AN EFFICIENT HOST/CO-PROCESSOR SOLUTION FOR MPEG-4 AUDIO COMPOSITION

L. Le Bourhis, G. Zoia, M. Mattavelli, D. J. Mlynek
Swiss Federal Institute of Technology, Integrated Systems Laboratory LSI, CH-1015 Lausanne Switzerland

Abstract

This paper presents an efficient software host/co-processor architecture for the implementation of MPEG-4 Audio composition. The proposed solution is based on a specific partition between general-purpose tasks and DSP-oriented functionality, thus achieving portability, efficient partitioning of the processing resources and memory management.

1 Introduction

The new MPEG-4 Audio and Systems standards [1, 2] provide extended capabilities for audio processing and composition. MPEG-4 Audio is built around two families of components. The first one is the decoding part (the *Audio layer*) while the second one is built around the BIFS (Binary Format for Scenes [1]) composition nodes (the *Systems layer*). MPEG-4 decoders take MPEG-4 elementary bit-streams, encoded in various formats such as AAC (Advanced Audio Coding [2]) based on state-of-the-art subband coding and shaping, CELP (Code Excited Linear Prediction [2]) for vocal compression and SA (Structured Audio [2]) for synthetic generation and digital audio processing. Once these bit-streams are decoded into decoding buffers, composition buffers are produced. Composition buffers are then processed in BIFS nodes which perform various processing stages on the audio streams so as to generate the output sound. MPEG-4 Systems provides audio extended capabilities, which are much more powerful and efficient than those present in similar multimedia languages such as VRML [7] (Virtual Reality Mark-up Language). It provides mechanisms to perform mixing, delay, 3-D spatial processing or other additional effects, to enhance the audio experience delivered by high-quality coding tools such as AAC (see for instance reference [4]).

However, these new powerful and flexible ways of handling audio rendering require a large amount of processing resources and present new implementation challenges in comparison with the classical audio coding schemes. Moreover, the complexity of audio scene structures requires appropriate strategies for resource partitioning, memory management and an efficient implementation of the various processing nodes [4].

This paper presents an architecture for the implementation of an MPEG-4 Audio Compositor aiming to achieve:

- portability on different host/co-processor platforms
- frame based processing
- flexibility for the integration of different processing node implementations specified by the audio scene at hand.

The results presented in this paper summarize the extensive studies and recent achievements of the authors concerning the implementation of MPEG-4 Systems Audio nodes [4,5]. Over the simple node implementations, a complete scene management needs several other fundamental strategies to be optimized: among them sampling rate conversion, composition buffer structure, synchronization and channel routings. In the first part, we will describe MPEG-4 audio composition features and the mechanisms necessary to build an efficient model for audio composition, avoiding pitfalls created by MPEG-4 new features. Once proper solutions have been found for these issues, it is possible to proceed with the definition of an optimal memory and processing organisation, in order to make it easily portable on a host/co-processor configuration. The second part of the paper reports the implementation and the validation of this Host/Co-processor architecture design.

2 An approach towards an efficient "MPEG-4 Audio Systems" implementation

This section describes general features offered by the MPEG-4 audio compositor, as well as some clues for an efficient implementation. The proposed solutions are optimized for version 1 of the MPEG-4 standard.

2.1 Sampling Rate Conversion

When the MPEG-4 system is operative, decoders produce composition buffers to feed the compositor. According to the decoder and audio content type, the frame length of

Revised manuscript received October 8, 1999

audio samples and the sampling frequency may be different from decoder to decoder according to the values in Table 1.

Sampling Frequency	Composition Unit Length
8000	80
11025	160
12000	240
16000	320
22050	1920
24000	2048
32000	
44100	
48000	
64000	
88200	
96000	

Table 1

As we will see in the description of a functional system, the wide range of possible sampling frequencies and frame lengths introduces relevant problems for the implementation of the composition process. So as to solve these problems, it is useful to develop a suitable model for composition buffers that enable a flexible management of the synchronization. This model must retain all valuable information that is necessary for managing composition (Composition Time Stamps, sampling rate, buffer length). These buffers are created by decoders and accessed by *AudioSource* nodes. If the various children of a node do not produce output at the same sampling rate, then the size of the output buffers of the children do not match. Therefore, the sampling rate of the children's outputs must be brought into alignment in order to place their output buffers in the input buffer of the parent node. The sampling rate of the input buffer for the node must be the fastest of the sampling rates of the children. The output buffers of the children must be re-sampled to be at this sampling rate. The specific method of re-sampling is non-normative.

In the example of Figure 1, the AudioSource at 22 kHz feeds two input channels of the AudioMix node.

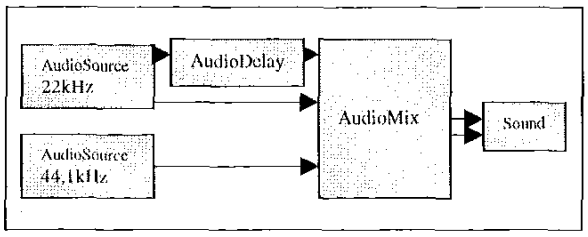

Figure 1 A simple audio scene

According to the standard, two sampling rate conversions are needed, one for the delayed channel and one for the direct stream. Such overhead must be carefully analyzed. One can imagine of having a single sampling rate conversion just after the AudioSource at 22kHz, hence avoiding a dual conversion. The drawback of this solution is the need to store a 44.1 kHz stream in the delay line instead of a 22 kHz stream. Such trade-off must be carefully considered to optimize processing efficiency. A good solution to efficiently handle sampling rate conversions is to design a dedicated sampling rate conversion node (SRC node). This is the solution chosen in the described implementation. It enables the flexible placement of this node anywhere in the scene description, thus leading to its optimal usage. The functional description of this node will be presented later.

2.2 A frame based compositor

When dealing with audio processing, two solutions are possible: working on sample by sample basis or with a frame-based approach. Frame-based solution is the best approach when homogeneous streams are available (i.e. same sampling rates). This solution allows reducing overhead processing, since the same processing is applied to a large number of samples. The problem of this approach in MPEG-4 is the wide range of sampling frequencies allowed in a scene. In our implementation it has been chosen to work on a frame base of 10 ms. This choice allows a good control rate and gives an almost non-noticeable delay for composition. This also solves the problem of buffering information between nodes. This solution leads to the frame lengths reported in Table 2. As one can notice, there remains a problem for 11025 Hz and 22050 Hz. A straightforward solution is to convert the corresponding buffers directly in a more suitable sampling frequency. A tree analysis provides the correct sampling rate choice.

Sampling Frequency	Frame Length
8000	80
11025	Special handling
12000	120
16000	160
22050	Special handling
24000	240
32000	320
44100	441
48000	480
64000	640
88200	882
96000	960

***Table 2** Frame length vs. sampling frequency*

For example if the 22kHz source is supposed to be mixed with a 44.1 kHz channel it could be useful to perform this conversion at the beginning. A drawback of this method is that composition units as defined by MPEG-4 are not entirely consumed at each iteration. Therefore, a specific buffer handling mechanism must be implemented.

2.3 Composition buffers and AudioSource Interaction

2.3.1 *Input buffers structures*

In an MPEG-4 audio scene, decoders send frames of samples to the compositor. This compositor is in charge of managing buffers storing these frames so to render the complete audio scene. Input buffer structures are necessary to handle the samples coming from different decoders and to provide mechanisms to easily identify buffers associated with *AudioSource* nodes. The basic idea is to associate a buffer Id with each channel of a decoder output and provide these Ids to the *AudioSource* nodes so that they can target the correct buffers.

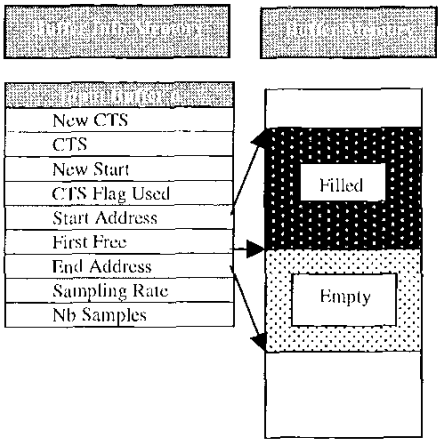

Figure 2 *A single channel buffer structure*

Since there is other information regarding buffers, an informative structure for each buffer is stored in a dedicated memory space illustrated in *Figure 2*. Each address corresponding to these information structures is located in a global buffer table. At the end, there are three memory structures to implement input buffers. The low-level memory is the buffer memory: it is where the samples are stored. A single buffer consists of a circular space reserved in this memory. *Start Address* and *End Address* are stored in its associated Information memory in a dedicated Buffer Info memory. Other specific information is the number of samples available in this buffer and the first free sample. This latest information is used by the decoder to know where to put new samples.

Other information like *NewCTS*, *CTSFlagUsed* are used for temporal synchronization and will be described later. Locations of these buffer information structures are stored in the Buffer Table as described in *Figure 3*.

With this structure when a new buffer is needed, the correct procedure is:

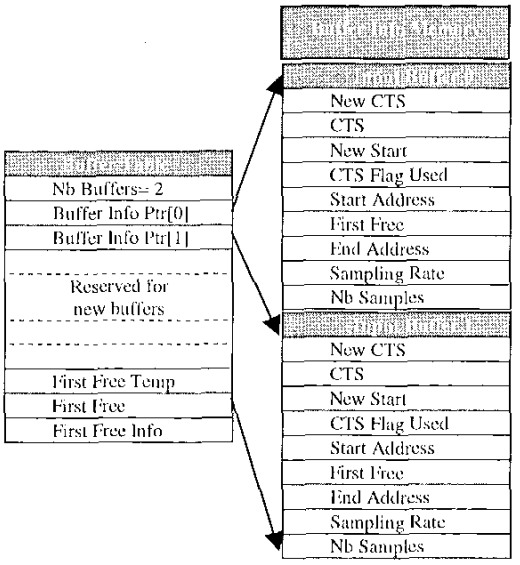

Figure 3 *Buffer table example*

- Request a new buffer in the buffer memory (returns *Start Address*) by looking at *First Free* in the *Buffer Table*
- Reserve a new space in the buffer information memory by looking at *First Free Info* and update the buffer location fields (*Start Address*, *End Address*) in this new structure. Set *First Free* to the *Start Address* and *NbSamples* to zero.
- Update *NbBuffesr* and *Buffer Info Ptr* in the buffer Table along with *First Free*, *First Free Info* and *First Free Temp*

The *NbBuffer* found when this new buffer is created constitutes the buffer Id. It is used as a reference for *AudioSource* nodes. When a specific buffer is targeted, looking in the buffer table with the correct offset returns the location of the specific buffer information structure.

2.3.2 *AudioSource synchronization*

In an MPEG-4 audio scene, temporal synchronization is performed through the use of Composition Time Stamps (*CTS*). These time stamps are not necessarily conveyed in each composition units. When this *CTS* is absent, it means that the composition time is continuous. Whenever the *CTS* is present, the compositor shall take care of its consistency. If there is a difference between the conveyed CTS and the supposed *CTS* (as if it were a continuous stream), some specific actions have to be performed. These actions are essential since all *AudioSource* are independent and must go on working even if a single source is not synchronized. Using *StartTime* and *StopTime* fields of an *AudioSource* node also provides another mean of synchronization.

The next subsection describes solutions to both synchronization issues.

2.3.3 *Composition Time Stamps management*

As we have seen so far, decoders are responsible to feed the audio scene tree with composition buffers (CBs) with associated time information (timestamps), which indicates at what time these buffers have to be composed. This leads to the concept of a synchronized and not synchronized source. When a new buffer frame arrives that does not contain a continuous time information, the audio scene shall take care of this "time jump" and wait for the appropriate time. To implement this system, the compositor needs its own time base reference (current system time) to which it will refer to know whether or not a *CTS* is correct. The storage of decoder timing information is done in Input Buffer Information structure. It is the role of the decoders to update these fields when it is necessary. Each of the input buffers has the following associated information:

- *NewCTS Flag*: set to one means that a new *CTS* is present for the next samples
- *CTS*: *CTS* Value
- *New Start*: indicates the location in the buffer memory where is the first sample corresponding to the new *CTS*
- *CTS Flag Used*: indicates that AudioSource has used the current CTS.

This information is used to synchronize *AudioSource*. When an *AudioSource* is executed, it retrieves timing information corresponding to the targeted buffer. To evaluate synchronization, we need to define two distances: a time distance and a sample distance. The number of samples between the current pointer location in the sample buffer and the address of NewStart gives the sample distance, i.e. the number of samples that are still available from the current time until the time of the new composition time stamp: they could be less than necessary, of course. The time distance is the time between the current time and the time corresponding to the new *CTS*. The time distance is converted on a sample base using the source sampling rate value. The first thing to do when a new *CTS* is present is to see if it is continuous with the previous frames. So, if Sample Distance = Time Distance ± Jitter, the new frame can be considered as synchronous. In this case, the flag *CTSFlagUsed* is raised and the processing goes on normally considering the buffer continuous. When all sources will have processed this new *CTS*, a dedicated buffer monitoring process will set the *NewCTSFlag* off according to *CTSFlagUsed*. On the next call *AudioSource* will not perform any special action since the *NewCTSflag* will be off and the buffer will be seen as continuous.

Problems occur when the new *CTS* does not correspond to a continuous frame. In this case there are different cases:

Case 1: Time Distance > Distance and Distance > Frame Length

There are enough samples before reaching *New Start* so that a complete frame can be processed. In this case *Frame Length* samples are sent. Remaining samples will be processed during a next call. *Current Ptr* is incremented by *Frame Length*.

Case 2: Time Distance > Frame Length and Distance < Frame Length

There are not enough samples before *New Start* and the new *CTS* does not correspond to the current time. In this case the remaining samples (*Distance*) are sent and the frame is padded with zeros to obtain a valid frame. On the next call *Distance* will be zero. *Current Ptr* is set to *New Start*.

Case 3: Time Distance> Frame Length and Distance = 0

The new *CTS* will not be reached during this frame and all the previous samples have already been consumed. In this case, the *AudioSource* generates a frame of zeroes. *Current Ptr* stays on *New Start*

Case 4:Time Distance < Frame Length and Distance = 0

This happens when the synchronization is about to occur during the current frame. The beginning of the frame is filled with *Time Distance* zeroes and the rest is filled with normal samples from the input buffer. *Current Ptr* is incremented by *Frame Length – Time Distance*. *CTSFlagUsed* is raised.

Case 5:Time Distance < Distance and Distance < Frame Length

This happens when there is a short desynchronization resulting in a *Distance – TimeDistance* zero padding in the middle of the frame.

Other cases like *Time Distance < Frame Length* and *Distance > Time Distance* are not handled since this constitutes an error. This would mean that the decoder produced a frame that is overlapped in time with the previous one.

2.3.4 *StartTime and StopTime management*

StartTime and *StopTime* are used in time sensitive nodes (like *AudioSource* and *AudioClip*). These fields give control over the node's activity. *StopTime* and *StartTime* are processed according to current time and frame's end time (=current time + frame duration). An important notion to have a working model is to associate a state with *AudioSource* Nodes. A node can be running or stopped. The system can be seen as a state machine since its output depends on the source state. According to the current

StartTime and *StopTime* values, the system decides on the output state and a correct frame is issued.

2.4 Channels Routing – A Common Node Representation

2.4.1 Channels addressing

The object-oriented nature of MPEG-4 scenes requires a flexible implementation. Dynamic node modification, insertion and deletion shall be handled correctly to provide the maximum interactivity. A good insight into this problem is when dealing with node connections. Linking nodes is one of the most important aspects of a compositor implementation.

MPEG suggests to use static buffers between all nodes as flow regulation mechanisms. Adopting this solution leads to a correct, but memory consuming implementation. Conversely the goal of our implementation is to minimize buffer usage and only use buffers on a temporary basis. This means that memory is allocated only in specific conditions and for a single execution cycle. This combined with a constant frame-based processing (10 ms) allows to drastically reduce memory usage.

Progression of samples in the audio tree is done by means of pointer passing. A node accesses its input samples by retrieving their location from its children. In this case, the only information a node has to provide is the location of its output. When a node does not perform a specific processing on the samples (i.e. *AudioSwitch*), it only has to copy its input pointers to the correct channel output pointer. For example an *AudioSwitch* gets the address of the children nodes that have to be passed and just provide these address to the parent nodes. This solution provides a great implementation advantage since no memory access is performed.

When temporary memory is required, a dedicated procedure allows locking some memory in the buffer memory for the current frame cycle. For example this occurs when two channels at 44.1kHz are mixed. A temporary buffer of 441 samples is locked in the memory where the *AudioMix* node outputs the mixing result. When the tree has been completely processed this memory is freed using a buffer monitoring mechanism.

To implement this processing, nodes shall contain the following information:
NbChildren: number of children.
ChildrenIdList[NbChildren]: children Ids
SamplePtr[i]: pointer to available samples for channel *i*.
Base[i]: base address of the circular buffer for channel *i*.
Length[i]: length of the buffer for channel *i*.

2.4.2 Channels activity

To further reduce memory usage, it is useful to avoid sending empty frames (filled with zeroes) through the tree. For example if an *AudioSource* is after its *StopTime*, there is no need to create a frame with zeroes. So as to optimize this processing, each channel has an activity flag associated with it. When normal processing occurs, this flag is on, whereas if no valid information is available it is set to off. Proper handling of this flag in the subsequent node allows saving a relevant amount of processing; there is no need to perform a mixing operation between two channels of zeroes. Some nodes can be reluctant to this, as it is the case for *AudioDelay* nodes. When its child channel has no activity, it still has to put zeroes in its delay line. At its output a detection mechanism (checking if the output frame is empty) can allow to get the "off status" back and to avoid propagating again an empty frame.

2.5 Node information structures

In the scene description, nodes have a parent -> child relationship. This relationship must be translated in a proper format. Since there are several node types, it is not possible to implement exactly the same information structure for each node. Specific information for each node need to be stored in a "Node information memory". Like for the input buffer management, a node table references the addresses of all node information. The reference is built around the node Id. Providing the table with a node Id, it returns the node information location. By this system, children of a node are listed as node Ids.

This node table also stores the node levels. This level is used to sort the table. Level-ordered nodes are then executed from the lowest (*AudioSource*) to the highest (*Sound*).

The last information contained in this table is a pointer (*FirstFree*) to the first free byte of node information memory. This pointer is used when more space needs to be reserved for new nodes. At the end, the node table contains the following information:
NbNodes: number of nodes in the scene
FirstFree: points to the first byte of free node info memory
NodeId[i]: Node Id *i*
NodeInfoPtr[i]: Pointer to the location of node *NodeId[i]*
NodeLevel[i]: Level of *NodeId[i]*

In the MPEG-4 object-oriented system, audio nodes can be linked in any way so that there is no particular assumption that can be made out of a node type depending on its parent. This feature requires having a common node structure that will be the basis for node information exchange. This information is located at the same place in the node and constitutes what its parents can know:
NodeId: node identification number
NodeType: node type
FrameLength: frame length in samples
NumChan: number of output channels
ChannelsActivity[i]: channels activity flags

OutSamplePtr[i]: output sample pointer for channel *i*
OutBase[i]: base address for channel *i*
OutLength[i]: buffer length for channel *i*

3 An Implementation of MPEG-4 audio BIFS on a co-processor architecture

The objective of the described implementation is to develop a portable software C model of the co-processor. This model aims to have a specific implementation oriented memory organization. A flat memory model has been used for the co-processor. Every information is stored in tables and accessed with offsets. There are no complex structures such as pointers of pointers of arrays. This option has been chosen to ease code portability to any DSP platform. Another strategy has been to base the host/co-processor relationship on a strict simple communication mechanism: data transfers between these instances are command frames. Data are stored as integer values and can be manipulated using floating-point arithmetic. Creating an object from this implementation allows its integration into a more sophisticated environment as it will be shown in a demonstration example.

3.1 Co-processor Overview

3.1.1 Memory organization

The co-processor memory is split in two main structures. The first one is used to implement nodes representation and the second deals with buffers as shown in Figure 4. Since the objective of this implementation is to provide a flat memory model the different structures are declared as simple arrays. Accessing a specific piece of information is done by offset addressing. A global file contains all the offsets necessary to correctly address these arrays.

Memory structures are re-sizable by constant declarations:

DspNodeMemSize	2048
DspNodeTableMemSize	256
DspBuffersInfoMemSize	2048
DspBuffersTableMemSize	32
DspBuffersMemSize	96000

Other audio scene elements strongly influence the memory requirements, like the number of nodes in the scene, the number of audio channels a node can receive or generate. *NbChannels* is the most relevant parameter for node occupancy. For example, a matrix in an *AudioMix* node can take 256 locations for an 8x8 mixing operation whereas it will require a static reservation of 65,536 locations for a 256x256 mixing.

3.1.2 Command interpreter

As previously explained, the co-processor model only receives commands from the host. Processing of these commands is performed in a specific procedure. Commands, for example, enable to receive composition buffers or to update node information. Relevant actions taken according to the command type are described below:

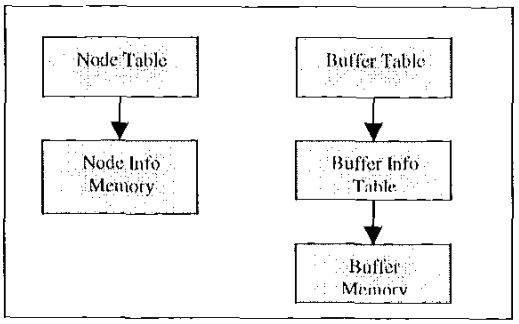

Figure 4 *Co-processor memory structures*

Initialise Node Memory: this command is used to send an entire Node Memory description. The scene can be compiled directly by the host. It process offsets and creates a valid node memory array. This is equivalent to loading a complete scene description.

Buffer Frame: this command has been defined to send a composition buffer to the input buffer memory of the coprocessor. According to buffer Id the program gets the correct buffer information pointer from the buffer table. If *NewCTS* is on, the corresponding buffer fields are updated with the *CTS* value contained in this frame and *NewStart* received the address where the first of these samples will be put. When all the samples have been copied starting from *FirstFree*, *NbSamples* and *FirstFree* fields are updated.

Node Info Update: this command has been defined to update node information fields. It can be called to implement BIFS update or by a specific application hence providing user interactivity as it will be shown in our demonstration example.

Note that the presence of *FrameLength* is maintained even if the frame length is fixed. The purpose is to maintain consistency through all command frames. This is useful when implementing an automatic mechanism to convey command frames (DMA for example).

Initialize a new buffer: this command has been defined to create a new buffer as described in the section dealing with input buffer mechanisms.

Initialize node table: can be used in conjunction with initializing node info memory. It sends the node table associated with the information contained in the node information memory.

Send a node description: this command is called when the host wants to send the description of a new node.
Node Information is stored in the corresponding memory according to the first free pointer stored in the node table.

Upon completion, Node Table is also updated according to the received node's level and Id.
It is then necessary to sort again the table by node level, therefore other functions have been defined.
Start Processing: This function toggles the co-processor's tree processing on.
Stop Processing: This command toggles the co-processor's tree processing off.

3.1.3 Single Co-processor Cycle

When using the presented compositor, processing one frame from each input buffer is performed in three basic steps (cycle processing can be triggered with a dedicated timer generating interruption each 10 ms). The first step is to process all the commands. Supposing these commands are stored in a FIFO-style buffer using non-intrusive mechanisms (DMA for example), this allows having a refresh rate of 10 ms. The second step is to execute all nodes according to their level. Since the node table is ordered according to these levels, the procedure is to take *NodeId* as they appear in the table and then call the appropriate function passing their Id. Once nodes have been processed, a buffer monitoring mechanism updates *CTS flag* fields to check whether or not new *CTS* have been used. Another task is to update the number of samples available in the different buffers and discard temporary buffers.

3.2 Audio BIFS node implementations

This section describes how the different nodes are implemented to fulfil the requirements of the model previously presented. Node functions are called by providing Node's Ids as a parameter. The first common step to all nodes is to retrieve the node information pointer from the node table. Node descriptions and offsets are based on an 8-channel model assuming that eight simultaneous channels can be processed in a single node.

3.2.1 AudioSource

The audio source node is used to target decoders and retrieves composition buffers from their outputs.

AudioSource nodes have a specific purpose. These nodes establish the link between input buffers and the audio tree. As a direct consequence they must be initialized correctly. A flag (*NewNode*) is provided to warn if it is a first call or not. On a first call, the node has to initialize its pointer to match input buffer locations. The procedure is to go through all its children buffers and update its internal information according to buffer parameters.

According to the buffer sampling rate, the frame length is calculated. The processing is then done on a buffer-per-buffer basis. *CTS* value is checked according to the procedure described above. According to the result, *StartTime* and *StopTime* are processed. Since there is a wide range of output frame configurations (a mix with samples and zeroes), a basic frame is decomposed in 4 sections:

|NbZeroes1 | NbSamples1 | NbZeroes2 | NbSamples2|

This allows a large flexibility over the output frame description. When frames are completely formed, a mechanism check if it is a frame full of zeroes. In this case, the corresponding channel activity flag is set to off. If it is a coherent frame (no zeroes, only samples) it means that no de-synchronization occurred, the output pointers directly points to input buffer location. Conversely when a de-synchronization occurred (*CTS*, *StartTime* or *StopTime*) it is necessary to store the produced frame in a temporary buffer and update output pointers accordingly.

3.2.2 AudioDelay

This node is used to delay a group of sounds, so that they start and stop playing later than specified in the AudioSource nodes.

Proper delay handling requires specific memory buffers. In this implementation, it has been decided to keep delay lines outside the main buffer memory (input buffers and temporary buffers) since it can be implemented in a specific external memory. The procedure is the same as the one used for input buffers: a delay table stores information about delay memory locations, and a system is in charge of reserving some space in this memory upon request.

Request of such delay memory is done during the first call to *AudioDelay*. A *NewNode* flag indicates a first time call, so that the node reserves necessary memory. Reservation is made according to the value of the *Delay* field. The reserved memory has been chosen to be twice as large as the delay value contained in the field to allow larger flexibility. This value has been empirically chosen since there is currently no normative MPEG-4 specification about this issue. If during the composition the user requests a larger delay, nothing has been implemented to de-allocate reserved space and allocate a larger space. During processing, the node checks if its number of output channels (*numChan*) corresponds to the current number of delay lines (*CurrentNbDelayLines*). If it is not the case, a new delay line is requested.

Two arrays of pointers are stored in the node information structure to handle delay lines. *DelayNodeInSamplePtr* locates input locations for incoming samples and *DelayNodeInOutPtr* locates outputs of the delay lines.

As described in [2], delay line updates require a dedicated processing. In this implementation a flag is raised (*Update*) when a new delay value is desired. This new value is stored in *OdelayNodeTarget*. From there, two cases are possible: a longer or a shorter delay. In this case dedicated processing occurs and results are output in a temporary buffer. When there is no update, normal processing is to

increment the node output pointers to match delay line outputs.

a) Target < Delay

In this case, the delay node has to consume more samples than a frame length. The chosen procedure depends on the distance between *Target* and *Delay*. If *Delay-Target* is greater than *FrameLength* then it means that at least two frames need to be consumed during one frame length. To solve this problem a simple but efficient solution has been implemented: decimation. The idea is to pick one sample out of two. This speed up output by a factor two and hence consumes 2**FrameLength* samples and output only *FrameLength* samples. This gives a speed up effect at the output, which is click-free. When *Delay-Target* is smaller than *FrameLength*, it means that there is not enough samples to perform the decimation. In this case a 4-point interpolation was implemented which allows to output a valid frame whatever *Delay - Target + FrameLength* is.

b) Target > Delay

In this case the node has to slow down to diminish its delay. As for the previous case, a simple solution is provided. When *Target-Delay>FrameLength*, a linear interpolation is performed hence consuming *FrameLength*/2 samples. When the *Target* value is about to be reached, a 4-point interpolation is performed to match both values.

These mechanisms are the basic ways to provide click-free delay updates. More sophisticated acceleration/slow-down procedures might enable simulating Doppler-effects and could provide better results.

3.2.3 *AudioMix*

This node is used to mix together several audio signals in a simple, multiplicative way. Any relationship that may be specified in terms of a mixing matrix may be described using this node.

The processing is based on the number of output channels (*numChan*). If there is not enough input channels corresponding to outputs, output activities are set to off. This method avoids producing empty buffers. The execution is simple: each input channel is copied on all output channels weighted by the corresponding matrix coefficient.

A click-free update has been implemented. An update flag warns the node if there is a change in the matrix coefficients. The new coefficients are stored in the node information in the *Target* area. A linear cross-fade is performed to smooth transition between sets of coefficients.

This allows obtaining a click-free coefficient update. The main drawback is the memory occupancy used to store *Target* matrix information.

3.2.4 *AudioSwitch*

The *AudioSwitch* node is used to select a subset of audio channels from the specified child nodes.

WhichChoice is used to specify which subset of input channel is desired at the output. Without any click-free mechanism, the implementation is trivial. The node goes through all its children and output the channel if the corresponding *WhichChoice* value is on. Conversely, the problem becomes tricky when a click-free update has to be implemented. As for the *AudioMix*, the target vector is stored in the node information and an update flag indicates whether or not it shall be used.

The main issue is to associate channels that where *on* in *whichChoice* with channels that are now *on* in *Target*. To perform a correct update, we must list channels *on* in *WhichChoice* and channels *on* in *Target*. While listing, Child Ids and channel indexes are stored in tables. The processing is based on the number of output channels specified by *numChan*. If there are fewer channels *on* in target than specified by *numChan*, extra channels have their activities set to *off*.

In the case two channels need to be exchanged, a cross-fade mechanism is used. The first one is faded-out where as the targeted one is faded-in.

3.2.5 *Sampling Rate Conversion Node*

Sampling rate conversion node is essential to keep the audio tree coherent. It provides frame length coherency between nodes allowing the frame-based compositor to work without any inter-nodes buffering methods. This node is instantiated by the host/pre-processor to cope with different sampling rates. Its judicious location depends on the tree structure as described previously.

The re-sampling method is based on a three-stage asynchronous converter as shown in Figure 5. This node allows converting all the sampling frequencies Table 1.

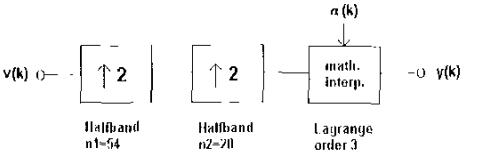

Figure 5 *SRC structure*

3.2.6 *SoundNode*

The *Sound* node is used to attach sound to a scene, thereby giving it spatial qualities and relating it to the visual content of the scene.

Sound node has been implemented without any spatial processing. Spatial audio processing represents indeed a very specific and particular task, to which several years of

activity have been devoted in various research centers, see for instance [4], [6] for more information and details. *Sound* node allows to output sounds from its children. Since the format of sound output is very specific to the hardware used, the described implementation provides at this moment files to output sounds. At the very beginning a procedure goes through all the nodes and creates a file for each *Sound* node channel. When executing the Sound node, a table is used to target the correct file. All fields have been declared and the implementation of a spatialization algorithm is just a matter of implementing the desired processing. The open implementation of nodes permits to ease the coding process.

3.3 Host features

A host class was created to feed the co-processor with coherent information. Features that allow to successfully exploit the co-processor architecture are described here.

3.3.1 Scene initialization

Nodes are described using nodes structures equivalent to those detailed above. An initialization procedure called *InitScene* is done online. The static nature of the audio scene description is somewhat restrictive and would require to be implemented as an open structure allowing to read scene description files. The study was dedicated to the co-processor implementation and behavior, therefore the host processing has not been addressed extensively.

3.3.2 Communication object

The communication object is implemented as a separate class. The same instantiation is shared among the host and the co-processor. It is mainly a FIFO-like structure. Access is done writing a command frame by the host and reading a frame by the co-processor. The number of frames that can be stored in this object is sizeable. The size determines the number of commands that can be sent before the co-processor empty the command buffer (every 10ms).

When the host needs to write a command frame, it creates the object command frame and sends it to the communication object (*comm*). The *comm* object stores a copy of this frame in its internal buffer and updates the number of frames available in it. Upon completion, the host is in charge of killing this frame. On the other side, the co-processor behaves accordingly; it creates a frame and calls the *ReadCommand* method on the *comm* object. This one fills the frame with information contained in its internal buffer and returns the frames back. Once the co-processor has processed this frame, it is charged of deleting it. Everything that goes from the host to the co-processor passes through this object; hence it constitutes a good viewpoint to see the transfer between the host and the co-processor.

3.4 Application example

An example of application has been developed under Windows NT to show some of the implementation model capabilities and features. The MPEG-4 BIFS scene described in Figure 6 has been implemented according to the model. The nodes and their relationships have been described in the HostTreeInit file and the associated input files have been described in the Host Class.

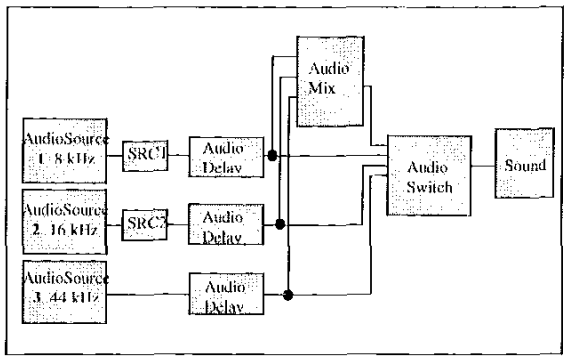

Figure 6 The implemented application example

3.4.1 Interface and Control methods

A main interface object is used to instantiate the Comm, Host and DSP (co-processor) objects. Host and DSP are instantiated with the same communication object. At the beginning host nodes structure are converted into command frames and written in the Communication object. During this initialization, *Initbuffer* commands are sent to initialize three input buffers corresponding to the three sources.

The GUI has been implemented using Visual C++ 5.0 and MFC to obtain the result reported in Figure 7.

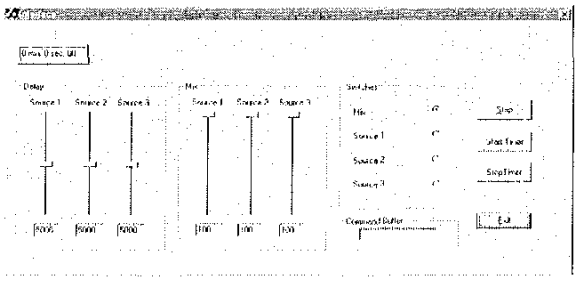

Figure 7 Application interface

Each slider is associated with an object Id. When the user moves a cursor, the new value is sent to the communication object in a *ChangeParameter* frame that contains the *objectId*, the offset and the new value. Along with this command, it is also send the update flag warning the corresponding node that something has changed. The *AudioSwitch* works on the same principle.

Start Timer and *Stop Timer* send the *Start* and *Stop* commands to the co-processor.

The processing part of the application is triggered on a timer event set every 10ms. So, every 10 ms, a co-processor cycle is performed: command interpretation, tree processing and buffer monitoring as described previously. The Step button allows executing a single cycle of the co-processor.

Feeding the co-processor with input samples is performed on a pooling basis. Since the host can count the number of co-processor cycles, it knows when the buffers are going to be empty. In this case, it sends input frames to the communication object.

3.5 DSP Implementation

The way in which the co-processor C code has been designed provides full flexibility and portability. Tests have been performed porting the audio BIFS code to a Motorola DSP56303 platform. The flat memory model (only tables, no complex structures) enables the straightforward porting to assembly code. Offsets to access different information in the table are already defined. Typical instructions to access data are of the form:

In C language :

```
NbNodes = NodeTable[ODSPNodeTableNbNodes];
```

In ASM :

```
Move #NodeTable,r0  ; load node table start
Move #ODSPNodeTableNbNodes,n0 ; set offset
Move x:(r0+n0),x0     ; x0 loaded with NbNodes,
```

which returns the number of nodes in the scene.

Communication between a DSP and a host has been observed to correctly operate on a DMA principle. Space is reserved on the DSP side to be used as a command buffer. *FrameLength* of the command frame is the second word in the frame header. It allows setting up a DMA transfer accordingly, providing a non-intrusive communication mechanism. Upon DMA completion, the number of available commands in the command buffer is updated. Performing this operation at the end avoid contention if a transfer is made while the DSP access its command buffer.

If commands must absolutely be grouped (if they must occur in the same cycle), one can imagine to have a command telling how many linked frames were sent.

4 Conclusion

The goal of this paper is to present an efficient implementation model for audio composition as specified by MPEG-4 audio scenes. A Host/Co-processor implementation model has been described which correctly render audio scenes. A specific memory organization has been fully described leading to an efficient storage of the node information as well as the corresponding buffer structures. The implementation in C of the Co-processor with this specific memory model has maximized the portability to any DSP platform. Communication mechanisms have been implemented to link the Host with the Co-processor thus enabling correct scene manipulation and user interaction. Solutions to solve the sampling rate conversion problems by inserting a new SRC node in the scene description have been presented. Useful features were added to the model such as click-free switching. The open architecture of the software allows new features like spatialization to be easily integrated. A program has been written to show how the user could interact with the BIFS scene using the proposed model.

Finally, experiments have been made using a Motorola DSP to validate the proposed implementation model.

Further work needs to be done in two directions for the full implementation of an MPEG-4 renderer: a complete integration of Structured Audio for downloadable processing algorithms and the extension to advanced spatial capabilities included in the new MPEG-4 version 2.

References

[1] ISO/IEC SC29WG11 Document No. 14496-1 (MPEG-4 Systems) Draft International Standard. MPEG 1998.

[2] ISO/IEC SC29WG11 Document No. 14496-3 (MPEG-4 Audio) Draft International Standard. MPEG 1998.

[3] E. Scheirer, R. Väänänen, J. Huopaniemi: AudioBIFS: the MPEG-4 Standard for Effects Processing. Proceedings of the COST-G6 Workshop on Digital Audio Effects Processing (DAFX '98), Barcelona, November 1998.

[4] G.Zoia, L.LeBourhis, U.Horbach and A.Karamustafaoglu: Proposed revision of Systems and Audio profiles and levels from an analysis of audio composition. MPEG98, Document M3604, Dublin - July 1998.

[5] L.LeBourhis, G.Zoia: About AudioBIFS level definitions. MPEG98, Document M4111, Atlantic City–Oct 98.

[6] Bergault, D. R: 3-D Sound for Virtual Reality and Multimedia. Academic Press, 1994.

[7] Official Draft #3, ISO/IEC SC29WG11 Document No. 14772: The Virtual Reality Modeling Language Specification. July 1996.

Laurent Le Bourhis was born in 1974 near Paris. He received his diploma in Electrical and Electronic Engineering from ESIEE, Paris in 1997 with an emphasis on integrated systems for signal processing. His first working experience was at FER university in Ljubljana (Slovenia) where he developed voice pitch determination on FPGA. He kept on working on sound in Singapore at SGS-Thomson to design a 3D audio algorithm for Dolby AC-3 on a DSP. After three months spent working at IBM on Sonet/SDH VHDL macros, he joined the Integrated System Laboratory (C3i) of the EPFL in 1998. He is now involved in the European project EMPHASIS, with specific tasks in MPEG-4 audio composition and 3-D audio.

Giorgio Zoia received his Diploma in Electrical Engineering from the Politecnico di Milano in April 1996. A month later he joined the Integrated Systems Laboratory of the EPFL (C3i) where he worked on the development of a PCI interface architecture for the PUMA audio chip. In September 1996 he started working on the PUMA chip, a collaboration with Studer Professional Audio AG; his main activities have been digital design and CAD synthesis optimization in submicron technology.
From summer 1997 to the end of 1998 he was involved in the European project EMPHASIS, with specific tasks in MPEG-4 synthetic and 3-D audio. He is actively collaborating to the MPEG-4 standardization process, to which he submitted several contributions concerning Structured Audio, Audio composition (Systems) and analysis of computational complexity. He is currently starting a new project for the implementation of MPEG-4 Audio Systems.

Marco Mattavelli received his Diploma of Electrical Engineering from the Politecnico di Milano, Milano, Italy in March 1987. Then he joined the "Philips Research Laboratories" of Eindhoven. Main research activities regarded channel and source coding for optical recording electronic photography, and signal processing of TV and HDTV signals. Since October 1991 he joined the "Signal Processing Laboratory" (LTS) of the "Swiss Federal Institute of Technology" (EPFL) where he got his PhD degree in 1996. Then he joined the Integrated Circuit Laboratory of EPFL where he is currently Scientific Advisor. Main research interests currently are: architectures and system for video coding, the application and implementation of combinatorial optimization techniques for image analysis, and tools for the aid to architecture design of complex systems. He is also involved in ISO-MPEG standardization activities where currently chairs the MPEG Implementation Study Group.

Daniel J. Mlynek obtained his Ph.D. degree from the University of Strasbourg, France in 1972. He joined ITT Semiconductors in 1973 as a Design Engineer for MOS circuits in the Telecommunication field. He was with ITT Semiconductors until 1989 and held several positions in the R&D, including that of the Technical Director in charge of the IC developments and the associate technologies. The main activities in the Design were in the area of Digital TV Systems where ITT is a World leader an in some of the advanced HDTV concepts. He has several patents on digital TV Systems. Dr. Mlynek was awarded the Eduard Rhein Price for his innovation in signal processing principles that have been implemented in the digital TV system "Digit 2000". In June 1989, Dr. Mlynek joined the Swiss Federal Institute of Technology (EPFL), Lausanne Switzerland, where he is a Professor responsible for the VLSI Integrated Circuits.