Using OpenMAX Integration Layer with GStreamer

- WHITE PAPER -

Author:	D. Melpignano, P. Sen
Version:	1.0
Date:	24st April, 2006

REVISION HISTORY

VERSION	DATE	AUTHORS	COMMENTS
0.9	21st April 2006	D. Melpignano, P. Sen	Final draft
1.0	24th April 2006		Released

GStreamer is a plugin-based framework licensed under the LGPL:
http://gstreamer.freedesktop.org

OpenMAX is a registered trademark of the Khronos Group. All references to OpenMAX components in this whitepaper are referenced from the publicly available OpenMAX IL specification on the Khronos web-site at:

http://khronos.org/openmax

Information furnished is believed to be accurate and reliable. However, STMicroelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of STMicroelectronics. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. STMicroelectronics products are not authorized for use as critical components in life support devices or systems without express written approval of STMicroelectronics.

The ST logo is a registered trademark of STMicroelectronics.
Nomadik is a trademark of STMicroelectronics
All other names are the property of their respective owners

© 2004 STMicroelectronics - All rights reserved

STMicroelectronics group of companies
Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

Table of Contents

1 Scope **4**
2 Introduction **4**
3 Gstreamer concepts **5**
3.1 GST terminology 5
3.2 GStreamer Elements 5
3.3 GStreamer Object Hierarchy 6
4 Gstreamer – IL integration **7**
4.1 GST element and OMX component states 8
4.2 Comparison of methods 9
4.3 Comparison of data structures 10
4.4 Initialization 14
4.5 Data flow 17
4.6 Example of IL-enabled GST plugin set 19
4.7 Optimizations 20
5 Conclusions **21**
6 References **21**

1 SCOPE

The purpose of this document is showing how the OpenMAX Integration Layer API can be used in the Linux GStreamer framework to enable access to multimedia components, including HW acceleration on platforms that provide it.
The white paper does not enter into implementation details and is intended to just provide indications of how the OpenMAX Integration Layer API might be used in the GStreamer multimedia framework.
Intended audience is the Linux development community, especially the development community around GStreamer.
It is assumed that the reader is familiar with the OpenMAX Integration Layer API (defined in [1]).

2 INTRODUCTION

GStreamer is a fully featured Multimedia framework for the Linux operating system.
The purpose of this white paper is to discuss how access to multimedia components - as offered by the OpenMAX Integration Layer (OMX IL) API - can be exploited by GStreamer.

Adding support for OMX IL inside GStreamer has the advantage of enabling access to multimedia components in a standardized way. Applications using the GStreamer API would take advantage of hardware acceleration on platforms that provide it, when OMX IL support is integrated.

Although the OpenMAX Integration Layer API may initially look similar to GStreamer in terms of concepts, it is aimed at a different purpose and lacks many advanced features that can be found in GStreamer.

This white paper briefly discusses the main concepts of the GStreamer framework, then it analyzes the main differences with OMX IL in terms of data structures and methods.
A simple approach for using the OpenMAX IL API with GStreamer is suggested, which does not require modifications to the GStreamer core framework.
Sequence diagrams show the interaction among GStreamer and OpenMAX IL function calls during the initialization and pipeline execution phases.

In the rest of the document, GStreamer version 0.8 is used as a reference. GStreamer 0.10 slightly differs in the API, but not in the fundamental concepts that are herein described. Where appropriate, footnotes are used to highlight the differences among version 0.8 and 0.10.

3 GSTREAMER CONCEPTS

In this section we briefly summarize the main concepts used in the GStreamer multimedia framework (see [2]).

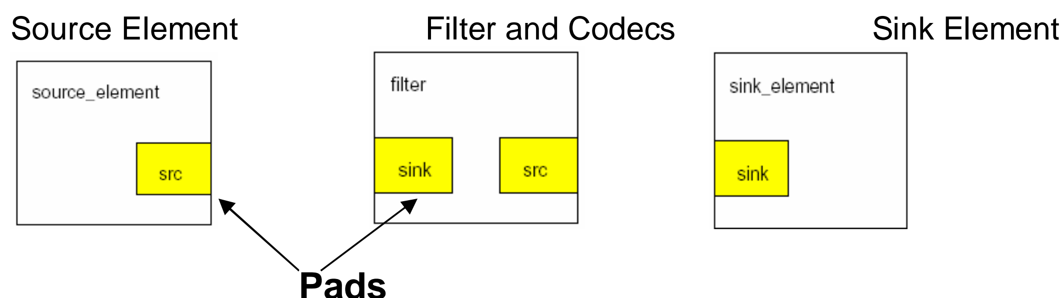
3.1 GST terminology

The following definitions are taken from the GStreamer official documentation.

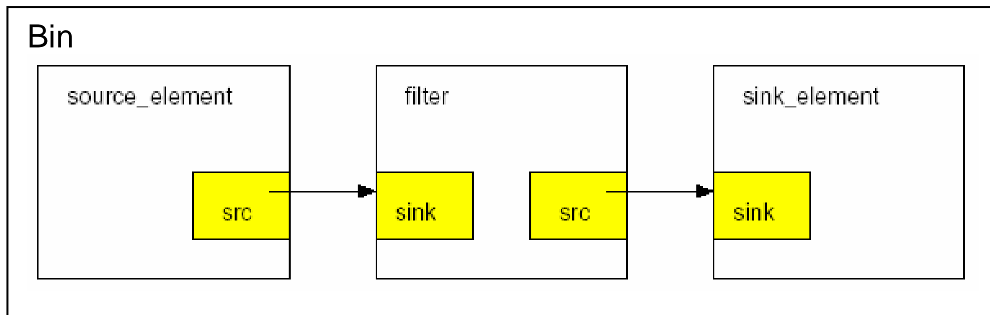
- A GStreamer plugin is a container for features (elements) loaded from a shared object module.
- A GstRegistry is an Abstract base class for management of GstPlugin objects
- A GstObject is a Base class for the GStreamer object hierarchy
- A GstPad is an object contained by elements that allows links to other elements
- A GstRealPad is a Real link pads
- A GstGhostPad is a Pseudo link pads
- A GstElement in an Abstract base class for all pipeline elements
- A GstBin is a Base class for elements that contain other elements
- A GstPipeline is a Top-level bin with scheduling and pipeline management functionality.

3.2 GStreamer Elements

GStreamer elements can be classified as sources, sinks and filters (or codecs) as depicted below. Elements can exchange data buffers and events through pads.
A source pad produces data buffers, while a sink pad consumes data buffers.



Linked elements



Elements can be linked together by connecting their pads. Furthermore, they can be grouped in a bin, which can be seen as a complex element from the outside, thereby providing a hierarchical structure in the GST elements available to application programmers.

3.3 GStreamer Object Hierarchy

The Gstreamer framework is based on Glib, a C library that allows object-oriented code to be developed. What follows is a simplified representation of the GStreamer class hierarchy.

```
GObject
      |----→GstRegistry
      |          |----→GstXMLRegistry
      |
      |----→GstObject
                 |----→GstPad
                 |          |----→GstRealPad
                 |          |----→GstGhostPad
                 |
                 |----→GstElement
                            |----→GstBin
                            |----→GstPipeline
```

4 GSTREAMER – IL INTEGRATION

This section discusses how the OMX IL API can be used by GStreamer (GST) elements to access multimedia components.
The relationship between GST elements and OMX components is analyzed, in terms of states, data structures and methods. We analyze similarities and differences between GStreamer and OpenMAX IL and we suggest a simple approach for using OpenMAX base profile components inside the Linux GStreamer multimedia framework.

As shown in Figure 1, a GST element can use the OMX core to load an OMX component. Through the OMX IL API, a GST element can manage allocation and exchange of data buffers with an OMX component as well as configure component operating parameters. Using OpenMAX IL terminology, a GST element is an IL client.
GST element pads have their logical counterpart in OMX component ports, with the main difference being that pads can be added dynamically to a GST element, whereas component ports are static and can only be enabled or disabled at runtime.

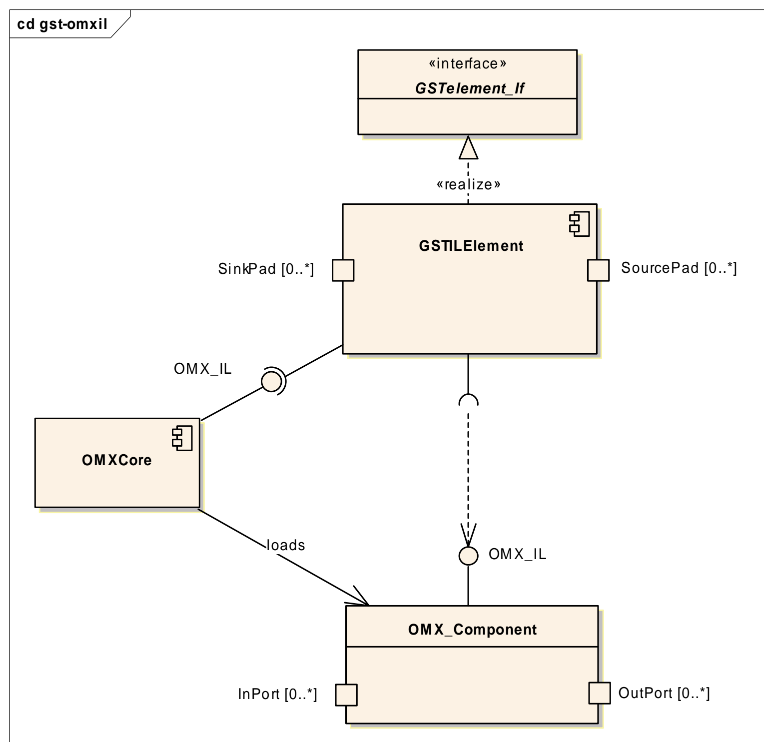


Figure 1: Mapping GStreamer elements to OMX components.

Differences in features and nomenclature between the two environments are also shown in Table 1.

GStreamer	OpenMAX IL	Remarks
element	component	The basic computing or interface block (granularity is not defined in either cases)
sink pad	input port	Interface for receiving data buffers
source pad	output port	Interface for transmitting data buffers

property (or capability)	param & config	Init time and run time parameters
bin, pipeline	-	No concept of chains of components in OpenMAX (except for resource management purposes)
plugin	-	Container for GST elements (dynamic library)
event	event	GST has an event propagation feature, which in OMX must be implemented by the IL client
buffer	buffer	A data unit with associated meta-data such as offset, timestamps,…
caps	port definition	The capabilities of pads/ports, including the supported data format

Table 1: GStreamer and OpenMAX IL features compared.

The next sections analyze differences in data structures and methods between the two environments. The reader interested in showing how OpenMAX Integration Layer API is used in practice by GStreamer elements can directly jump to section 4.4.

4.1 GST element and OMX component states

States of Gstreamer elements and OMX components are rather similar and a comparison can be found in Table 2.

GST element state	OMX component state	comments
`GST_STATE_VOID_PENDING`		
`GST_STATE_NULL`	`OMX_STATELOADED`	Initial state with no resources allocated
`GST_STATE_READY`	`OMX_STATEIDLE`	Ready with resources allocated
`GST_STATE_PLAYING`	`OMX_STATEEXECUTING`	Processing buffers
`GST_STATE_PAUSED`	`OMX_STATEPAUSE`	Paused with resources allocated, can queue buffers
	`OMX_STATEWAITFORRESOURCES`	HW resource conflict pending
	`OMX_STATEINVALID`	Component corruption

Table 2: GST element and OMX component states.

OpenMAX IL defines two states, which do not have counterparts in the Gstreamer world, namely `OMX_STATEWAITFORRESOURCES` and `OMX_STATEINVALID`. The first one is entered by a component in case of hardware resource conflicts, whereas the second one is a pseudo-state, which is entered in case of unrecoverable errors and which results in the OMX component being unloaded by the IL client.

One notable difference between GST and OMX environments relates to state transitions. In fact, it is possible for a GST application to request a state transition for a GST element from NULL to PLAYING. The GStreamer framework then takes care of managing all the intermediate element transitions (NULL to READY, READY to

PAUSED and finally PAUSED to PLAYING). In OpenMAX, state transitions are specified in [1], page 44. In that context, an IL client cannot request a component to go from LOADED to EXECUTING state, as this command would return an error.

4.2 Comparison of methods

In Table 3 we compare common methods that are used by GST applications and by IL clients, respectively. As stated above, a GST element that uses the OMX IL API becomes an IL client. Therefore the implementation of a GStreamer method inside an element, may call the corresponding OMX IL function.

Purpose	GStreamer method	OMX IL method
Initializing the environment	`gst_init()`	`OMX_Init()`
Instantiating an element	`gst_element_factory_make()`	`OMX_GetHandle()`
Changing an element state	`gst_element_set_state()`	`OMX_CommandSendCommand (<state>)`
Connecting two elements	`gst_element_link()` or `gst_element_link_pads()`	`OMX_SetupTunnel()`
Setting/getting an element property at init time	`g_object_set()` `g_object_get()`	`OMX_SetParameter()` `OMX_GetParameter()`
Setting/getting an element property at run time[1]	`g_object_set()` `g_object_get()`	`OMX_SetConfig()` `OMX_GetConfig()`
Buffer structure allocation[2]	`gst_buffer_new()`	`OMX_UseBuffer()`
Buffer structure and data buffer allocation	`gst_buffer_new_and_alloc()`	`OMX_AllocateBuffer()`
Free a buffer	`gst_buffer_unref()`	`OMX_FreeBuffer()`

Table 3: Comparison of methods used by applications.

[1] As a matter of fact, GStreamer is much richer in terms of interfaces and methods to define properties for elements and pads, also dynamically, here we only mention g_object_get/set for simplicity.

[2] `gst_buffer_new()` has no input parameters and it only allocates a structure to hold buffer metadata; `OMX_UseBuffer()` allocates the equivalent buffer header but a pointer to an already allocated memory region is passed as an input parameter to the function.

4.3 Comparison of data structures

In this section we compare the main data structures used by GStreamer and OMX IL. In general, GStreamer provides a superset of the functionality found in OMX IL, but using OMX IL in GST elements is still possible without introducing big limitations.

4.3.1 Buffers

The following table compares buffer structures found in Gstreamer (GstBuffer) and OpenMAX IL (OMX_BUFFERHEADERTYPE), respectively.

Purpose	GstBuffer	OMX_BUFFERHEADERTYPE
size of the structure in bytes		`nSize`
OMX specification version information		`nVersion`
pointer to buffer data	`data`	`pBuffer`
size of buffer data	`size`	`nFilledLen`
max size of this buffer	`maxsize`	`nAllocLen`
Timestamp	`Timestamp`	`nTimeStamp`
Tick count	`Duration`	`nTickCount`
start offset of valid data in bytes from the start of the buffer	`Offset`	`nOffset`
	`offset_end`	
	`free_data`	
Pointer to application private data	`buffer_private`	`pAppPrivate`
	`_gst_reserved`	
Pointer to platform specific data		`pPlatformPrivate`
pointer to any data the input port wants to associate with this buffer		`pInputPortPrivate`
pointer to any data the output port wants to associate with this buffer		`pOutputPortPrivate`
The component that will generate a mark event upon processing this buffer		`hMarkTargetComponent`
		`pMarkData`
buffer specific flags		`nFlags`
The index of the output port (if any) using this buffer		`nOutputPortIndex`
The index of the input port (if any) using this buffer		`nInputPortIndex`

Table 4: GST and OMX IL buffer structures compared.

Buffers contain the data that will flow through the GST pipeline. In GStreamer a source element will typically create a new buffer and pass it through a pad to the next element in the chain by calling gst_pad_push().
A GstBuffer consists of a pointer to a piece of memory, the size of the memory, a timestamp for the buffer, a refcount that indicates how many elements are using this buffer.
This refcount will be used to destroy the buffer when no element has a reference to it.
After using a buffer an element can return the buffer to its provider immediately or pass it to the next element but finally it comes back to its allocator.

GStreamer provides functions to create custom buffer create/destroy algorithms, called a GstBufferPool[3]. This makes it possible to efficiently allocate and destroy buffer memory. It also makes it possible to exchange memory between elements by passing the GstBufferPool. A video element can, for example, create a custom buffer allocation algorithm that creates buffers with XSHM as the buffer memory. An element can use this algorithm to create and fill the buffer with data.
The simple case is that a buffer is created, memory allocated, data put in it, and passed to the next element. That element reads the data, does something (like creating a new buffer and decoding into it), and unreferences the buffer. This causes the data to be freed and the buffer to be destroyed. A typical MPEG audio decoder works like this.
A more complex case is when the filter modifies the data in place. It does so and simply passes on the buffer to the next element. This is just as easy to deal with. An element that works in place has to be careful when the buffer is used in more than one element; a copy on write has to make in this situation.
Buffer can be allocated at any time during the execution, when ever needed.
In contrast, in case of OpenMAX IL, a buffer can be allocated by its client, or by either of the components connected using a tunnel. If the IL client allocates a buffer then it requests the component to use the buffer and to allocate the buffer header; the IL client can also request the component to allocate a buffer along with its header.
In case of data tunneling a negotiation is made among components that implement the interop profile, in order to decide which component will allocate the buffer and which will use it. When ports become disabled or flushed, the buffers are returned to its allocator and freed subsequently when state changes from OMX_StateIdle to OMX_StateLoaded.
Buffers are allocated only when state is changing from OMX_StateLoaded to OMX_StateIdle.

4.3.2 Buffer Flags

On GStreamer as well as in OpenMAX IL data buffers have flags which carry properties associated with the buffer itself. From the table below, it is clear that buffer flags in the two environments are hardly comparable.

Purpose	GStreamer	OMX IL
The buffer is read-only	GST_BUFFER_READONLY	
The buffer is a sub buffer	GST_BUFFER_SUBBUFFER	
Buffer is not a copy of another buffer	GST_BUFFER_ORIGINAL	
Do not try to free the data when this buffer is unreferenced	GST_BUFFER_DONTFREE	
The buffer holds a key unit, a unit that can be decoded independently of other buffers	GST_BUFFER_KEY_UNIT	
The buffer should not be ref () ed, but copied instead before doing anything with it (for specially allocated hw	GST_BUFFER_DONTKEEP	

[3] This class is no longer found on GStreamer 0.10. The same functionality might be achieved using gst_pad_alloc_buffer and gst_pad_set_bufferalloc_function.

buffers and such)		
The buffer has been added as a field in a GstCaps	GST_BUFFER_IN_CAPS	
Additional flags can be added starting from this flag	GST_BUFFER_FLAG_LAST	
Sets EOS when it has no more data to emit		OMX_BUFFERFLAG_EOS
STARTTIME flag is directly associated with the buffers timestamp		OMX_BUFFERFLAG_STARTTIME
Sets the DECODEONLY flag on any buffer that should shall be decoded but should not be displayed		OMX_BUFFERFLAG_DECODEONLY
Set when the IL client believes the data in the associated buffer is corrupt		OMX_BUFFERFLAG_DATACORRUPT
The buffer contains exactly one end of frame		OMX_BUFFERFLAG_ENDOFFRAME

Table 5: Buffer flags.

4.3.3 Pads and ports

Pads have capabilities that describe media type formats, so that only pads with compatible capabilities can be linked.
Similarly, OMX ports parameters can be found using the OMX_PARAM_PORTDEFINITIONTYPE structure defined in [1] at page 64. Port compatibility is checked among OMX interop profile components during the data tunneling setup phase.

4.3.4 Events

Both Gstreamer elements and OpenMAX IL components can generate asynchronous events, as detailed below.

Unknown event.	GST_EVENT_UNKNOWN	OMX_EventMax
An end-of-stream event.	GST_EVENT_EOS	OMX_EventBufferFlag
A flush event.	GST_EVENT_FLUSH	
An empty event	GST_EVENT_EMPTY	
A discontinuous event to indicate the stream has a discontinuity.	GST_EVENT_DISCONTINUOUS	
A quality of service event	GST_EVENT_QOS	
A seek event	GST_EVENT_SEEK	
A segment seek with start and stop position	GST_EVENT_SEEK_SEGMENT	
The event that will be emited when the segment seek has ended	GST_EVENT_SEGMENT_DONE	
A size suggestion for a peer element	GST_EVENT_SIZE	
Adjust the output rate of an	GST_EVENT_RATE	

element		
A dummy event that should be ignored by plugins	GST_EVENT_FILLER	
An event to set the time offset on buffers	GST_EVENT_TS_OFFSET	
Mainly used by _get based elements when they were interrupted while waiting for a buffer.	GST_EVENT_INTERRUPT	
Navigation events are usually used for communicating user requests, such as mouse or keyboard movements, to upstream elements.	GST_EVENT_NAVIGATION	
A new set of metadata tags has been found in the stream.	GST_EVENT_TAG	
Component has successfully completed a command		OMX_EventCmdComplete
Component has detected an error condition		OMX_EventError
Component has detected a buffer mark		OMX_EventMark
Component has reported a port settings change		OMX_EventPortSettingsChanged
Component has been granted resources and is automatically starting the state change from OMX_StateWaitForResources to OMX_StateIdle		OMX_EventResourcesAcquired

Table 6: Events.

4.4 Initialization

In order to show how Gstreamer elements and OMX components interact at init time, we use the sequence diagram in Figure 2 as a reference.
Here, both initialization and buffer allocation steps are shown. It should be noted that this is just an example of how OMX could be used from GST elements and the buffer management strategy could be different in other implementations.
Having used a push model in this example, we adopt the convention that the OMX component output port is the buffer allocator.

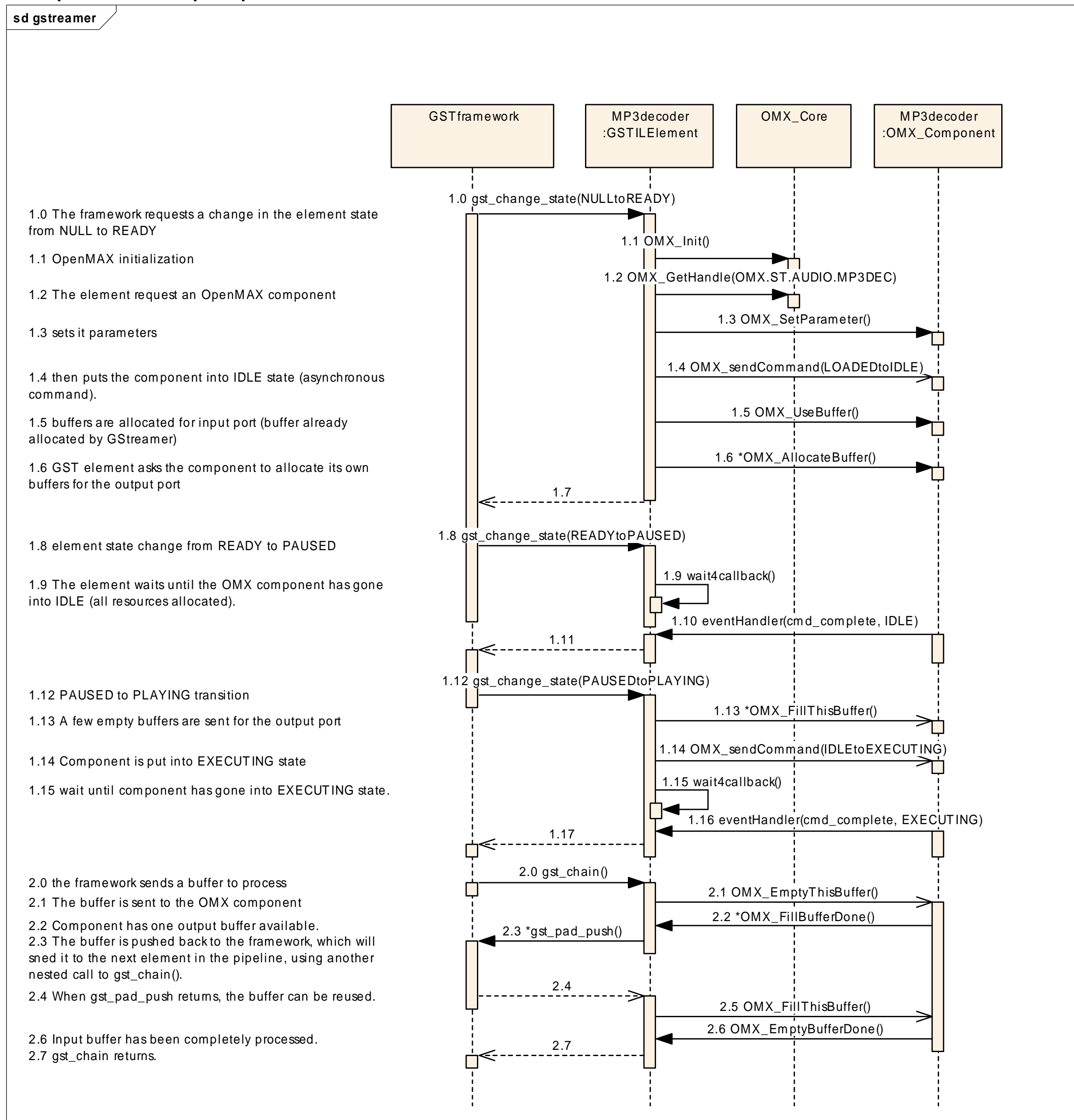


Figure 2: Data flow sequence diagram.

The GStreamer model for passing data among elements is usually a push model, i.e., when a component has finished producing an output buffer, it sends it through its source pad to the downstream element by calling the gst_pad_push function. However, a pull model can also be supported, if so desired.
The gst_pad_push() function is then handled by the gst_chain() entry point of the downstream element and returns as soon as the buffer has been consumed, unless

in-place processing occurs in the component. In this case the same buffer is reused and further sent down in the pipeline by incrementing a refcount.
Referring to Figure 2, when an element goes from NULL to READY state, the corresponding OMX component handle can be requested (steps 1.0 to 1.2). In the OMX_GetHandle() function the GST element can pass the IL core and the OMX component a pointer to a private structure (pAppData). In this private structure, the GST element may store relevant parameters like upstream and downstream GST elements, pads as well as OMX port information and buffer pointers. A pointer to pAppData is returned by the OMX component during callbacks, so that the GST element can retrieve all references to buffers, ports and pads.
If a valid OMX component handle is returned, then the element can configure the component parameters by using the OMX_SetParameter macro with the correct data structure, which depends on the component domain and type[4]. At this point (step 1.4) the GST element can request the OMX component to go to the IDLE state.
Since this is an asynchronous call, it will return immediately and buffer allocation must take place at this point.
In our example, we assume to use a buffer that was previously allocated by the GStreamer framework for the component input port. Therefore the OMX_UseBuffer function is called, which will allocate an OMX buffer header. The OMX buffer header will point to the same memory area of the originally GST-allocated buffer.
In step 1.6 the allocation of buffers for the component output port is performed by calling OMX_AllocateBuffer() multiple times, depending on the number of buffers that the component requires. In this case, the OMX component will allocate both a memory area for the data and an OMX buffer header pointing to it.
If the allocation process is successful, the OMX component will eventually go to the IDLE state and generate a callback. However, the GStreamer element just returns after calling the buffer allocation functions and does not wait for the callback at this point (this is to allow buffer negotiation among multiple GST elements when data tunneling is used to connect OMX components).
When the GST element is requested by the framework to go from READY to PAUSED state by the framework, then it must make sure that the component is in the IDLE state (all resources have been allocated) and it will then block until an event is received that indicates the new expected state for the component (step 1.10).
Alternatively in step 1.8, the GST element may command the OMX component to go to PAUSED state, but this is implementation dependent.
The GST element can now go to the PLAYING state where it can start processing input buffers (1.12). At this point (1.13), the GST element can pass a number of buffers to be used by the OMX component for its output port. The OMX component is now ready to process input data and therefore a command is sent by the GST element to go to EXECUTING state. Since the command is asynchronous, the GST element waits until the corresponding callback is received, which indicates the component has entered the expected state (1.16).
Then, the GST element can return from the gst_change_state function (1.12 → 1.17) and is now ready to receive data buffers to be processed.

[4] Depending on the implementation of the OMX component it might be required to specify some port properties that depend on the input media stream. These properties might only be obtained by parsing the stream. Properties like the dimensions of video frames (width x height) might be only required as a parameter by the OMX component. This means it might be needed to get all the way to the playing state before setting the parameters of the OMX component.

When one data buffer arrives to the GST element as a parameter of the gst_chain() function (2.0), an OMX buffer header is filled in with relevant information (timestamp, size,…) derived from the GSTBuffer structure and passed down to the component by means of the OMX_EmptyThisBuffer asynchronous function (2.1).
As soon as one output buffer is available form the output port of the OMX component, the OMX_FillBufferDone callback is invoked on the GST element. As anticipated above, the pAppData parameter can be used to retrieve the application private structure where all relevant information can be found. The OMX buffer is also passed as a parameter to the callback. One of the GST buffers associated with the GST element source pad is then prepared; this step does not involve memory copies, only buffer metadata will be copied from the OMX buffer header. The GST element then pushes this buffer through the source pad with gst_pad_push (2.3). The gst_pad_push function will pass the buffer to the downstream element, where the gst_*_chain() entry point will be invoked. As soon as this buffer has been completely consumed by the downstream element, the gst_pad_push function will return (2.4). The same buffer can then be recycled by the OMX component, therefore the GST element calls OMX_FillThisBuffer (step 2.5).
The gst_chain() function (step 2.0) returns as soon as the input buffer has been completely consumed by the GST element – and by the underlying OMX component. This corresponds to the OMX_EmptyBufferDone callback being generated by the OMX component (step 2.6).

4.5 Data flow

In order to show how buffers are managed in a pipeline of IL-enabled GST elements, let us consider a simple MP3 playback example.

In the sequence diagram below, the following notation is used for buffers: A represents a GStreamer buffer, whereas A' is an OpenMAX IL buffer. Although the data structures are different, the payload of A and A' is the same and no memory copy is involved in translating buffers.

It is assumed that the MP3 decoder OMX component output port allocates buffers and such buffers are passed by the IL client to the audio renderer OMX component input port. The sequence diagram shows how the OMX API is used to guarantee correctly synchronized buffer circulation among OMX component ports using the base profile.

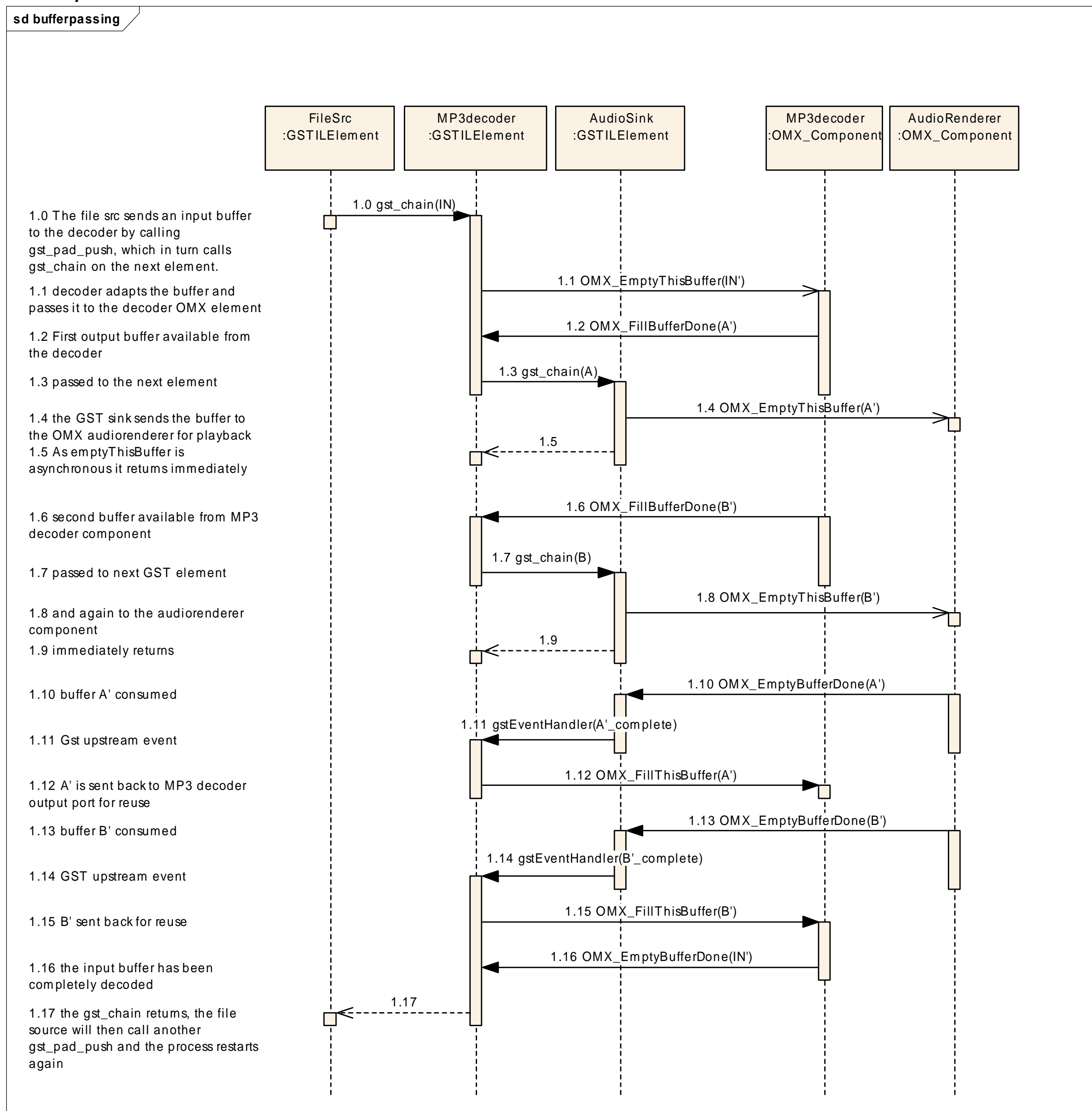


Figure 3: Buffer passing using OMX base profile.

It should also be noted that IL v.1.0 does not specify a file reader component, therefore reading a file and parsing its content is accomplished by using a standard GStreamer “filesrc” element.

When the file source GST element starts reading the input file, it fills a buffer (IN), which is passed to the next downstream element by invoking the gst_pad_push() function of the GStreamer framework. This call then results in the gst_chain() functionentry point of the next element being invoked (step 1.0 in Figure 3). The MP3 decoder element, prepares an IN’ OMX buffer and passes it to the MP3 decoder component input port using OMX_EmptyThisBuffer (step 1.1). It is assumed that all the GST elements and corresponding OpenMAX components are in the PLAYING/EXECUTING state, meaning that buffer allocation has been made, for example using the approach of Figure 2. Component ports are also all enabled. In particular, the MP3 decoder component output port has been passed two empty buffers (A’ and B’) to be filled with raw PCM data as a result of MP3 decoding (not shown for simplicity).

As soon as the first output buffer of the MP3 decoder OMX component is ready, it is passed to the MP3 decoder GST element by means of the OMX_FillBufferDone callback (step 1.2). This buffer is translated into a GST buffer and passed on to the audio sink GST element (step 1.3). This in turn calls OMX_EmptyThisBuffer on the OMX audio renderer component. Since the call is asynchronous, it does not block until the buffer has been consumed and returns immediately. And so does the gst_chain() (step 1.5).

As soon as a second buffer B’ is ready on the MP3 OMX component output port, it is sent to the GST element using the OMX_FillBufferDone callback (1.6) and, again, the buffer is passed to next element (1.7) and then to the OMX audio renderer component for playback (1.8).

Let us now see how synchronization is achieved. When the audio renderer OMX component is finished processing the first buffer A’, it sends an OMX_EmptyBufferDone callback to its GST audio sink; this means that the buffer has been completely consumed and can be reused on the MP3 decoder output port.

When the GST element receives the OMX_EmptyBufferDone callback, it sends a GST event to the upstream element (step 1.11). The MP3 decoder then calls an OMX_FillThisBuffer on the MP3 decoder OMX component, passing A’ as a parameter (1.12).

Similarly, as soon as B’ is consumed, the same process is followed (1.13 to 1.15).

When the MP3 decoder OMX component is finished processing the IN’ buffer, it sends a OMX_EmptyBufferDone callback (1.16), so that the initial gst_chain function returns (1.17).

The file source element will then send a new buffer for decoding and the process loops until EOF is reached.

4.6 Example of IL-enabled GST plugin set

In this section we describe an example of plugins that use the OMX IL API.
Figure 4 shows the `gst-editor` that is used to build a simple pipeline for an MP3 playback application.

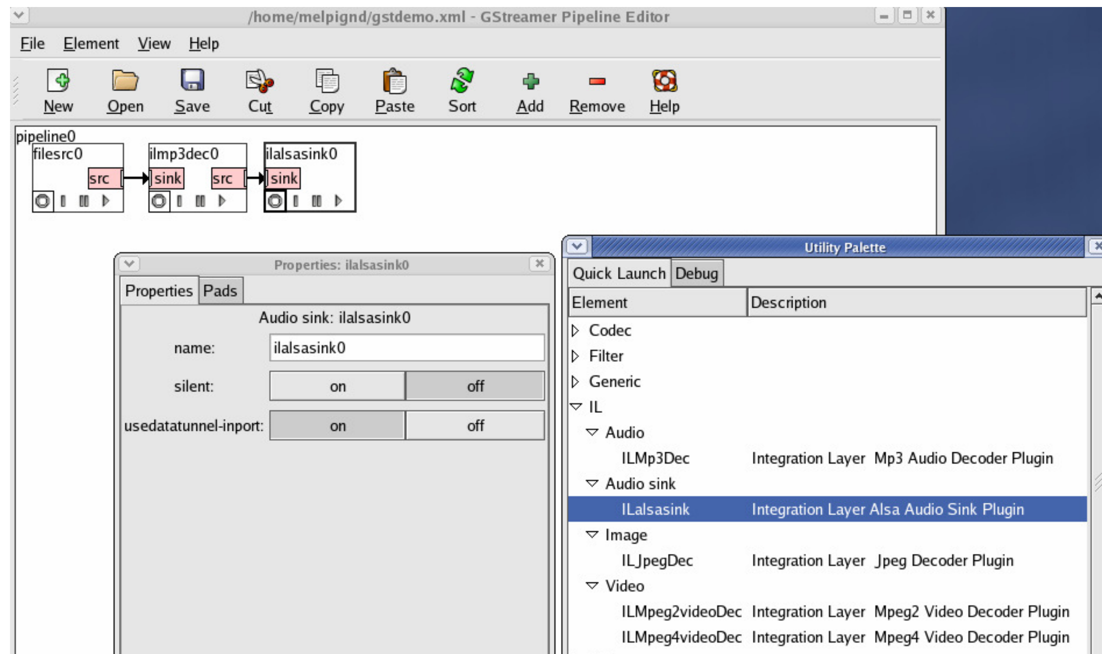


Figure 4: IL-enabled Gstreamer plugins as seen in gst-editor.

Elements that use the OMX IL API have been grouped under the “IL” category and then further hierarchical levels have been created for Audio, Video and Imaging domains.
In order to use data path optimization, a property has been added to GST IL elements, indicating if OMX data tunneling should be used for a pad. The next section explains this optimization technique, whose main advantage is power consumption reduction on mobile platforms.

4.7 Optimizations

One of the key optimizations enabled by the OpenMAX Integration Layer API is data tunneling. With data tunneling, two component ports can be connected so that the output buffers of one component can be directly sent to the next component input port, without being returned to the IL client. When a data tunnel is setup between two components whose implementation run on a HW or DSP-accelerated platform, it is possible to exploit proprietary communication mechanisms to transfer data (e.g. DMA or shared memory) among them. This translates into considerable power consumption savings.
An example of such an optimization is shown in Figure 5.

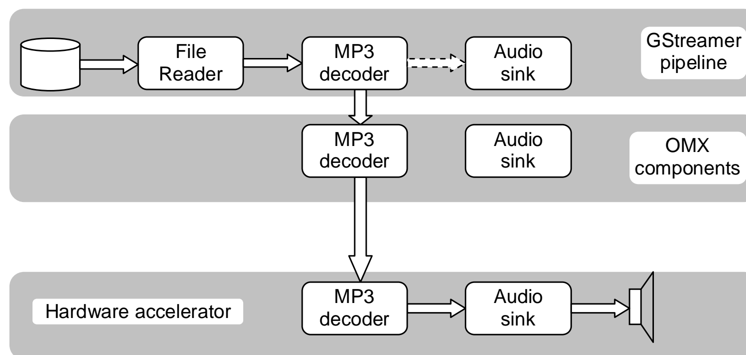


Figure 5: Using OMX IL data tunneling with GStreamer.

The application programmer uses the GStreamer API to build a pipeline using IL enabled elements. In the example above, a simple MP3 playback application is considered.
The MP3 decoder and audio sink elements map onto corresponding OMX components. In turn, such components are simply wrappers on top of a hardware accelerator, where the actual processing takes place. OMX components control hardware accelerator blocks through dedicated device drivers, but this aspect is outside the scope of this document.
During the creation process of the Gstreamer pipeline[5], each element can use the OMX IL API to create data tunnels with a downstream component, if so desired. This behaviour can be controlled by the application through a specific GST element property. If OMX components being tunneled are compatible, a data tunnel can be established, which can be implemented by setting up direct communication between the MP3 decoder and the audio sink inside the HW accelerator.
When a data buffer is routed by the GStreamer framework from the file reader element to the MP3 decoder element (as in the example above), the buffer payload is encapsulated in an OMX buffer and passed to the hardware accelerator. Since no output buffer will be returned by the component to the IL client, the `gst_pad_push()` function will never be called by the GStreamer MP3 decoder element, which acts as a sink as far as the GStreamer data flow in concerned.

[5] In particular, in the gst_*_link() entry point of the GST v.0.8 element or the equivalent gst_pad_set_setcaps_function and gst_pad_set_getcaps_function functions in GStreamer 0.10.

5 Conclusions

In this white paper we have discussed how the OpenMAX Integration Layer API specified by the Khronos group can be used in the Linux GStreamer multimedia framework.
The interaction among GST elements and OpenMAX base profile components has been shown for a simple MP3 decoding example.
By just providing GST plugins with elements that use the Integration Layer API, it is possible to exploit standardized access to multimedia components in a platform independent way.
If hardware acceleration is available, OpenMAX IL provides a convenient mechanism to exploit it, including data path optimization features that concur in reducing power consumption and offloading the host CPU.

6 References

[1] http://www.khronos.org/openmax, OpenMAX Integration Layer 1.0 API specification, Khronos Group.
[2] http://gstreamer.freedesktop.org, Gstreamer User's Manual, v.0.8.