TECHNICAL REPORT

ISO/IEC TR 21000-12

First edition
2005-04-01

Information technology — Multimedia framework (MPEG-21) —

Part 12: Test Bed for MPEG-21 Resource Delivery

Technologies de l'information — Cadre multimédia (MPEG-21) —
Partie 12: Lit d'essai pour livraison de ressources MPEG-21

Reference number
ISO/IEC TR 21000-12:2005(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.org
Web www.iso.org

Published in Switzerland

Contents

Page

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. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. 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.

In exceptional circumstances, the joint technical committee may propose the publication of a Technical Report of one of the following types:

— type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;

— type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;

— type 3, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

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

ISO/IEC TR 21000-12, which is a Technical Report of type [3], was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

ISO/IEC TR 21000 consists of the following parts, under the general title *Information technology — Multimedia framework (MPEG-21)*:

— *Part 1: Vision, Technologies and Strategy* [Technical Report]

— *Part 2: Digital Item Declaration*

— *Part 3: Digital Item Identification*

— *Part 5: Rights Expression Language*

— *Part 6: Rights Data Dictionary*

— *Part 7: Digital Item Adaptation*

— *Part 8: Reference Software*

— *Part 9: File Format*

— *Part 10: Digital Item Processing*

— *Part 11: Evaluation Tools for Persistent Association Technologies* [Technical Report]

— *Part 12: Test Bed for MPEG-21 Resource Delivery* [Technical Report]

— *Part 16: Binary Format*

The following parts are under preparation:

— *Part 4: Intellectual Property Management and Protection Components*

— *Part 15: Event Reporting*

Introduction

This document describes the component API of ISO/IEC TR 21000-12: Test Bed for MPEG-21 Resource Delivery. The test bed is mainly composed of a streaming player, a media server, and an IP network emulator. This document describes the API of each components of the test bed to facilitate a component oriented development process. This platform provides a flexible and fair test environment for evaluating scalable media streaming technologies for MPEG contents over IP networks.

Information technology — Multimedia framework (MPEG-21) —

Part 12:
Test Bed for MPEG-21 Resource Delivery

1 Scope

This Technical Report specifies a test bed that is designed to assist in performance assessment of MPEG-21, Scalable Video Coding (SVC) for streaming applications and for the evaluation of resource delivery technologies over unreliable packet-switched networks. A subset of MPEG-4 IPMP is also included in the test bed so that encrypted streaming and layered access functionality of a DRM system can be tested for different SVC designs.

2 Overview of Functionality

2.1 Functionality of Test Bed

This platform provides a flexible and fair test environment for evaluating scalable media streaming technologies for MPEG contents over IP networks. In particular, the test bed is designed for the evaluation of Scalable Video Coding (SVC). This test bed has capabilities of simulating different channel characteristics of various networks, therefore,

- Various scalable codec (audio, video, scene composition) technologies could be evaluated.
- Various packetization methods and file formats can be evaluated.
- Various multimedia streaming rate control and error control mechanisms can be plugged into the test bed and evaluated.

2.2 Target Use Cases

Currently, the test bed is targeted at scalable audio and video streaming applications with some DRM support.

2.3 Relation with Reference Software

It must be emphasized that the software provided with this TR is not part of the MPEG-21 reference software. In addition to providing some useful utility software for resource delivery system development, this TR tries to show a solid example of how MPEG technologies can be integrated together in a working system for scalable audio/video streaming applications.

2.4 MPEG Technologies within Test Bed

The following MPEG technologies are supported by the test bed:

- MPEG scalable audio and video codecs
- MPEG-4 on IP
- MPEG-4 IPMP (a small subset)
- MPEG-21 DIA Network Adaptation QoS

The architecture and the API are not tied to any particular media codecs. However, only the MPEG-4 FGS video and MPEG-4 BSAC audio are officially supported by the software.

2.5 API Overview and Language

The API is divided into three parts, namely, server component API, client component API, and common component API. The network emulator descried in clause 7 is a standalone application that is not part of the component-based framework. The programming language used to describe the API is C++. This is because the project is mainly implemented using the C++ language, except for the network emulator GUI, which is done in Java. This will make it easier for the implementers to follow the API and to design different modules to merge into the test platform.

3 Overall Architecture

The overall architecture of the Test Bed is illustrated in Figure 1.

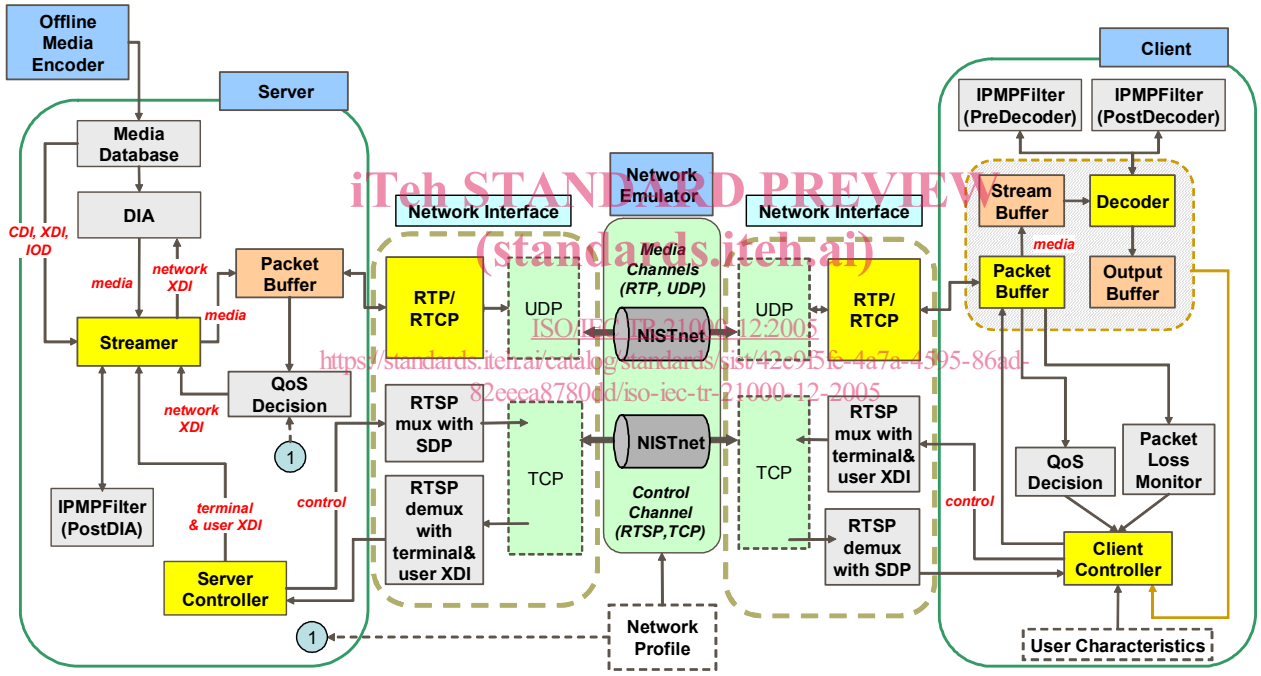

Figure 1 — Architecture of the resource delivery test bed

Subsequently, the various server and client components that can fit in this test-bed architecture are described in clause 4.

4 Client Components

4.1 Introduction

This section lists the detail component APIs so that third party component providers can design their own client modules that can be integrated into this test bed.

4.2 Decoder Object

This is the base object for all decoders, including real-time and offline decoders. The task of fetching coded units from the input streams and storing decoded units into the output stream is carried out by this base object. This is done as follows:

1. **Decoder** gets some coded bits from the input bitstream by calling **StreamBuffer::GetBitstreamData()**. A bitstream can contain multiple layers of sub-bitstreams. If no data in any layers of the sub-bitstream is available, the **Decoder**'s thread is suspended till data is available.
2. **Decoder** calls a private virtual function, *Decode()*. This function is overloaded by specific decoders and performs the actual decoding of a coding unit.
3. The output of *Decode()* is saved in a buffer via the method **OutputBuffer::WriteDecodedData()**. The GUI can later calls **OutputBuffer::GetDecodedData()** to display (video) or play (audio) the decoded content. A presentation timestamp will be provided in this later API.

Methods
The following methods are implemented by the base class **Decoder**.

Decoder()

The object constructor (serious initialization is done in *Setup*()). No work is done till *Start()* is called.

int Setup(int track_id, uint8 *config, int32 config_length, int32 codec_type, void* streambuffer_link, void* outputbuffer_link)

This function initializes the decoder. The video header (or video configuration descriptor is passed in for parameters like video width & height or audio frame size). *config_length* is the length in bits of *config*, and *streambuffer_link* and *outputbuffer_link* are the pointers to StreamBuffer object and OutputBuffer object respectively. A non-zero error code returns if it fails.

void Start()

Start the **Decoder** thread, *Run()*, to perform the decoder task.

void Close()

Must be called before the **Decoder** object is deleted in order to terminate the **Decoder** thread.

static void Run(void *)

This method, which runs the **Decoder** thread, fetches compressed units from the input **StreamBuffer** object, calls *Decode()* to decode them, and saves the decoded presentation units to the output **OutputBuffer** object through the **OutputBuffer::WriteDecodedData()** method. The function terminates when the input stream returns END_OF_STREAM on a call to *GetBitstreamData()*.

bool Decode()

Decode one coding unit. The function returns TRUE if successful.

uint32 GetAttributes(ATTRIBUTE_CODE code, void *value)

Depending on the value of *code*, the requested attribute is returned through the pointer parameter *value*. In some codecs, this function returns meaningful values after the *Setup()* function (i.e. after the decoding of the audio-visual configuration header). A nonzero error code returns if it fails. For some codecs (e.g. ITU-T H.263), this function only returns meaningful values after decoding of a frame (after calling the function *Decode()*).

When the attribute "SUPPORTED_CODEC_NAMES" is requested, the function returns a pointer of an array of UTF-8 strings of codec names. Each string in the array begins with an one byte length field, followed by the chacters. Strings are concatenated one after another, the whole string arrays ends with a zero byte. For example, a **Decoder** object may returns a pointer points to the following string array:

{ 0x03, 'A', 'S', 'P', 0x03, 'F', 'G', 'S', 0x00 }.

The default implementation of this function always returns zero.

ATTRIBUTE_CODE is defined as follows:

```
typedef enum
{
        // common attributes
        SUPPORTED_CODEC_NAMES

        // for visual
        FRAME_WIDTH,     // the width in pixels of a video frame
        FRAME_HEIGHT,    // the height in pixels of a video frame
        FRAME_BITS_PER_PIXEL,  // bits per pixel in a video frame
        FRAME_PIXEL_FORMAT, // pixel format in a video frame

        // for audio
        PITCH,        // the pitch of an audio sample
        AUDIO_FREQUENCY,     // the frequency of an audio sample
        NUM_CHANNELS,    // number of audio channels
        BITS_PER_SAMPLE,     // number of bits per audio sample
        AUDIO_CU_DURATION,  // the duration in milliseconds of a sample
} ATTRIBUTE_CODE;
```


As you can see from this definition, codes for different types of decoders are mixed. A decoder is expected to return value only for attribute codes that are relevant to it. Otherwise it returns zero. The returned attribute is always type-casted to uint32. The method returns zero if the requested attribute is not recognized or available by the decoder.

4.3 OutputBuffer Object

This object class is used as an abstraction of the output device (for either audio and video). The decoder, after decoding, should dump the decoded data into this object. The client GUI would then send the data to the actual device based on the timestamp.

Methods

OutputBuffer()

This is the object constructor.

int Setup(int32 buffer_size)

buffer_size is the size of one decoded data unit (e.g. an YCbCr video frame or a PCM audio frame) in bytes.

int WriteDecodedData(uint32 timestamp, uint8 *data, int32 data_length)

This method sends a decoded unit to the output buffer. The pointer **data* points to the decoded frame and the size of the decoded data is specified by *data_length*. The timestamp, *timestamp*, of the decoded frame is in milliseconds. The function returns zero upon success, otherwise, it returns a non-zero code when it fails.

int GetDecodedData(uint32 *timestamp, uint8 **data, int32 *data_length)

The client GUI will call this method to render the decoded content (audio or video). The indirect pointer ***data* returns a pointer to the decoded data and the size of the decoded data is returned via *data_length*. The presentation time is returned via the parameter **timestamp*. The client GUI should render the data at time *timestamp* (in millisecond). The function returns zero upon success, otherwise, it returns a non-zero code when it fails.

int ReleaseBuffer(void)

After client player gets decoded data, it will call this method to release the control to the buffer which is obtained by using **GetDecodedData()**.

4.4 StreamBuffer Object

StreamBuffer is a FIFO buffer that holds raw bitstream data. Even though the data structure is intended for raw data bits, the actual implementation should record raw bitstream boundary "markers" set by the transport layers, e.g. IP packet boundaries (see method **GetBitstreamData()** for more explanations), through some auxiliary data structure.

Methods

StreamBuffer(int number_of_tracks)
This is the object constructor. *number_of_tracks* is the total number of media tracks (e.g. audio-visual tracks or multiple levels/layers for scalable codecs).

int Setup(int track_id, int32 buffer_size, void *packetbuffer_link)
buffer_size is the size of the raw bitstream buffer in bytes. *Packetbuffer_link* is the link to **PacketBuffer** object.

int SetBookmark(int track_id, int32 bookmark_number)
This method records the current position in stream buffer to a bookmark array for track *track_id*, and the corresponding index in this array is *bookmark_number*. This method is always called by **Decoder** object to do backward searching preparation. It retruns a non-zero code when it fails.

int GotoBookmark(int track_id, int32 bookmark_number)
This method will move the current position to that recorded in the bookmark array by index *bookmark_number* for track *track_id*. It returns a non-zero code when it fails.

int32 GetBitstreamData(int track_id, int32 nbytes, uint8 *data)
This method retrieves *nbytes* bytes from the track *track_id* of a **StreamBuffer** object, and returns the value to the caller. The returned data is stored in the pointer *data*. The function returns zero upon success, otherwise, returns a non-zero code when it fails.

int32 GetOffsetToNextDataBoundary(int track_id)
In a streaming-over-IP system, bitstream data has a natural boundary set by the transport layer (e.g. for each frame, slice or video packet, etc.). This function returns the number of bits to the next marker (or packet boundary) position in track *track_id*. Some decoders can use this information to avoid resync marker searching. It returns zero if there is no further boundary point.