INTERNATIONAL STANDARD

ISO/IEC 14496-1

Second edition
2001-10-01

AMENDMENT 1
2001-10-15

Information technology — Coding of audio-visual objects —

Part 1: **Systems**

AMENDMENT 1: Extended BIFS

Technologies de l'information — Codage des objets audiovisuels —

Partie 1: Systèmes

AMENDEMENT 1: BIFS étendus

Reference number
ISO/IEC 14496-1:2001/Amd.1:2001(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.ch
Web www.iso.ch

Printed in Switzerland

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

In the field of information technology, ISO and IEC have established a joint technical committee ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this Amendment may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Amendment 1 to International Standard ISO/IEC 14496-1:2001 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

Information technology — Coding of audio-visual objects —

Part 1:
Systems

AMENDMENT 1: Extended BIFS

1) Add the following at the end of subclause 8.6.15.3.1:

"

class QoS_Qualifier_REBUFFERING_RATIO `extends QoS_Qualifier` : bit(8) tag=0x02 {
    bit(8) REBUFFERING_RATIO;
}

"

2) Add the following at the end of subclause 8.6.15.3.2:

"

REBUFFERING_RATIO – Ratio of the decoding buffer that should be filled in case of prebuffering or rebuffering. The ratio is expressed in percentage, with an integer value between 0 and 100. Values outside that range are reserved.

8.6.15.3.2.1 Rebuffering

In certain scenarios the System Decoder Model cannot be strictly observed. This is the case of e.g. file retrieval scenarios in which the data is pulled from a remote server over a network with unpredictable performances. In such a case prebuffering and/or rebuffering may be required in order to allow for a better quality in the user experience. Note that scenarios involving real time streaming servers do not fall in this category, since a streaming server presumably delivers content according to the appropriate timeline.

An elementary stream is prebuffered when the decoder waits until the decodingBuffer has been filled up to a certain threshold before starting fetching data from it.

An elementary stream is rebuffered when a decoder stops fetching data from the decodingBuffer and before resuming fetching data waits until that buffer has been filled again up to a certain threshold.

In order to inform a receiver whether a certain elementary stream requires prebuffering and/or rebuffering the QoS_Qualifier_REBUFFERING_RATIO qualifier can be included in the Elementary Stream Descriptor (see subclause 8.6.15.3.1). By default, in the absence of such qualifier, an elementary stream does not require pre-buffering or rebuffering.

"

3) Replace Table 31 (Compensation process) in subclause 9.3.4 by the following:

"

Table 31 — Compensation process for multiple fields and BIFS-Anim

<table>
<tr><th>quantType</th><th>animType</th><th colspan="2">Compensation Process</th></tr>
<tr><td>1,2,4,6,7,8, 9 (other than SFVec3fType), 10 (other than SFRotationType), 11,12,13</td><td>1,2,4,6,7,8 11,12,13</td><td colspan="2">The components of v_q^2 are:
vq2[i] = vq1[i] + vDelta[i]</td></tr>
<tr><td rowspan="6">9 (SFVec3fType), 10 (SFRotation)</td><td rowspan="6">9,10</td><td colspan="2">The addition is first performed component by component and stored in a temporary array:
vqTemp[i] = vq1[i] + vDelta[i].
Let scale = $2^{\max(0,nbBits-1)} - 1$.
Let N the number of reduced components (2 for normals, 3 for rotations)
There are then three cases are to be considered:</td></tr>
<tr><td>For every index I,
$|vqTemp[i]| \le scale$</td><td>v_q^2 is defined by,
vq2[i] = vqTemp[i]
orientation2= orientation1
direction2 = direction1 * inverse</td></tr>
<tr><td rowspan="4">There is one and only one index k such that
$|vqTemp[k]| > scale$</td><td>v_q^2 is rescaled as if gliding on the faces of the mapping cube.
Let inv = 1 if vqTemp[k]>=0 and –1 else
Let dOri = k+1
The components of vq2 are computed as follows</td></tr>
<tr><td>$0 \le i < N - dOri$: vq2[i] = inv*vqTemp[(i+dOri) mod N]</td></tr>
<tr><td>$i = N - dOri$: vq2[i] = inv*2*scale–vqTemp[dOri–1]</td></tr>
<tr><td>$N - dOri < i < N$: vq2[i] = inv*vqTemp[(i+dOri-1) mod N]
orientation2 = (orientation1 + dOri) mod (N+1)
direction2 = direction1 * inverse * inv</td></tr>
<tr><td>There are several indices k such that
$|vqTemp[k]| > scale$</td><td>The result is undefined</td></tr>
</table>

Note: The BIFS-Anim process is identical to the process applied for optimal encoding of BIFS multiple fields.

"

4) Replace the reserved bit in subclause 9.3.5.3.1 by the following:

"

bit(1) usePredictiveMFField;

"

5) *Insert the following in subclause 9.3.5.3.2 at the end of the second paragraph (on the use3DmeshCoding):*

"

The usePredictiveMFField flag is used to signal that the syntax for predictive MFField instead of the non-predictive mode is used to encode IndexedFaceSet nodes. This flag is used for terminals supporting this tool.

"

*6) Replace subclause 9.3.7.2.4 (**PROTOcode**) by the following subclauses:*

"

9.3.7.2.4 PROTOcode

9.3.7.2.4.1 Syntax

```
class PROTOcode(isedNodeData protoData) {
    bit(1) isExtern
    if (isExtern) {
        MFUrl locations;
    } else {
            PROTOlist subProtos;
    }
    do {
        SFNode node(SFWorldNodeType,protoData);
        bit(1) moreNodes;
    } while (moreNodes);
    bit(1) hasROUTEs;
    if (hasROUTEs) {
        ROUTEs routes();
    }
}
```

9.3.7.2.4.2 Semantics

First a flag signals whether the prototype is a PROTO, which then has his code included in the proto declaration, or if is an EXTERNPROTO, in which case only an external reference is provided. The EXTERNPROTO is an authoring facility that enables to distribute PROTOs in external libraries and be reused across scenes. The EXTERNPROTO opens a BIFSCommand stream that contains a ReplaceScene command with a BIFSScene with the PROTO definitions. The EXTERNPROTO code is found in the PROTO in this new scene with the same ID in this scene. The nodes that may be contained in this scene are ignored.

In case of a PROTO, the PROTOcode contains a (possibly empty) list of the sub-PROTOs of this PROTO in subProtos, followed by the code to execute the PROTO. The code is specified as a set of SFNodes, using a standard SFNode definition with the additional possibility to declare an IS field. Moreover, the PROTO body may contain ROUTEs if the hasROUTE flag is set to 1.

"

*7) Replace subclause 9.3.7.6 (**Field**) with the following subclauses:*

"

9.3.7.6 Field

9.3.7.6.1 Syntax

```
class Field(FieldData field) {
    if (isSF(field))
        SFField svalue(field);
    else {
        if (BIFSConfig.usePredictiveMFField == 1) {
            bit(1) usePredictive;
            if (usePredictive)
                PredictiveMFField mvalue(field);
            else
                MFField mvalue(field);
        }
        else {
            MFField mvalue(field);
        }
    }
}
```

9.3.7.6.2 Semantics

A field is encoded according to its type: single (SFField) or multiple (MFField). A multiple field is a collection of single fields.

"

*8) Add the following as new subclauses after subclause 9.3.7.9 (**MFVectorDescription**):*

"

9.3.7.10 PredictiveMFField

9.3.7.10.1 Syntax

```
class PredictiveMFField (FieldData field) {
    AnimFieldQP aqp = new AnimFieldQP();
    aqp.useDefault = FALSE;
    field.aqp = aqp;
    ArrayHeader   header(field);
    ArrayOfValues    values(field);
}
```

9.3.7.10.2 Semantic

The array of data is composed of a **Header**, and an **ArrayOfValues**. Note that the FieldData structure is filled as described in the BIFS-Scene quantization process (subclause 9.3.3.1 of ISO/IEC 14496-1:2001).

The process applied for optimal encoding of BIFS multiple fields is exactly identical to the BIFS-Anim process (See Table 31):

1 Compensation on the P values
2 Inverse Quantization into single field values

The compensation process uses the quant type as well as Pmin and PNbBits, defined in the ArrayQP and InitialArrayQP, and can be summarized in the following table.
The inverse quantization process uses the values of floatMax, floatMin, and NbBit as defined in the BIFS quantization process and as defined by the QuantizationParameter node.

9.3.7.11 ArrayHeader

9.3.7.11.1 Syntax

```
class ArrayHeader(FieldData field){
    uint(5)       NbBits;
    int(NbBits)      numberOfFields;
    bit(2)     intraMode;
InitialArrayQP     qp(intraMode,field);
}
```

9.3.7.11.2 Semantic

The array header contains first information to specify the number of fields (**NbBits** is the number of bits used to code the **numberOfFields**). Then the Intra/Predictive policy (**intraMode**) is specified as follows:

- 0 : Only one Intra value at the beginning and then only predictive coded values
- 1 : An Intra every given number of predictive values
- 2 : A bit for each value to determine whether the value is an Intra or predictive value

Lastly, the **InitialArrayQP** is coded.

9.3.7.12 InitialArrayQP

9.3.7.12.1 Syntax

```
class InitialArrayQP(int intraMode, FieldData field){
    switch (intraMode)
        case 1 :
            unsigned int(5)        NbBits;
            unsigned int(NbBits)    intraInterval;
            // no break
        case 0 :
        case 2 :
    int(5) CompNbBits;
    for (int i=0;i<getNbComp(field);i++) {
                        int(field.NbBits+1) vq;
                        field.aqp.Pmin[i] = vq-2^field.NbBits;
}

    }
    // no break
    case 3:
        break;
}
```

9.3.7.12.2 Semantic

If intraMode is 1, the size of the interval between two intras is first specified. Independent of the intraMode, the number of Bits used in Predictive mode CompNbBits and the CompMins are coded. The function getNbComp() is a function that returns the number of components of the quantizing bounds, and depends on the object. For instance it returns 3 for 3D positions, 2 for 2D positions, and 3 for rotations. See Table 17 (Return values of getNbComp) in ISO/IEC 14496-1:2001. CompNbBits and CompMin are stored in the field.aqp AnimationQP structure, and are used for the compensation process as defined in Table 31 (Compensation process for multiple fields and BIFS-Anim) and subclause 9.3.4 of ISO/IEC 14496-1:2001.

9.3.7.13 ArrayQP

9.3.7.13.1 Syntax

```
class ArrayQP(int intraMode, FieldData field){
    switch (intraMode)
        case 1 :
            int     NbBits;
            int(NbBits) intraInterval;
            // no break
        case 0 :
        case 2 :
            boolean(1) hasCompNbBits
            if (hasCompNbBits)
            int(5) CompNbBits;
            boolean(1) hasCompMin
            if (hasCompMin) {
            for (int i=0;i<NbComp(field)) {
                                    int(field.NbBits+1) vq;
                                    field.aqp.Pmin[i] = vq-2^field.NbBits;
}
            }
        case 3:
            break;
}
```

9.3.7.13.2 Semantic

ArrayQP fulfills the same purpose as InitialArrayQP, but in this case, the parameters are optionnaly set. If they are not set in the stream, they are set by default, in reference to the InitialArrayQP or the latest received value of the parameter.
If IntraMode is 1, the size of the interval between two intras is first specified. In any case, the number of Bits used in Predictive mode (**CompNbBits**) and the **CompMin**s are coded. The function getNbComp() is a function that returns the number of components of the quantizing bounds, and depends on the object. For instance it returns 3 for 3D positions, 2 for 2D positions, and 3 for rotations. See Table 17 in ISO/IEC 14496-1:2001. CompNbBits and CompMin are stored in the field.aqp AnimationQP structure, and are used for the compensation process as defined in Table 31 and subclause 9.3.4 of ISO/IEC 14496-1:2001.

9.3.7.14 ArrayOfValues

9.3.7.14.1 Syntax

```
class ArrayOfValues(FieldData field) {
    ArrayIValue value[0];
    for (int i=1; i < numberOfFields;i++)
    {
        Switch (intraMode) {
            case  0:
                ArrayPValue value(field);
                break;
            case 1:
                if ( (i % intraInterval) == 0) {
                    bit(1) hasQP;
                    if (hasQP)
                        ArrayQP qp(field);
ArrayIValue value(field);
                } else {
                    ArrayPvalue value(field);
                }
                break;
            case 2:
                bit (1) isIntra;
                if (isIntra) {
                    bit(1) hasQP;
                    if (hasQP)
                        ArrayQP qp(field);
ArrayIValue value;
                } else {
                    ArrayPvalue value;
                }
                break;
    }
}
```

9.3.7.14.2 Semantic

The array of values first codes a first intra value, and then according to the IntraMode, codes Intra and Predictive values. In P only mode, no more intra values are coded. In the second mode, a bit decides of the P or I mode at each value. In that case, a QP can be sent for Intra values. If a QP is sent, the statistics of the arithmetic encoder are reset. In the third mode, an Intra is sent every intraInterval values.

9.3.7.15 ArrayIValue

9.3.7.15.1 Syntax

```
class ArrayIValue(FieldData field) {
    switch (field.quantType) {
    case  9:  // Normal
        int(1)        direction
    case 10: // Rotation
        int(2)        orientation
        break;
    default:
        break;
}
    for (j=0;j<getNbComp(field);j++)
        int(field.nbBits)   vq[j];
}
```

9.3.7.15.2 Semantic

The ArrayIValue represents the quantized intra value of a field. The value is coded following the quantization process described in the quantization section, and according to the type of the field. For normals the direction and orientation values specified in the quantization process are first coded. For rotations only the orientation value is coded. If the bit representing the direction is 0, the normal's direction is set to 1, if the bit is 1, the normal's direction is set to –1. The value of the orientation is coded as an unsigned integer using 2 bits. The compressed components vq[i] of the field's value are then coded as a sequence of unsigned integers using the number of bits specified in the field data structure. The decoding process in intra mode computes the animation values by applying the inverse quantization process.

9.3.7.16 ArrayPValue

9.3.7.16.1 Syntax

```
class ArrayPValue(FieldData field) {
    switch (field.quantType) {
    case  9:  // Normal
        int(1)     inverse
        break;
    default:
        break;
}
    for (j=0;j<getNbComp(field);j++)
        int(aacNbBits)     vqDelta[j];
}
```

9.3.7.16.2 Semantic

The ArrayPValue represents the difference between the previously received quantized value and the current quantized value of a field. The value is coded using the compensation process as described above. The values are decoded from the adaptive arithmetic coder bitstream with the procedure ν_aac = aa_decode(model). The model is updated with the procedure model_update(model, ν_aac). For normals the inverse value is decoded through the adaptive arithmetic coder with a uniform, non-updated model. The compensation values νqDelta[i] are then decoded one by one. Let νq(t-1) the quantized value decoded at the previous frame and ν_aac(t) the value decoded by the frame's Adaptive Arithmetic Decoder at instant t with the field's models. The value a time t is obtained from the previous value as follows :

νDelta(t) = v_acc(t) + Pmin

νq(t) = AddDelta(νq(t-1), vDelta(t))

ν(t) = InvQuant(νq(t))

The field's models are updated each time a value is decoded through the adaptive arithmetic coder. If the animType is 1 (Position3D) or 2 (Position2D), each component of the field's value is using its own model and offset PMin[i]. In all other cases the same model and offset PMin[0] is used for all the components.

"

*9) Modify the subclause numbers in 9.3.7.10 to 9.3.8.10 after the insertion of **ArrayPValue** subclause.*

*10) Insert the following as a new subclause after subclause 9.4.2.92 (**Script**):*

"

9.4.2.93 ServerCommand

The ServerCommand in BIFS enables the application signaling in MPEG-4 Systems. The application-signaling framework allows an application to communicate the application signaling messages or commands to a server(s). Commands are sent to servers upon the occurrence of events (synchronous events specified in the scene description or asynchronous events as a result of user interaction). The ServerCommand framework consists of two elements; a ServerCommand node, and a

ServerCommandRequest structure. While the ServerCommand enables event routing to the server, the ServerCommandRequest structure specifies the syntax for the messages communicated to the server over a back channel.

9.4.2.93.1 Node Interface

ServerCommand {

eventIn	SFBool	**trigger**	
exposedField	SFBool	**enable**	FALSE
exposedField	MFString	**url**	[]
exposedField	SFString	**command**	""

}

9.4.2.93.2 Functionality and Semantics

This node is used to communicate application-signaling messages (commands) from the client back to the server. The ServerCommand is processed only when **trigger** receives a TRUE event and **enable** is TRUE. When the ServerCommand is processed, the **command** is sent to the servers indicated by the specified **url**. A **url** identifies the object descriptor that contains an elementary stream that flows from the terminal back to the server. If that object descriptor has more than one such elementary stream, then the one specified will be used. The **command** field contains the information that is transmitted back to the server. The syntax and semantics of the **command** string are application specific and not specified. The syntax of the **ServerCommandRequest** structures used to communicate the **command** to a server is specified below.

9.4.2.93.3 ServerCommandRequest

When the ServerCommand is processed the associated **command** is communicated to the servers specified in the **url** using the **ServerCommandRequest** structures. The ServerCommandRequest is encapsulated into SL packets, using the SLConfigDescriptor contained in the ESDescriptor of the upchannel elementary stream that carries the commands. If a timestamp is provided in the SL layer (either decoding or composition) then it is directly derived from the System Time Base of the terminal.

Syntax

```
class ServerCommandRequest(BIFSConfig cfg) {
        bit(cfg.nodeIDbits) nodeID;
        SFString command;
    }
```

where **nodeID** is node ID of the ServerCommand node that trigger the command (all such nodes must have IDs in order to route events into them), and **command** is the string contained in the ServerCommand node's **command** field.

"

*11) Modify the subclause numbers in 9.4.2.93 to 9.4.2.99 after the insertion of **ServerCommand** node subclause.*

*12) Insert the following as a new subclause after subclause 9.4.2.99 **(Switch**):*

"

9.4.2.100 TemporalGroup

The **TemporalGroup** node carries the temporal constraints of its child nodes that will be used by the FlexTime model (or Advanced Synchronization Model). The FlexTime Model supports synchronization of objects from multiple sources with possibly different time bases. The FlexTime Model specifies timing using a flexible, constraint-based timing model. In this model, media objects can be linked to one another in a time graph using relationship constraints such as "CoStart", "CoEnd", or "Meet". And, in addition, to allow some flexibility to meet these constraints, each object may have a flexible duration with specific stretch and shrink mode preferences that may be applied.

The FlexTime model is based upon a so-called "spring" metaphor. A spring has a set of three constants: the minimum length below which it will not shrink, the maximum length beyond which it will break, and the optimal length at which it rests comfortably being neither compressed nor extended. Following this spring model, the temporal playback of media objects can be viewed as springs, with a set of playback durations corresponding to these three spring constants. The optimal playback duration (optimal spring length) can be viewed as the author's preferred choice of playback duration for the media object. A player should, where possible, keep the playback length as close to the optimal duration as the presentation allows but may choose any duration between the minimum and maximum durations as specified by the author. Note, that whereas stretching or shrinking the duration continuous media, e.g. for video, implies respectively slowing down or speeding up playback, for discrete media such as a still image, shrinking or stretching is merely adjusting the rendering period to be shorter or longer.

The FlexTime model requires a small change to the MPEG-4 buffer model in terms of media delivery and decoding. Decoding may be delayed on the client, beyond the standard decoding time, by an amount determined by the flexibility expressed in the relationships. The buffer model for FlexTime can thus be specified as follows: "At any time from the instant of time corresponding to its DTS up to a time limit specified by FlexTime, and AU is instantaneously decoded and removed from the decoding buffer."

To support synchronization of nodes within the scene to a media stream, or part thereof, a new node supporting flexible transformation to scene time is introduced. This grouping node is the **TemporalTransform** and can flexibly support the slowing down, speeding up, freezing or shifting of the scene time for rendering of nodes contained within. This transform node is also a grouping node and provides the flexible component for the FlexTime model.

The **TemporalGroup** provides the constraint for the FlexTime model and gives it the tools it needs to align in time both nodes and media streams with nodes in the scene graph. **TemporalGroup** can examine the temporal properties of its children, check for the availability of media in the composition buffer, and consequently decide which temporal transformation parameters to apply to each of its child nodes.

9.4.2.100.1 Node Interface

TemporalGroup {

eventIn	MFNode	**addChildren**	
eventIn	MFNode	**removeChildren**	
exposedField	MFNode	**children**	
field	SFBool	**costart**	
field	SFBool	**coend**	
field	SFBool	**meet**	
exposedField	MFFloat	**priority**	[]
eventOut	SFBool	**isActive**	
eventOut	SFInt32	**activeChild**	

}

9.4.2.100.2 Functionality and semantics

The **TemporalGroup** node specifies the temporal relationship between a given number of **TemporalTransform** nodes.

The **children** field specifies the list of **TemporalTransform** or **TemporalGroup** nodes on which the constraint is applied.

The **costart**, **coend** and **meet** fields specify the temporal relationships amongst the node's **children**. If **costart** is TRUE, all child nodes must be activated (start) together. If **coend** is TRUE, all child nodes must be deactivated (end) together. When **meet** is TRUE, the child nodes are activated one after another in a row. When one node ends, the next node in the list needs to start. If either **costart** or **coend** are set to TRUE the **meet** field is ignored.

The **priority** field specifies the list of priority numbers that determines the preferred scaling direction when two child nodes need to meet a constraint. The list of priorities is in the same order as the **children** field. More than one child can have the same value. In the case of **coend** the highest priority object will determine the end and cause all other objects to end at that time, providing all objects have at least reached their minimum durations. If the field is empty all nodes are assumed to have equal priority.

The **isActive** eventOut is triggered at the following events.

- If **costart** is true a TRUE value will be sent when the co-start constraint is met.
- If **coend** is true a FALSE value will be sent when the co-end constraint is met.
- If **meet** is true a TRUE value will be sent when the first child is activated, and a FALSE value will be sent when the last one finishes.

The **activeChild** eventOut is sent when a new child is activated under a **meet** constant and will indicate the index of that child. The first child is index 0.

9.4.2.101 TemporalTransform

TemporalTransform is a grouping node that assigns temporal properties, and applies temporal transformation, to scene nodes and elementary streams.

9.4.2.101.1 Node Interface

TemporalTransform {

eventIn	MFNode	**addChildren**	
eventIn	MFNode	**removeChildren**	
exposedField	MFNode	**children**	
exposedField	MFString	**url**	[]
exposedField	SFTime	**startTime**	-1.0
exposedField	SFTime	**optimalDuration**	-1.0
exposedField	SFBool	**active**	FALSE
exposedField	SFFloat	**speed**	1.0
exposedField	SFVec2F	**scalability**	[1.0, 1.0]
exposedField	MFInt32	**stretchMode**	[0]
exposedField	MFInt32	**shrinkMode**	[0]
exposedField	SFTime	**maxDelay**	0
eventOut	SFTime	**actualDuration**	

}

9.4.2.101.2 Functionality and semantics

The **TemporalTransform** node serves two purposes:

- To apply temporal transformation to media objects.
- To hold properties that will be used when the node has as a parent a **TemporalGroup** node.

The node operates on two types of objects. Its **children** field may contain a list of nodes of the type SF3DNode. In addition, it has a **url** field that may reference an elementary stream. In the first case, the node has the effect of slowing down, speeding up, freezing or shifting the time base of the compositor when it renders the child nodes that are transformed by the node. In the second case, the node affects the time base of the stream. Note that a Route between two nodes whose time bases are different, because one or both are affected by a **TemporalTransform**, will have undefined behavior.

The **startTime** specifies the starting point of the media stream relative to the composition time of the first access unit received from the **url** that is controlled by this node. If **startTime** is negative, the entire media referred by this **url** is controlled.

The **optimalDuration** field specifies the nominal duration of the objects that are controlled by this node. This is also the *optimal* duration, which the FlexTime model opts for when scaling this node. If **optimalDuration** is negative, or outside the bounds defined by the **scalability** field, optimal duration is not available.

The **active** field determines whether the node, its children, and the stream controlled by the node are active. When the node is inactive, the time base of the compositor is frozen when the child nodes are composed. This means that:

- The nodes are not visible and the stream is not played.
- Timed nodes, e.g. **TimeSensor**, do not have their time running.
- Node fields such as **startTime** and **stopTime** are processed as if the time is not running.
- Nodes that react to user interaction, such as **TouchSensor**, or to their spatial position, such as **ProximitySensors** cannot be activated.

However operations that would normally be performed at that time are still performed, even if the node is frozen. For instance:

- **Script** nodes are executed if activated.
- **ROUTE**s are executed.
- eventIns are processed (with no rendering).
- DMIF (network stack) methods are called if necessary. Therefore the delivery of streams, if required, will be requested, even though their sync layer time base is frozen.

Another field that affects the temporal transformation is **speed**. When the value of this field is not 1 and the node is active, the scene time base of the node, its children, and the time base of the stream will slow down or speed up according to this factor. If **speed** is set to zero, the node remains active but its time stops. Therefore time-related operations behave as if time is constant, and audio rendering pauses.

The other fields of the **TemporalTransform** node have no effect on the execution of the node, but are used by a parent **TemporalGroup** node to determine the temporal layout of the node in relation to other **TemporalTransform** objects.

The **scalability** field specifies the maximum ratios by which this object is allowed to shrink or stretch. If a nominal duration is known, either from **optimalDuration** or the stream length, the ratio determines the absolute values of the minimum and maximum duration. Otherwise, for unknown duration, the field dictates either the ratio by which the time bases controlled by the node are allowed to scale; or, when **optimalDuration** lies outside the bound of the values calculated, they are minimum and maximum durations (optimal duration unknown).

The **stretchMode** field specifies an ordered list of the preferred modes of stretching according to the table below.

The **shrinkMode** field specifies an ordered list of the preferred modes of shrinking according to the table below.

Table 39 — Preferred Mode of Stretch/Shrink Values

StretchMode Value	StretchMode Description
0	Hold rendering of the last Access Unit
1	Linear Composition Unit rendering rate decrease
2	Repeat

ShrinkMode Value	ShrinkMode Description
0	Stop rendering
1	Linear Composition Unit rendering rate increase

The **maxDelay** field specifies how long the FlexTime model can wait for the stream specified by the **url** field. If this time elapses before the stream is available, the model behaves as if the node starts at that time, but the player will not render the children of this node and will discard the stream if it arrives later.

The **actualDuration** eventOut is triggered when the node is activated and sends the value of the estimated actual play duration of the node.
"

*13) Modify the table numbers 39 to 60 after the insertion of **TemporalTransform** node subclause.*

*14) Modify the subclause numbers 9.4.2.100 to 9.4.2.111 after the insertion of **TemporalTransform** node subclause.*

15) Insert the following item as the last bullet item in subclause 15.3.2 (Scene Graph Profiles Tools):

"

- 3D audio scene graph profile as defined in 15.3.3.3.

"

16) Add the following subclause as a new subclause at the end of subclause 15.3.3.2:

"

15.3.3.3 3D Audio Scene Graph Profile

The 3D Audio Scene Graph profile provides tools for three-dimensional sound positioning in relation either with acoustic parameters of the scene or its perceptual attributes. The user can interact with the scene by changing the position of the sound source, changing the room effect or by moving the listening point.

The following list defines the 3D Audio Profile scene graph profiles:
AcousticScene, Anchor, AudioBuffer, AudioClip, AudioDelay, AudioFX, AudioMix, AudioSource, AudioSwitch, Billboard, Conditional, DirectiveSound, Group, Inline, ListeningPoint, LOD, NavigationInfo, OrderedGroup, PerceptualParameters, QuantizationParameter, Sound, Sound2D, Switch, Transform, Viewpoint, WorldInfo, Node Update, Route Update, Scene Update, AnimationStream, Script, CoordinateInterpolator, OrientationInterpolator, PositionInterpolator, PositionInterpolator2D, ProximitySensor, ROUTE, TermCap, TimeSensor, TouchSensor, VisibilitySensor, Valuator.

"

17) Replace subclause 15.3.4.1.2 (Levels) by the following subclause:

"

15.3.4.1.2 3D Audio Profile Level Definitions

In the following table, levels definitions for the 3D Audio profile are given. The levels are based on sampling rate of 44100 Hz at 16-bit resolution. Their complexities depend on

- The maximum number of spatialized sources per scene (these spatialized sources can include discrete reflections that are perceptually equivalent to individual sound sources)
- The number of temporal sections whose levels and time limits can be controlled individually for each source
- The maximum number of independent late reverberation processes per scene
- The maximum number of control frequencies in reverberation process filters, source directivity filters, and material filters.