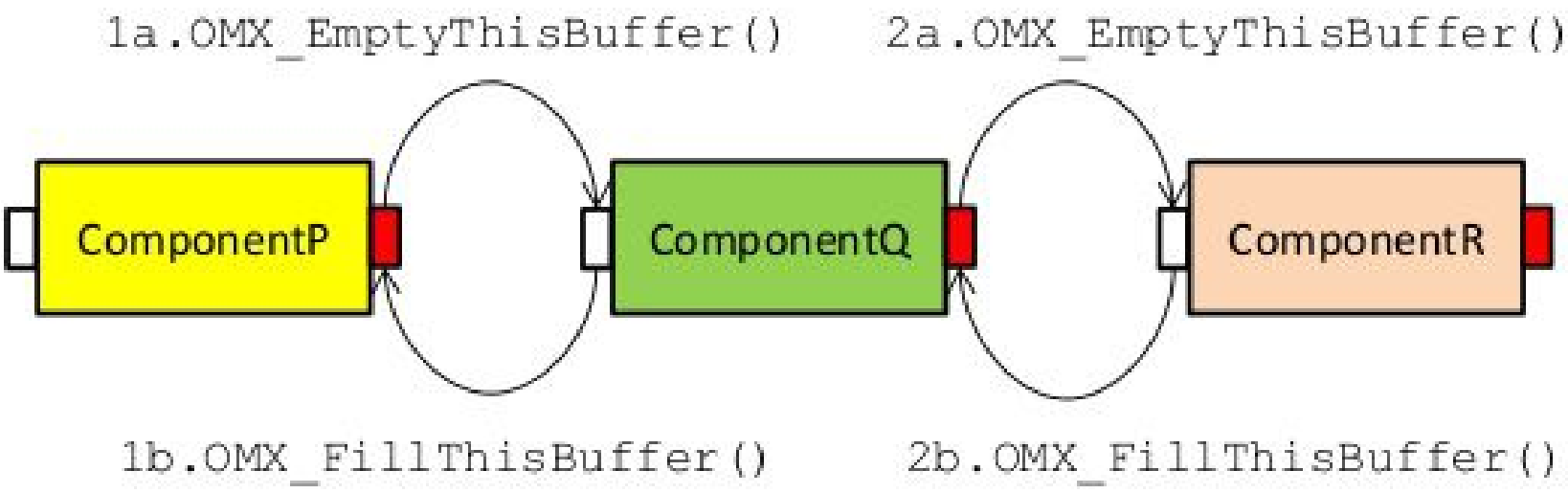
	BHs1*
	BHs2*
	BHs3*
	BHs4*

Input and output queues of components, showing buffer header possession

Tunneled – Idle to Executing

- Idle to Executing:
  - IL client asks the component to move from Idle to Executing state
  - Component moves to Executing state and initiates buffer exchanges
    - an input buffer supplier port calls `OMX_FillThisBuffer`
- In Executing state
  - an output buffer supplier port calls `OMX_EmptyThisBuffer`, after filling it with data
  - The buffers are then exchanged using Fill/Empty APIs.

Tunneled - Executing



BHp1*	
BHp2*	
BHp3*	
BHp4*	

BHq1*	
BHq2*	
BHq3*	
BHq4*	

BHr1*	BHs1*
BHr2*	BHs2*
BHr3*	BHs3*
BHr4*	BHs4*

Input and output queues of components, showing buffer header possession

Tunneled - Executing to Idle

- IL client asks each component to move from Executing to Idle state
- A component on such request, returns buffers to buffer supplier using `OMX_EmptyThisBuffer/OMX_FillThisBuffer`
- Thus all buffers get home.
- On getting all its owned buffers, a component moves from Executing to Idle state.

Tunneled - Idle to Loaded

- IL client asks each component to move from Loaded to Idle state
- A component then, deletes the buffer and then calls `OMX_FreeBuffer` on its tunneled non-supplier port.
- The non-supplier tunneled port deletes the buffer header.
- On deleting all buffers, each component moves to Loaded state