MOBIO

Mobile Biometry

http://www.mobioproject.org

Funded under the 7th FP (Seventh Framework Programme)
Theme ICT-2007.1.4
[Secure, dependable and trusted Infrastructure]

D6.2: System architecture and draft API definition for integration of biometric modules

Due date: 31/10/2008 **Submission date**: 19/12/2008
Project start date: 01/01/2008 **Duration**: 36 months
WP Manager: Giorgio Zoia **Revision**: 1

Author(s): Giorgio Zoia (EPM)

Project funded by the European Commission in the 7th Framework Programme (2008-2010)		
Dissemination Level		
PU	Public	No
RE	Restricted to a group specified by the consortium (includes Commission Services)	Yes
CO	Confidential, only for members of the consortium (includes Commission Services)	No

D6.2: System architecture and draft API definition for integration of biometric modules

Abstract:

This deliverable introduces the main concepts of the eyeP Communicator system architecture. eyeP Communicator is the application where the MOBIO bi-modal authentication systems will be integrated in order to validate the MOBIO approach to mobile security.

Communicator is a modular system implementing a full communication suite on both desktop and portable mobile devices. Its functionality ranges from simple VoIP calls to messaging in a fully open-standard and interoperable environment. Support of the main IP communication and security protocols makes it the ideal candidate for the integration task of MOBIO.

Furthermore, the deliverable will introduce in more detail the layer of the media stack running inside Communicator, and the software interface that needs to be implemented in order to allow the integration of additional processing modules. A few source code files are provided in Annex for a more exhaustive reference.

Table of Contents

1.Introduction

The goal of MOBIO is to study, develop and evaluate bi-modal biometric authentication technologies in the context of portable devices and related network infrastructures. More particularly, MOBIO focuses on two natural and unintrusive biometric modalities, face and voice. The face is considered because it is an accepted biometric already being extensively used in passports and identity-related documents across the world; it is also biometric information that can easily be obtained as most mobile devices now have at least one camera. The voice is considered as it is naturally and readily available from mobile devices such as mobile phones.

The proof of concept of the MOBIO approach requires the integration of bi-modal biometric tools inside a communication application offering all the necessary feature set for the implementation of one or more use case scenarios presented in previous deliverable D2.1. Feature requirements include audio-visual capture and rendering, audio-visual streaming with the possibility to exploit secure protocols and services, and of course the possibility to integrate new developed functionality for both desktop and mobile platforms. The communicator from eyeP Media constitutes a good base for this task.

eyeP Communicator is the most recent generation of SIP software phone applications from eyeP Media. It is available for Windows OS and porting to MacOS is planned in the medium term. Featuring a contact-centric interface, eyeP Communicator is an intuitive communication toolbox that allows to make calls or conferences by simply selecting contacts from a local address book, including support for some of the major mailers. eyeP Communicator has been designed to offer modular packages, with special focus on telecommunication and media robustness: it has been verified on major SIP telecommunication equipments (including IMS platforms) and for interoperability with other programs. It further integrates state-of-the-art voice and video processing and a wide choice of audio and video codecs.

A simpler but still powerful version of eyeP Communicator exists for mobile platforms. eyeP Communicator for Windows Mobile is a seamlessly integrated software phone for Pocket-PC or Smart-phone devices. Reusing the native address book, dial pad and call history interfaces of the cellular phone, it allows the end-user to make VoIP calls whenever a Wi-Fi connection is available without affecting the simultaneous cellular service. Except for the UI and user tools, it shares with the desktop version the same architecture, which will be introduced in this deliverable.

Understanding the architecture of Communicator is necessary in order to have an overall vision on the application feature set and possibility of extension. At the same time it is even more important to analyze where and how MOBIO baseline modules can be integrated, and the EPM media stack appears the best place for reasons that will be considered in the deliverable.

Following this analysis, in the second part of this document a detailed presentation of the media stack and its real-time execution engine is provided, together with the main guidelines of how new processing modules can be integrated. It is finally highlighted what is supposed to be the task of third party developers and what instead will be the EPM role in the overall integration process. These draft guidelines propose a programming interface to be followed during the development, in order to make the subsequent process straightforward; a couple of iterations are expected in order to make this programming interface stable in the months following the publication of this first draft.

Some suggestions are also given in order to anticipate and reduce possible problems due to the intense use of processing resources, especially for video processing.

In annex to the document, some source code files (C++ header files for a few base classes) are included for a more complete reference overview of the current interface. Purpose of this document is also to stimulate further identification of missing extensions, if necessary, before finalizing the programming interface and start the code integration. Some of these missing extensions (like for buffers) have been already identified and added to the specification.

2.CDE soft-phone system architecture

This section provides an overview of the eyeP Media soft-phone system architecture. The soft-phone application is the best candidate to the integration of the MOBIO bi-modal authentication modules into a communication application, as Communicator implements a full telephony-messaging-presence functionality by support of the SIP protocol (Session Initiation Protocol), which is also underlying the new IMS (IP Multimedia Subsystem) architecture recently standardized by 3gpp and IETF[1]. The next sub-sections will present the Communicator architecture and some basic elements, finally identifying the best way to integrate MOBIO modules.

2.1. EyeP Communicator Architecture

The eyeP Communicator is a multi-process, multimedia client implementing voice and video over IP (V^2OIP) functionality together with other services typical of communication suites like presence monitor, instant messaging, SMS-MMS capabilities, etc. in a possibly secured scenario (thanks to the support of S-RTP and more recently of IMS, see below).

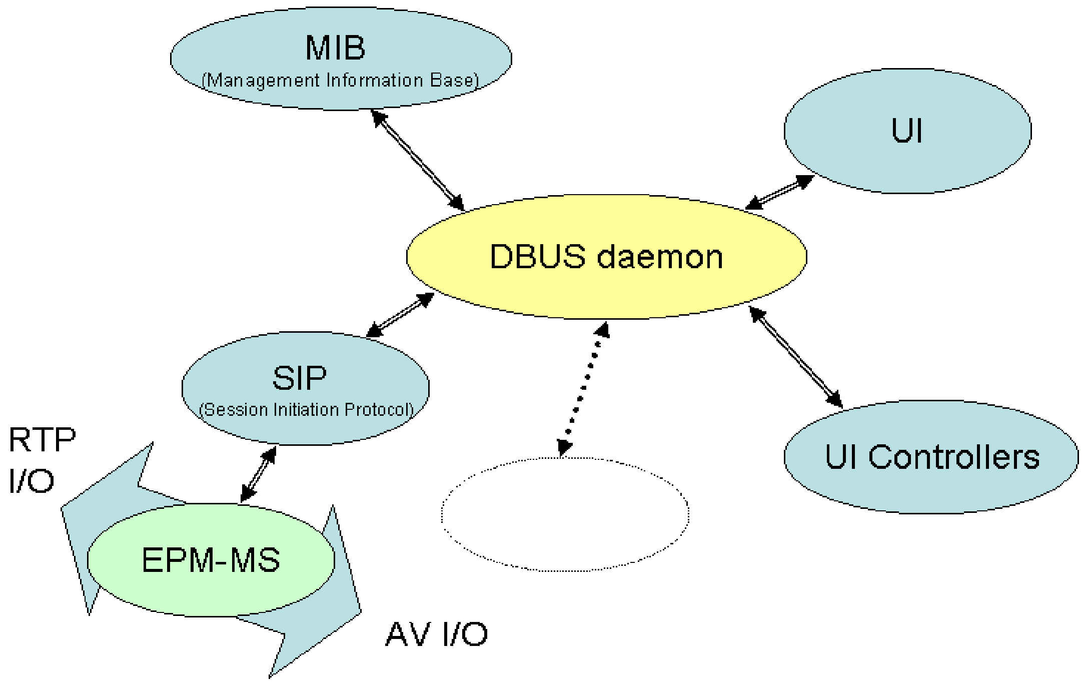

Figure 2.1. eyeP Communicator main architectural elements.

The multi-process and/or multi-thread modular architecture is actually based on d-bus as inter-process communication system (see again next subsection 2.2 below). All the other module (Management Information Base or MIB, UI, UI controllers, SIP and media, etc.) can communicate among each others either by changing values of shared variables in the information base so that

1“TS 23 228: IP Multimedia Subsystem (IMS); Stage 2 (Release 7),” Tech. Rep., March 2007 other modules are notified through the d-bus, or by directly calling a “proxy” of another module's function, again through d-bus interfaces. Figure 2.1 above graphically shows the eCom architecture. The application component that implements the telephony functionality is called eCom_sip; this component is in charge of the sip layer negotiation and management, and furthermore it initiates and drive the media stack whenever an active call is started.

For the mobile device version, native device UI and databases are exploited (dialer, call logs, contacts, etc.) allowing to integrate the default cellular functions with VoIP functionality in the same application framework.

The main advantage in such a kind of architectural approach is the fully independent execution that each component can have, so that its whole life cycle (implementation, debugging, integration, maintenance) is completely independent from all the other modules once the d-bus communication interfaces are specified; in fact, the emulation of the rest of the application can be simply obtained via a number of d-bus messages that replicate the expected behavior from other modules.

The use of d-bus also rationalize the system architecture as it maintains real separation between modules, and then smooths execution and limit the risk of side effects between different functionalities.

On the other hand, d-bus mainly allows for control-rate data exchange, while it it is not suitable as a media bus. A traffic of a few messages per second is expected in typical situations.

2.2. D-BUS

D-Bus is a system for interprocess communication (IPC). Architecturally, it has several layers:

- A library, libdbus, that allows two applications to connect to each other and exchange messages.
- A message bus daemon executable, built on libdbus, that multiple applications can connect to. The daemon can route messages from one application to zero or more other applications.
- Wrapper libraries or bindings based on particular application frameworks. These wrapper libraries are the API most people use, as they simplify the details of D-Bus programming.

libdbus only supports one-to-one connections, just like a raw network socket. However, rather than sending byte streams over the connection, messages are sent.

The message bus daemon forms the hub of a wheel. Each spoke of the wheel is a one-to-one connection to an application using libdbus. An application sends a message to the bus daemon over its spoke, and the bus daemon forwards the message to other connected applications as appropriate. The d-bus daemon acts in fact as a router.

The d-bus daemon can have multiple instances on a typical computer, system-wide and per-session instances. The system-wide and per-user daemons are separate. Normal within-session IPC does not involve the system-wide message bus process and vice versa.

EyeP Communicator only makes use of a single per-user instance for each instance of the Communicator, which is normally one running.

2.3.SIP Component

The SIP component implements the telephony layer. The actual sip layer is in charge of the initiation and management of telephone calls and conferences being compatible with the SIP specification (IETF RFC 3261).

The SIP component receives user and application informations, such as input coming from the UI and settings stored in the MIB, via d-bus and performs all the necessary steps to realize a call (or to manage an incoming call) notifying other components whenever this may be necessary for e.g. a display information or asynchronous action to take place (e.g. display incoming call and wait for Answer or Drop events).

Once the call initiated, the SIP component starts and manages the media stack (see section 3), which is in charge to open and configure possibly secured network connections for media streams, to process the media streams, and to deal with capture and rendering devices.

The SIP component implements support for IMS servers. The IP multimedia Subsystem is a network functional architecture that proposes a solution for facilitating multimedia service creation and deployment, as well as supporting interoperability and network convergence. IMS allows network operators to control traffic distribution. To be noted that IMS support Diameter, a recent Authentication, Authorization, and Accounting (AAA) protocol that can be used for secured transactions, using e.g. Transport Layer Security (TLS).

2.4.Integration of Baseline Systems

When new functionality has to be integrated into eyeP Communicator, one straightforward solution is to interface a new component to the d-bus after defining a proper interface. This is possible when the candidate functionality can share with the rest of the application messages running at the control rate.

It is instead evident that MOBIO baseline systems need to share audio and video information with the EPM media stack, that is, all baseline systems need to receive media streams captured and/or received inside the media stack; this makes rather hard to imagine an integration at the level of the d-bus.

The next section will detail the EPM media stack and how it interacts with the SIP component and with the rest of the Application via d-bus.

3.EPM Media Stack Architecture

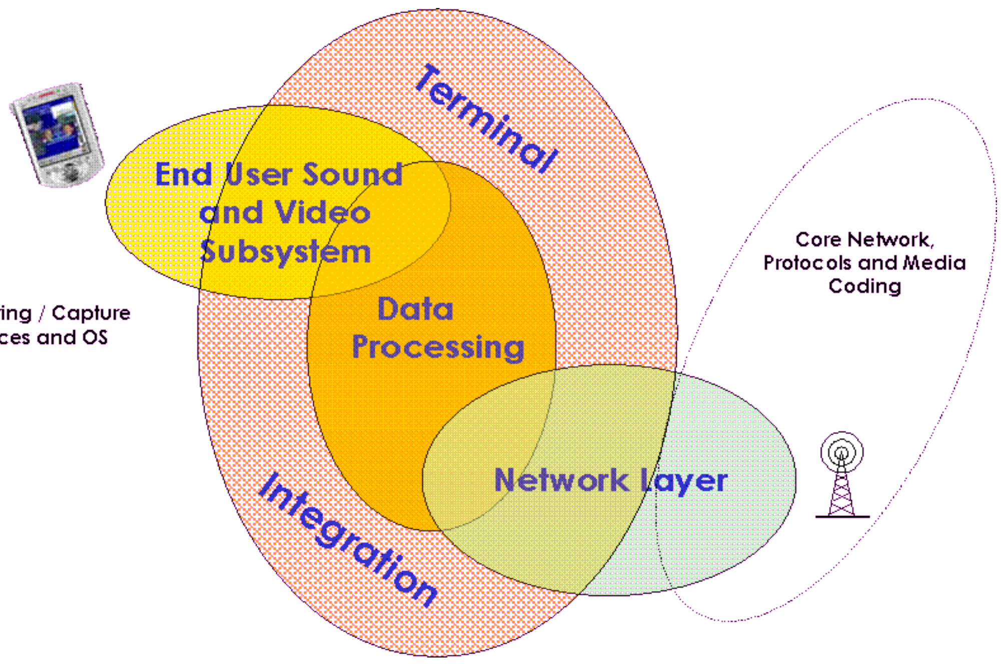
Terminal
End User Sound and Video Subsystem
Data Processing
Network Layer
Integration
Rendering / Capture Devices and OS
Core Network, Protocols and Media Coding

Figure 3.1. eyeP Media media stack main layers.

3.1.Introduction

The EPM media stack (EPM-MS) is a multimedia stack especially designed to deliver a high quality user experience for interactive voice and video communication over IP; it provides in this sense a complete implementation of audio and video streaming capabilities. The media stack supports RTP/RTCP (IETF RFC 1889) streaming with in addition secure RTP if required.

EPM-MS comes with fully integrated audio and visual feature set, by integrating high-end speech coders and advanced digital signal processing tools including Acoustic Echo Cancellation (AEC), Automatic Gain Control (AGC), Voice Activity Detection (VAD), Noise Reduction (NR), Packet Loss Concealment (PLC), and Clock Skew Management in order to improve the user experience in all kind of communication and call-oriented operations.

Furthermore EPM-MS, while being essentially exploited as media stack for software telephones, is also delivered as standalone products, allowing to develop or integrate any other peer-to-peer application requiring interactive voice and video support. It is also possible to set-up conferencing server applications.

Figure 3.1 above shows the three basic layers which are addressed by EPM-MS: the OS and audio-visual subsystem layer, the data processing layer and the signaling and network layer. The three layers are exported to the application (e.g. the CDE soft-phone) through a specific API in order to allow integration of a complete communication terminal. The next section introduces in some more detail the functionality and architecture of the EPM media stack.

3.2.EPM Media Stack features and basic architecture

The whole feature set implemented by EPM-MS, being it at the operating system, signal processing or network layer is realized through the concept of filters as basic building block, similarly to what can be found in similar toolboxes (e.g. MS DirectShow or Apple mov SDK). Among the built-in functionality of EPM-MS, the most relevant is:

- support of a broad range of speech and video codecs: G.711, G.729, gsm 6.10, iLBC, speex (narrow- and wide-band), G.722 (wideband), H.261, H.263, H.264
- fully complies with RTP/RTCP protocols (RFC 1889)
- play and record audio and video over RTP sessions
- (video)conferencing possible among multiple RTP sessions
- NAT/Firewall support
- Low-latency adaptive jitter buffering
- Noise Reducer and Acoustic Echo Canceller to improve the call quality
- Real-time host based voice activity detection for asynchronous transmission and bandwidth optimization
- Audio automatic gain controller
- Support of USB handsets, headsets and USB cameras

The full-feature set is available for Windows NT-based operating systems (Win2K, XP, Vista). A reduced feature set is instead available for Windows Mobile 5 and 6 mobile operating systems, first because some features are already integrated in devices and sometimes cannot be disabled (like AGC on the microphone), but especially because capture and rendering devices are tightly related to the hardware chipsets and because of that the flexibility of audio and video formats is highly reduced in comparison to desktop computer platforms. This is no surprise as mobile portable devices are often less general-purpose machines than desktop workstations are. Furthermore, some processing filters like echo control and noise reduction are implemented with reduced quality to comply with reduced resources and the lack of hardware floating point co-processor units.

Filters in the EPM-MS are organized into *Pipes*, which can be simply seen as daisy chains of filters. Pipes for all the necessary I/O media and network streams are predefined and hard coded (audio input, audio output, video input, video output, network i/o, etc.). Of course new pipes may be added if necessary, but this requires re-compiling the media stack; all filters are in any case conceived to be dynamically attached/removed to/from any pipe, and they are usually tested in such a way, so that the definition of new Pipes is normally a simple matter of connectivity definition.

Figure 3.2 below shows a typical Pipes set for a simple audio-video soft-phone. Yellow boxes indicate Pipes, the green background highlighting the receiving side and the pink one the transmitting one.

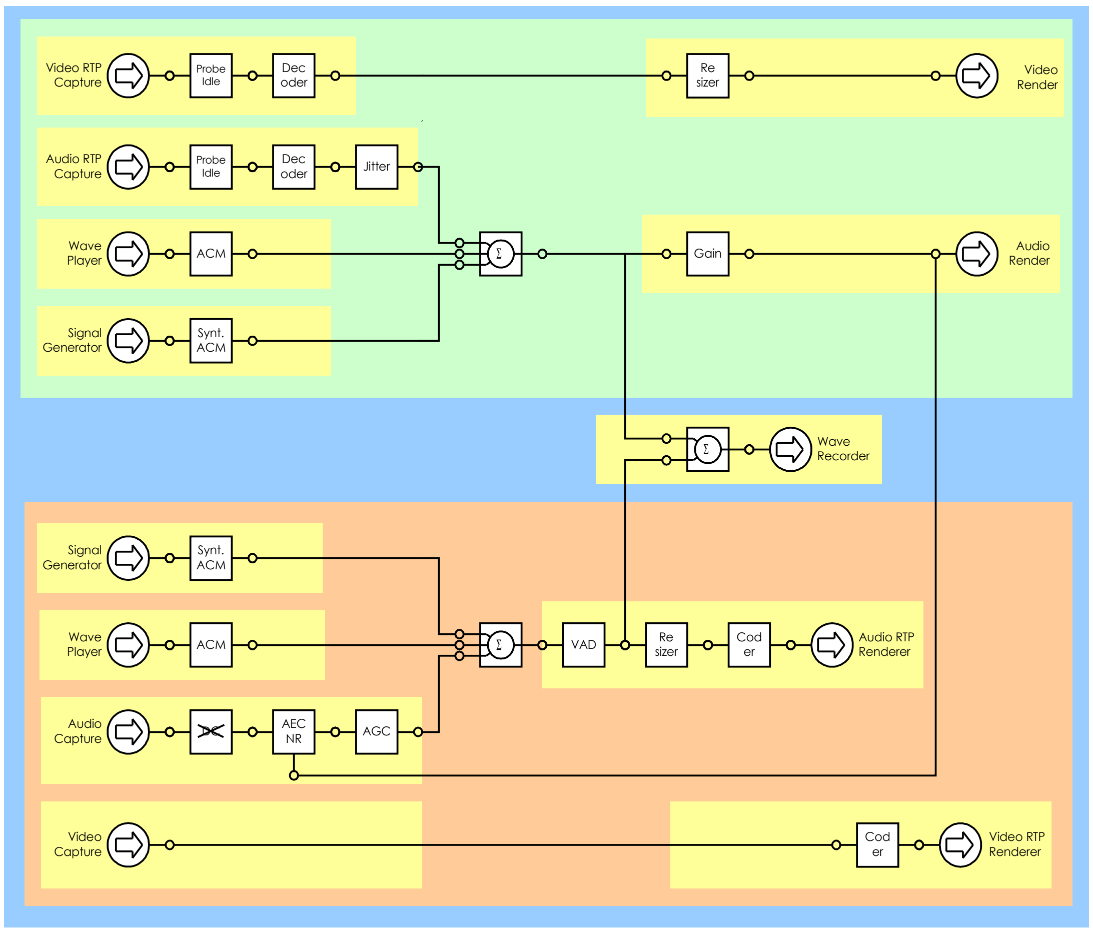

Figure 3.2. eyeP media simple Filter connections into Pipes. Yellow boxes indicate Pipes, the green background highlighting the receiving side and the pink one the transmitting one.

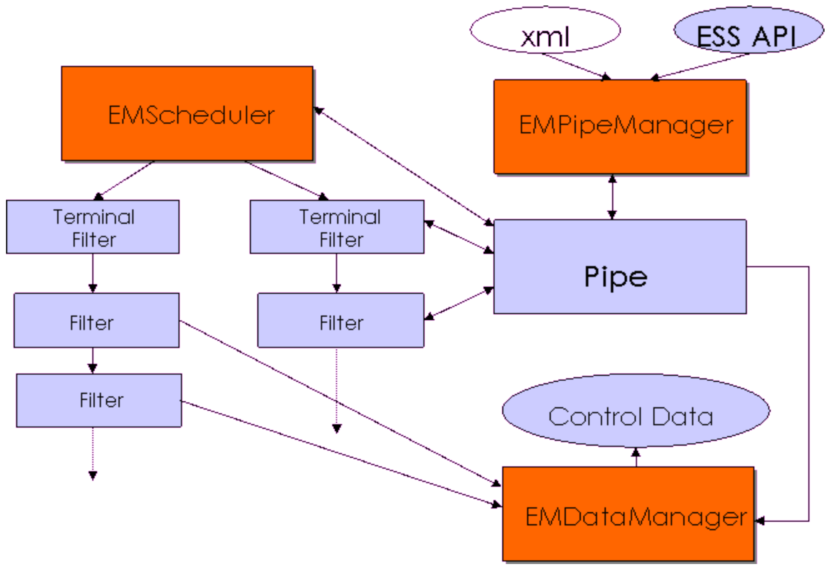

Figure 3.3. EPM media stack main building blocks.

In Pipes, Filters are connected through *Pins,* up until *termination* filters that are the input points to execution of Pipes at scheduler clock ticks.

3.3.Integration of third-party functionality into EPM-MS

EPM-MS allows integrating new codecs through the mechanism of plug-in filters, but the same is not possible for signal processing and signaling/network filters. Nevertheless, the internal filter encapsulation structure inherits from a unique base Filter class, which provides a general filter interface to be used for the integration of new functionality, whenever this may be necessary.

Parameters at the inner stack layers (like control of specific filters) cannot be directly addressed from outside the media stack; instead it is possible to specify extensions to the API so that proper actions can be executed. The API is the proper way to address control of the specific filters (see Figure 3.4).

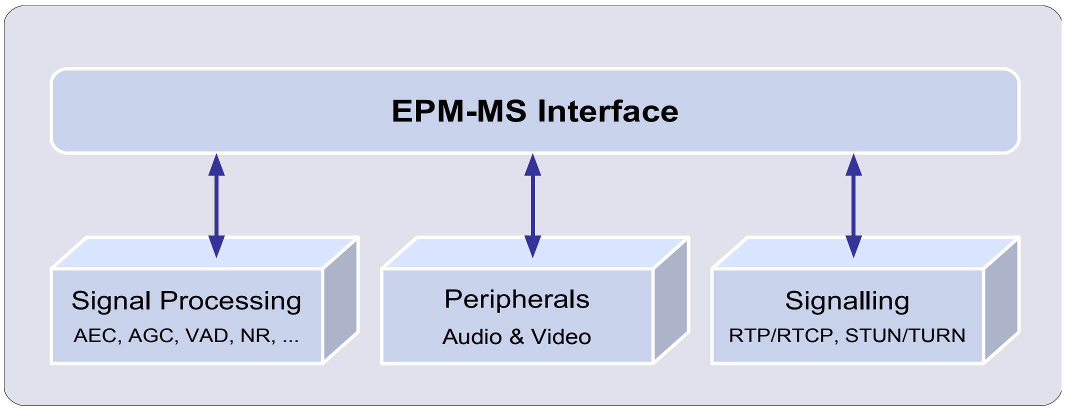

Figure 3.4. The EPM-MS API interface allows accessing filter controls at the lower layers.

The Filter class abstraction is included in Annex A of this document for a complete reference (section 7.2 *EMStreamFilterImpl.h*).

As said earlier, filters are linked to each other through pins, and they communicate through Buffers which are returned over the connection Pins. Two types of buffers are defined, audio and video buffers inheriting from a common buffer base class; the definition of new buffer types or extensions to current buffer types is possible. Annex A also includes the buffer class definition for a complete reference (section 7.1 *EMStreamBufferImpl.h*).

4.Integration of Baseline Systems into EPM Media Stack

4.1.Introduction

This section is a draft guideline, based on the draft API specification contained in Annex A, of how a MOBIO baseline system may be integrated in the EPM-MS in form of a new filter; since this cannot be done in terms of external plug-in, it will be necessary that an empty filter is created inside EPM-MS; at the same time once the library API and data buffers correspond to the EMFilter and EMBuffer implementations, this integration and link will be straightforward. Next section is a short summary of the way in which baseline systems are pipelined to each other (deliverable D3.1), while section 4.3 consider in more detail integration concepts and requirements. Section 4.4 provides an example of how a module library can be modified to be accommodated in a new filter. Finally, section 4.5 shortly anticipate how baseline systems can be integrated into an advance authentication system through an iterative and refinement process.

4.2.Baseline Systems

Work done in WP3 for D3.1 (Baseline systems for uni-modal authentication) led to the definition of classes and functions to read and write data; these classes and functions are used to pass information between command line tools in the MOBIO project, supposing that these tools are available as executable files that can be scripted for execution in daisy-chain.

All tasks (detection, feature localization and authentication) use a common *mobio_inputfile* object that essentially defines the original audio-video file (and then the audiovisual stream); as such, all modules need to receive the raw media information.

In addition to the input file, the tools exchange from one to the other: lists of faceboxes (coordinates), scores, lists of faces (face points) with scores, overall scores organized in *segments* (that is, scores that apply to a certain number of frames).

The EMFilter class defines buffers and some related parameters. See subsection “4.3/Adaptation of audio and video buffer data” later on for a discussion on how this information can be accommodated in the data flow.

4.3.Integration Concepts: Requirements and Constraints

In order to allow seamless integration of baseline systems as EPM-MS filters and pipeline them it is necessary to work at two levels:

- adapt the library interface so that it can fit into the base filter class interface to allow the real time process on the media
- adapt the buffer classes so that they can allow the propagation of useful additional information (additional to basic audio and video samples) that must be made available by one module to the next one in the chain.

Adaptation of library interfaces

Two important remarks to start with:

- pipes in EPM-MS are conceived as a virtual DSP data-paths, and as such are designed and can only work with a periodical execution over a "frame" of data, being this an actual video frame in the case of video or a buffer that can be typically 10 ms long in the case of audio (speech). This implies that all processing (baseline) filter blocks must be made in a way that allows processing per-frame and not in an off-line working mode.
- as part of a real-time data flow, all filters must be able to react to control-rate changes in typical rates and settings: sampling rate for audio, frame rate and/or frame size for video. SetSamplingRate, SetFrameSize in the Filter class support this functionality. This does not mean that the library must implement a dynamic adaptation: in the worst case the e.g. SetSamplingRate method will be overloaded with code that simply shuts down the tool and restart it with the new settings. But especially if this is very resource-consuming (many memory allocation etc.) this should be avoided as far as possible.

This second remark introduces an important point: the methods of the filter base class are not purely virtual ones, they have a base implementation in the filter class itself, and must be overloaded only when specific functionality need to be executed. Next sub-section will clarify this point through a provided example.

Once this done, the main functions correspond to typical media streaming functionality: initialize and start, stop and shutdown, pause, restart, and processing, constitute the main candidates for an overloading inside a filter (together with the already mentioned management of dynamic changes).

A special feature being implemented in the case of video filters (when the frame rate may be 5 to 10 times lower than the control rate typically at 100 Hz) is the possibility to "pipeline" the execution of the processing methods in 2 or 3 successive steps. This is particularly useful when a processing is very demanding in terms of computing resources, in order to avoid a peak in CPU load during a certain clock tick (in the EPM-MS sense) for 10ms only, with the risk to block the normal real-time processing if the CPU is not released in time, while the successive clock periods will be much lower. The possibility to execute processing in 2-3 successive steps is then encouraged when possible and this addition to the API may be finalized before the beginning of the actual integration.

Adaptation of audio and video buffer data

As explained earlier, baseline systems communicate between each other through text files essentially containing tables of data. These data can not be included yet in the buffering mechanism of EPM-MS (raw text or XML generic data transfer is supported in other parts of the stack but not in Buffers); this means that an extension of the buffer specification will be necessary.

Two solutions may be possible: a) define an additional general purpose buffer other than the audio and video one, or b) extend the audio and video buffers to allow the addition of extra information.

Solution b) seems preferable for two main reasons. The first is that this does not impact the mainly single buffer communication mechanism between filters; and the second is that in any case the additional information is tightly related to audio and video frames, so an extra buffer would most probably require a synchronization mechanism to keep the different kinds of information in exact order.

EPM will take care to extend the buffer definition in terms of code implementation. The draft definition of the buffer extensions are given in section 5, while the final specification will be refined prior to starting code integration.

4.4.Example

In order to better exemplify what each baseline system is supposed to provide in order to be integrated in EPM-MS, we can consider the case of integration of a face localizer. In such a case EPM is in charge of implementing the empty filter encapsulation, while the providing partner is supposed to provide through a convenient local API, equivalent functionality to that needed at the corresponding filter method.

In the case of a simple face localization tools, the following methods may need to be overloaded (for each of them it is explained what exactly it must contain). Please note that in the filter method specification all parameters referring to the graph, the pipe and overall timestamps are managed by the encapsulation part and do not need to be used by the library.

1) Constructor and Destructor

The filter inherits from the base class CEMStreamFilterImpl, which provides the base implementation of the virtual methods. All base methods not overloaded here are of course executed from the base class directly when necessary

```
CEMStreamFilterFaceLocImpl::CEMStreamFilterFaceLocImpl(CEMStreamGraphImpl *_pGraph, CEMStreamPipeImpl *_pPipe):
CEMStreamFilterImpl(EMStreamFilterType_FaceFinder, _pGraph, _pPipe)
, m_ulInWidth(0), m_ulInHeight(0), m_ulNbFaces(0), m_pWorkingBuffer(NULL),
m_ulSizeDataRGB(0), m_OutputCodec(EMStreamCodec_RGB),
m_ulSizeDataYUV(0), m_pFaceFinder(NULL)
{
	// all the code necessary to create (allocation) of the library should go here
	// please try to avoid putting large memory allocations later than this, as it will run in real-time
}

CEMStreamFilterFaceLocImpl::~CEMStreamFilterFaceLocImpl()
{
	// destructor code
}
```

2) Configuration

```
bool CEMStreamFilterFaceLocImpl::SetConfiguration(LPCSTR _szConfiguration)
{
	// this method may be used to configure through external xml file the initialization of the filter
	// note that this cannot be used for updates at runtime
	// it would be preferable not to use a config here as it may imply hard setting, but instead let the input buffer also do that
}
```

3) Processing

```
CEMStreamBufferImpl *CEMStreamFilterFaceLocImpl::OnProcess(ULONG _ulGraphTimeStamp, ULONG _ulLength)
{
	// Retrieve data on the input pin (calling the base class OnProcess first to retrieve input from previous filter
	CEMStreamBufferImpl *pBuffer = CEMStreamFilterImpl::OnProcess(_ulGraphTimeStamp, _ulLength);
	if (!pBuffer)
	{
		// the following can be used for debugging purposes anywhere in filter methods
		EMTrace("CEMStreamFilterFaceLocImpl", TRACE_INFO, "::OnProcess() : No data received");
		return NULL;
	}

	// Put here further custom code to be executed at every clock tick
	// Code should include calls to actual processing
	// Code should include management of format changes

	// store the information of the detected faces (if any). E.G.
	 // get the current face information
	ff_faceinfo_t faceinfo = ff_faceinfo_get(m_pFaceFinder, i);

	// add necessary information in faceinfo to the corresponding field(s) of the buffer

	// At the end return the video pBuffer as this is being called from following filter
	if (/*possibly conditions met*/)
		return pBuffer;
	else
		return m_pWorkingBuffer; // buffer needed to be copied into another one because content was changed
}
```

4) Start and Stop

```
EEMStreamErrors CEMStreamFilterFaceLocImpl::Start()
{
	// if necessary put here code needed to start the processing functions.
	// a startup of the tool that should not include big memory allocation
	// with or without additional code, call the base start so that all the common flags are set
	return CEMStreamFilterImpl::Start();
}

EEMStreamErrors CEMStreamFilterFaceLocImpl::Stop()
{
	// the same as Start
	return CEMStreamFilterImpl::Stop();
}
```

4) Connection and Disconnection

```
// these are normally not necessary. Put here something if necessary when the input pin is connected or disconnected

bool CEMStreamFilterFaceLocImpl::OnConnect(CEMStreamPinInputImpl *_pInputPin)
{
	return CEMStreamFilterImpl::OnConnect(_pInputPin);
}

bool CEMStreamFilterFaceLocImpl::OnDisconnect(CEMStreamPinInputImpl *_pInputPin)
{
	return CEMStreamFilterImpl::OnDisconnect(_pInputPin);
}
```

5) Misc

```
EMString CEMStreamFilterFaceLocImpl::GetName()
{
            // may be used for debug purposes
            return "Face Localization Filter";
}

// and then it is possible to add further class methods to be used by the above, mainly private methods
```

4.5.Towards advanced authentication systems

MOBIO will aim at combining simple baseline authentication system into an advanced authentication technology. The first step towards this achievement will be an iterative process following this first draft API specification. All partners involved in integration will receive example code through which it will be possible to start the integration process and to find out possible inconsistencies or missing parameters in the first specification.

At the end of this iterative process, it will be fundamental to refine and clarify the specification of the merging of different data types. As a first idea, the EPM-MS allows to merge two or more pipes into a new one with specified functionality. In this sense, other than defining new baseline filters, it is possible to design a new simple *Authentication Pipe*, which may receive audio and video buffers with the necessary buffer fields inside and proceed to the necessary post-processing and/or merging of information. The final specification of this process will be provided with deliverable D6.3.

Figure 4.1 below shows a first example of enhanced media stack including MOBIO functionality.

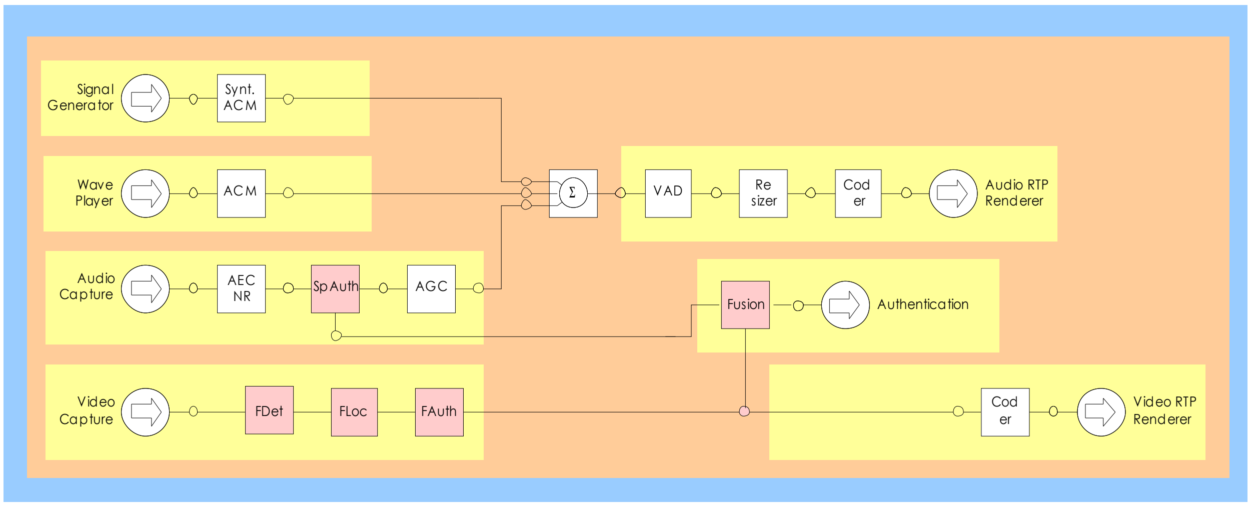

Figure 4.1. Filter connections into Pipes: a simple example of new input stream structure with MOBIO functionality integrated. New filters are highlighted in pink: Speech Authentication in the Audio Capture pipe; Face Detection, Localization, and Authentication in Video Capture; Data Fusion to provide en enhanced authentication system (will be reviewed in further releases).

5.Summary

This documents has presented a quick introduction to the main architectural elements of the eyeP Communicator for both desktop and mobile platforms.

The EPM media stack has been introduced in more detail as it is the actual module where MOBIO baseline systems will have to be integrated; furthermore an example and a few source code definition files have been provided for better understanding the actual integration process.

Some necessary modifications to the existing classes and buffer definitions have been identified. Other minor modifications may be necessary once actual baseline systems are being prepared for integration.

Before integration will start, EPM will provide to partners one or two examples of actual filter codes to be used as templates for code refinement.

6.Glossary

3GPP: 3rd Generation Partnership Project. A Committee of telecommunications associations, to make a globally applicable third generation (3G) mobile phone system specification.

CDE: Communicator Desktop Edition

CME: Communicator Mobile Edition

EPM-MS: eyeP Media – Media Stack. A software toolset providing media streaming and processing functionality for V^{2}oIP applications.

IETF: Internet Engineering Task Force. An international Committee developing and promoting Internet standards.

IMS: IP Multimedia Subsystem. A standardized solution for Session Initiation Protocol (SIP) based applications for multiple accesses.

IPC: Inter-Process Communication. A set of software interfaces, which is usually programmed in order for a programmer to communicate between a series of processes.

MIB: Management Information Base. A type of database used to manage the devices in a communications network.

RFC: Request for Comments. A document that describes the specifications for a recommended technology. Although the word "request" is in the title, if the specification is ratified, it becomes a standard document.

RTP: Real-Time Transport Protocol. A protocol standardizing a packet format for delivering audio and video over the Internet

SIP: Session Initiation Protocol. A signaling protocol used for negotiation and release of multimedia communication sessions such as voice and video calls over the Internet.

SRTP: Secure Real-Time Transport Protocol. A profile of RTP providing encryption, message authentication and integrity, and replay protection to the RTP data.

TLS: Transport Layer Security. A protocol that guarantees privacy and data integrity between client/ server applications communicating over the Internet.

UI: User Interface. Everything designed into an information device with which a human being may interact, including screen, mouse, keyboard, etc.

VoIP: Voice over IP. The technology that turns voice conversations into data packets and sends them out over a packet-switched Internet Protocol (IP) network.

V^{2}oIP: (V2OIP) Voice and Video over IP. The extension of VoIP to audio-visual communications.

7.Acknowledgments

The author would like to thank all the MOBIO partners for useful feedback and review during the preparation of the deliverable.

8.Annex A

Two files from EPM-MS are reported here for a more comprehensive reference to the buffer and filter classes. For sake of clarity, some methods and class members are in bold, so that the files can be read more easily. For each method, exact explanation of input and output parameters is given in form of C++ comment just above.

The class CEMStreamBufferVideoImpl at the end of the following subsection is the best place where mobio_video_faceboxes, mobio_video_face_points, and mobio_output objects (see again deliverable D3.1) can be added in order to fully support MOBIO compatible data streams

8.1.EMStreamBufferImpl.h

```
#ifndef EM_STREAMBUFFERIMPL_H
#define EM_STREAMBUFFERIMPL_H

#include "EMBuffer.h"
#include "EMPlugInCodec.h"

#define SAMPLESPERSEC 8000
#define BITSPERSAMPLE 16
#define CHANNELS 1

/// Internal format of audio data
extern SEMStreamBufferAudioFormat g_DefaultAudioFormat;

/// Internal size of audio data in ms
extern ULONG g_ulDefaultAudioSizeMs;

/// Internal size of audio data in samples
extern ULONG g_ulDefaultAudioSizeSamples;
/// Internal size of audio data in bytes
extern ULONG g_ulDefaultAudioSizeBytes;

#ifdef EM_VIDEO

// Theses values are for CIF picture on RGB format
#define VIDEOWIDTH 640
#define VIDEOHEIGHT 480
#define VIDEOBITSPERPIXEL 24
#define VIDEOFRAMERATE 15

/// Internal format of video data
const SEMStreamBufferVideoFormat g_DefaultVideoFormat = { VIDEOWIDTH, VIDEOHEIGHT,
VIDEOBITSPERPIXEL, VIDEOFRAMERATE };
/// Internal size of video data in bytes
const ULONG g_ulDefaultVideoSizeBytes = VIDEOWIDTH * VIDEOHEIGHT * VIDEOBITSPERPIXEL / 8;

#endif //EM_VIDEO

/** Initialize global audio settings
@param _ulClock Clock of the graph in ms
*/
void FEMInitializeAudioSettings(ULONG _ulClock);


/// Forward declaration of CEMStreamBufferManagerImpl class. It implements a buffer pool
class CEMStreamBufferManagerImpl;

class CEMStreamBufferImpl : public IEMPlugInBuffer
```

```
{
        friend CEMStreamBufferManagerImpl;

protected:
        /** Constructor
        @param _Type Type of the buffer
        @param _ulSize Size of the requested buffer
        */
        CEMStreamBufferImpl(EEMStreamBufferType _Type, ULONG _ulSize);

        /** Constructor with data copy
        @param _pBuffer Pointer of the source buffer to copy
        */
        CEMStreamBufferImpl(CEMStreamBufferImpl *_pBuffer);

        /** Destructor
        */
        virtual ~CEMStreamBufferImpl();
public:

        /** Retrieve the type of the buffer
        @return Type of the buffer
        */
        EEMStreamBufferType GetType();

        /** Retrieve the pointer on data
        @return Pointer on data
        */
        BYTE *GetPointer();

        /** Retrieve the user size of the buffer
        @return User size of the buffer
        */
        ULONG GetSize();

        /** Copy data from an other buffer
        @param _pBuffer Pointer of the source buffer to copy
        @param _bCopyData Flag that indicate if the copy of data is made
        */
        virtual void Copy(IEMPlugInBuffer *_pBuffer, bool _bCopyData = true);

        /** Copy data to an external buffer
        @param _pBuffer Pointer of the destination buffer
        */
        virtual void CopyTo(SEMStreamBuffer *_pBuffer);

        /** Copy data from an external buffer
        @param _pBuffer Pointer of the source buffer
        */
        virtual void CopyFrom(SEMStreamBuffer *_pBuffer);

        /** Reset the content of the buffer
    @param _bKeepHeader Flag that indicate if the header of buffer is cleared. True to clear only
data, False to clear data and header. Default is false.
        */
        virtual void Clear(bool _bKeepHeader = false);

        /** Set the size of data
        @param _ulActualDataLength Number of data in bytes
        */
        void SetActualDataLength(ULONG _ulActualDataLength);

        /** Retrieve the size of data
        @return Number of data in bytes
        */
        ULONG GetActualDataLength();

        /** Set the time stamp of data
        @param _ulStartTime Start time of data in ms
        @param _ulEndTime End time of data in ms
        */
        void SetTime(ULONG _ulStartTime, ULONG _ulEndTime);

        /** Retrieve the time stamp of data
        @param _pulStartTime Pointer to the start time of data in ms
        @param _pulEndTime Pointer to the end time of data in ms
        */
        void GetTime(ULONG *_pulStartTime, ULONG *_pulEndTime);

        /** Retrieve the start time stamp of data
        @return Start time of data in ms that the buffer contains
        */
        ULONG GetStartTime();

        /** Retrieve the time length of data
        @return Time length of data in ms that the buffer contains
        */
        ULONG GetLengthTime();

        /** Retrieve the end time stamp of data
        @return End time of data in ms that the buffer contains
        */
        ULONG GetEndTime();

        /** Set if data follow a silence period
        @param _bDiscontinuity True if data follow a silence period. False if data are the
continuuity of the previous one.
        */
        void SetDiscontinuity(bool _bDiscontinuity);

        /** Retrieve if data follow a silence period
        @return True if data follow a silence period. False if data are the continuuity of the
previous one.
        */
        bool IsDiscontinuity();

        /** Set the type of data
        @param _Codec Type of data.
        */
        void SetCodec(USHORT _uiPayload);

        /** Retrieve the type of data
        @return Type of data.
        */
        USHORT GetCodec();

        /** Set if other data are available to provide for the same graph tick
        @param _bAlone True if no more data are available. False if other data are available.
        */
        void SetAlone(bool _bAlone);

        /** Retrieve if other data are available for the same graph tick
        @return True if no more data are available. False if other data are available.
        */
        bool IsAlone();

        /** Set if data are allowed to be mixed with other data
        @param _bMixingAllowed True if data are allowed to be mixed. False if data are not allowed
to be mixed.
        */
        void SetMixingAllowed(bool _bMixingAllowed);

        /** Retrieve if data are allowed to be mixed with other data
        @return True if data are allowed to be mixed. False if data are not allowed to be mixed.
        */
        bool IsMixingAllowed();

        /** Set number of buffer lost before this buffer
        @param _ulDataLost Number of buffer lost.
        */
        void SetDataLost(ULONG _ulDataLost);

        /** Retrieve the number of data lost
        @return Number of data lost
        */
        ULONG GetDataLost();

        /** Lock data access
        */

        void LockData();

        /** Unlock data access
        */

        void UnlockData();

    /** Retrieve the buffer related to the main one, for mm or recording
    @return Pointer to the requested buffer
    */
        CEMStreamBufferImpl *GetRelatedBuffer();

    /** Set a secondary buffer related to the main one, for mm or recording
    @param _pRelBuf buffer to set the related one
    */
        void SetRelatedBuffer(CEMStreamBufferImpl *_pRelBuf);

public:
                /** Resize the buffer
        @param _ulSize New allocated size of the requested buffer
        */
        void Resize(ULONG _ulSize);

protected:
        /// Type of the buffer. EMStreamBufferType_Audio or EMStreamBufferType_Video
        EEMStreamBufferType m_Type;
        /// Pointer to the data
        BYTE *m_pucData;
        /// Type of data (raw, RGB, yuv, coded, etc.)
        USHORT m_Codec;

        /// User size of the buffer
        ULONG m_ulSize;
        /// Allocated size of the buffer
        ULONG m_ulSizeAllocated;
        /// Available number of data in bytes
        ULONG m_ulActualDataLength;

        /// Start time stamp of data in ms
        ULONG m_ulStartTime;
        /// End time stamp of data in ms
        ULONG m_ulEndTime;

        /// Flag that indicate if data follows a silence period
        bool m_bDiscontinuity;
        /// Flag that indicate if other data are available
        bool m_bAlone;
        /// Flag that indicate if data are allowed to be mixed with other data
        bool m_bMixingAllowed;
        /// Number of buffer lost before this buffer
        ULONG m_ulDataLost;

        /// Attached buffer possibly containing data for video (or mm) transport or recording
        CEMStreamBufferImpl *m_pRelatedBuffer;

        // this is used to lock buffers during copy, to prevent the other thread writing on it at
the same time
        CEMLock m_LockBuffer;
        /** Set the user size the buffer
        @param _ulSize New user size of the requested buffer
        */
        void SetSize(ULONG _ulSize);

        /** Retrieve the allocated size of the buffer
        @return Allocated size of the buffer
        */
        ULONG GetSizeAllocated();
};

class CEMStreamBufferAudioImpl : public CEMStreamBufferImpl, public IEMPlugInBufferAudio
{
        friend CEMStreamBufferManagerImpl;

protected:
        /** Constructor
        @param _ulSize Size of the requested buffer
        */
        CEMStreamBufferAudioImpl(ULONG _ulSize);

        /** Constructor with data copy
        @param _pBuffer Pointer of the source buffer to copy
        */
        CEMStreamBufferAudioImpl(CEMStreamBufferAudioImpl *_pBuffer);

        /** Destructor
        */
        virtual ~CEMStreamBufferAudioImpl();

public:
        /** Copy data from an other buffer
        @param _pBuffer Pointer of the source buffer to copy
        @param _bCopyData Flag that indicate if the copy of data is made
        */
        virtual void Copy(IEMPlugInBuffer *_pBuffer, bool _bCopyData = true);

        /** Copy data to an external buffer
        @param _pBuffer Pointer of the destination buffer
        */
        virtual void CopyTo(SEMStreamBuffer *_pBuffer);

        /** Copy data from an external buffer
        @param _pBuffer Pointer of the source buffer
        */
        virtual void CopyFrom(SEMStreamBuffer *_pBuffer);

    /** Reset the content of the buffer
    @param _bKeepHeader Flag that indicate if the header of buffer is cleared. True to clear only
data, False to clear data and header. Default is false.
    */
       virtual void Clear(bool _bKeepHeader = false);

        /** Set the format of data
        @param _ulSamplesPerSec Number of samples per second
        @param _uiBitsPerSample Number of bits per sample
        @param _uiChannels Number of channels
        */
        virtual void SetFormat(ULONG _ulSamplesPerSec, USHORT _uiBitsPerSample, USHORT
_uiChannels);

        /** Set the format of data
        @param _ulSamplesPerSec Number of samples per second
        @param _uiBitsPerSample Number of bits per sample
        @param _uiChannels Number of channels
        */
        virtual void GetFormat(ULONG *_pulSamplesPerSec, USHORT *_puiBitsPerSample, USHORT
        *_puiChannels);

        /** Set the format of data
        @param _Format Format of data.
        */
        void SetFormat(SEMStreamBufferAudioFormat &_Format);

        /** Retrieve the format of data
        @return Format of data.
        */
        SEMStreamBufferAudioFormat &GetFormat();

        /** Set the status boolean to full
        */

        void BufferStatusFull();

        /** Set the status boolean to empty
        */

        void BufferStatusEmpty();

        /** Return the status of m_bFull
        */

        bool BufferFull();

    /** Retrieve the specialized interface of the buffer
    @return Pointer on the specific interface
    */
    virtual void *GetSpecificInterface();

         /** Initialize the Data Buffer with negligeble data , sinusoid {2,0,-2,0}, for silence
packets purposes
    */
         void SetSilenceData();

protected:
        /// Format of data
        SEMStreamBufferAudioFormat m_Format;

        // this is used to know the status of the buffer, full to consume or just consumed and ready
to fill
        bool m_bFull;
};

class CEMStreamBufferVideoImpl : public CEMStreamBufferImpl, public IEMPlugInBufferVideo
{
        friend CEMStreamBufferManagerImpl;

protected:
        /** Constructor
        @param _ulSize Size of the requested buffer
        */
        CEMStreamBufferVideoImpl(ULONG _ulSize);

        /** Constructor with data copy
        @param _pBuffer Pointer of the source buffer to copy
        */
        CEMStreamBufferVideoImpl(CEMStreamBufferVideoImpl *_pBuffer);

        /** Destructor
        */
        virtual ~CEMStreamBufferVideoImpl();

public:

        /** Copy data from an other buffer
        @param _pBuffer Pointer of the source buffer to copy
        @param _bCopyData Flag that indicate if the copy of data is made
        */
        virtual void Copy(IEMPlugInBuffer *_pBuffer, bool _bCopyData = true);

        /** Copy data to an external buffer
        @param _pBuffer Pointer of the destination buffer
        */
        virtual void CopyTo(SEMStreamBuffer *_pBuffer);

        /** Copy data from an external buffer
        @param _pBuffer Pointer of the source buffer
        */
        virtual void CopyFrom(SEMStreamBuffer *_pBuffer);

    /** Reset the content of the buffer
    @param _bKeepHeader Flag that indicate if the header of buffer is cleared. True to clear only
data, False to clear data and header. Default is false.
    */
    virtual void Clear(bool _bKeepHeader = false);

        /** Set the format of data
        @param _ulWidth Width of the video resolution
        @param _ulHeight Height of the video resolution
        @param _ulBytesPerPixel Number of bytes used to code 1 pixel
        @param _fFramesPerSec Number of frames per second. Can be float in many standards
        */
        virtual void SetFormat(ULONG _ulWidth, ULONG _ulHeight, ULONG _ulBytesPerPixel, float
_fFramesPerSec = 0.0);

        /** Set the format of data
        @param _Format Format of data.
        */
        void SetFormat(SEMStreamBufferVideoFormat &_Format);

        /** Retrieve the format of data
        @return Format of data.
        */
        SEMStreamBufferVideoFormat &GetFormat();

    /** Retrieve the specialized interface of the buffer
    @return Pointer on the specific interface
    */
    virtual void *GetSpecificInterface();

        /** Set if the buffer has already been used once or not
    @param _iUsed is a new one (false) or has been used (true)
    */
        void SetUsed(bool _iUsed);

        /** Get if the buffer has already been used once or not
    @return _iUsed is a new one (false) or has been used (true)
    */
        bool GetUsed();

protected:
        /// Format of data
        SEMStreamBufferVideoFormat m_Format;

        /// check if it has already been used
        bool m_iIsUsed;

};

#endif // EM_STREAMBUFFERIMPL_H
```

8.2.EMStreamFilterImpl.h

```
#ifndef EM_STREAMFILTERIMPL_H
```

```
#define EM_STREAMFILTERIMPL_H

#include "EMStreamBufferManagerImpl.h"
#include "EMStreamPinImpl.h"

enum EEMStreamFilterState
{
        EMStreamFilterState_Stopped,
        EMStreamFilterState_Running,
        EMStreamFilterState_Paused
};

// all the currently defined filters are enumerated here
enum EEMStreamFilterType
{
        EMStreamFilterType_PlugAudio = 0,
#ifdef EM_VIDEO
        EMStreamFilterType_PlugVideo=1,
        EMStreamFilterType_MixerVideo=2,
        EMStreamFilterType_ResizerVideo=3,
        EMStreamFilterType_CoderVideo=4,
        EMStreamFilterType_DecoderVideo=5,
#endif // EM_VIDEO
        EMStreamFilterType_PlugTransport=6,
        EMStreamFilterType_CoderAudio=7,
        EMStreamFilterType_DecoderAudio=8,
        EMStreamFilterType_MixerAudio=9,
        EMStreamFilterType_GeneratorAudio=10,
        EMStreamFilterType_DspDcRemove=11,
        EMStreamFilterType_JitterAudio=12,
        EMStreamFilterType_ResizerAudio=13,
#ifdef EM_VAD
        EMStreamFilterType_ProbeSilence=14,
#endif // EM_VAD
#ifdef EM_AGC
        EMStreamFilterType_DspAgc=15,
#endif // EM_AGC
#if defined(EM_AES)|| defined(EM_PAEC)
        EMStreamFilterType_DspAec=16,
#endif // EM_PAEC || EM_AES
#ifdef EM_NOISEREDUCTOR
        EMStreamFilterType_DspNr=17,
#endif // EM_NOISEREDUCTOR
#ifdef EM_AUDIO_CONVERTER
        EMStreamFilterType_ConverterAudio=18,
#endif // EM_AUDIO_CONVERTER
#ifdef EM_PLAYER
        EMStreamFilterType_PlayerAudio=19,
#endif // EM_PLAYER
#ifdef EM_RECORDER
        EMStreamFilterType_RecorderAudio=20,
#endif // EM_RECORDER
#ifdef EM_CONFERENCE
        EMStreamFilterType_ConferenceAudio=21,
#endif // EM_CONFERENCE
        EMStreamFilterType_VolumeAudio=22,
        EMStreamFilterType_ProbeIdle=23,
        EMStreamFilterType_Synchro=24,
#ifdef EM_MONITOR
        EMStreamFilterType_ProbeMeter=25,
#endif // EM_MONITOR
        EMStreamFilterType_ErrorConcealmentAudio=26,
#ifdef EM_AUDIO
    EMStreamFilterType_PlugAudioDeviceInput=27,
    EMStreamFilterType_PlugAudioDeviceOutput=28,
#endif // EM_AUDIO
#ifdef EM_VIDEO
    EMStreamFilterType_PlugVideoDeviceInput=29,
    EMStreamFilterType_PlugVideoDeviceOutput=30,
#endif // EM_VIDEO
#ifdef EM_RTP
```

```
    EMStreamFilterType_PlugTransportRtpAudioInput=31,
    EMStreamFilterType_PlugTransportRtpAudioOutput=32,
    EMStreamFilterType_PlugTransportRtpVideoInput=33,
    EMStreamFilterType_PlugTransportRtpVideoOutput=34,
#endif // EM_RTP
#ifdef EM_AES
       EMStreamFilterType_DspAes=35,
#endif // EM_AES
#ifdef EM_PAEC
       EMStreamFilterType_DspPaec=36,
#endif // EM_PAEC
#ifdef EM_SIMULATOR
    EMStreamFilterType_SimulatorPacketLoss=37,
#endif // EM_SIMULATOR
#ifdef EM_CONFERENCE
    EMStreamFilterType_ConferenceOutputAudio=38,
#endif // EM_CONFERENCE
#ifdef EM_RECORDERVIDEO
       EMStreamFilterType_RecorderVideo=39,
#endif // EM_RECORDERVIDEO
#ifdef EM_MMUXER
       EMStreamFilterType_MuxerMultimedia=40,
#endif // EM_MUXER
#ifdef EM_SPEECHENHANCER
       EMStreamFilterType_DspSpEn=41,
#endif //EM_SPEECHENHANCER
#ifdef EM_FACEFINDER
    EMStreamFilterType_FaceFinder=42,
#endif // EM_FACEFINDER
#ifdef EM_VIDEOCONF
    EMStreamFilterType_ConferenceOutputVideo=43,
       EMStreamFilterType_ConferenceVideo=44,
#endif // EM_CONFERENCE
#ifdef EM_NOREFVIDEOQUALITY
       EMStreamFilterType_NoReferenceVideoQuality=45,
#endif

};

enum EEMStreamFilterEvent
{
       EMStreamFilterEvent_Data,
       EMStreamFilterEvent_Complete,
};

// this one is corresponding to StreamPin types for characterization or realtime behavior
enum EEMStreamFilterRTType
{
       EMStreamFilterRTType_NoType,
       EMStreamFilterRTType_Input,
       EMStreamFilterRTType_Output,
       EMStreamFilterRTType_LiveAudio,
       EMStreamFilterRTType_LiveVideo,
       EMStreamFilterRTType_Recording,
};

/// Forward declaration of the CEMStreamGraphImpl class
class CEMStreamGraphImpl;
/// Forward declaration of the CEMStreamPipeImpl class
class CEMStreamPipeImpl;
/// Forward declaration of the CEMStreamPinInputImpl class
class CEMStreamPinInputImpl;

class CEMStreamFilterImpl
{
       friend CEMStreamGraphImpl;
       friend CEMStreamPinInputImpl;

protected:
       /** Constructor
       @param _Type Type of the filter
```

```
        @param _pGraph Pointer to the graph that contains the filter
        @param _pPipe Pointer to the pipe that contains the filter
        */
        CEMStreamFilterImpl(EEMStreamFilterType _Type, CEMStreamGraphImpl *_pGraph,
CEMStreamPipeImpl *_pPipe);

        /** Destructor
        */
        virtual ~CEMStreamFilterImpl();

public:
    /** Set the configuration of the filter. This function store the xml configuration in the filter
    @param _szConfiguration Configuration string in XML format
    @return True if success
    */
    virtual bool SetConfiguration(LPCSTR _szConfiguration);

    /** Retrieve a string in the configuration string
    @param _szParam Name of the parameters to retrieve
    @param _szDefault Default value to return if the param is not present in the configuration
string
    @return Content of the parameters in the configuration string or the default value
    */
    EMString GetConfigurationString(LPCSTR _szParams, LPCSTR _szDefault = "");

    /** Retrieve a string in a folder of the configuration string
    @param _szFolder Name of the folder to find parameters to retrieve
    @param _szParam Name of the parameters to retrieve
    @param _szDefault Default value to return if the param is not present in the configuration
string
    @return Content of the parameters in the configuration string or the default value
    */
    EMString GetConfigurationStringInFolder(LPCSTR _szFolder, LPCSTR _szParams, LPCSTR _szDefault =
"");

    /** Retrieve a number in the configuration string
    @param _szParam Name of the parameters to retrieve
    @param _ulDefault Default value to return if the param is not present in the configuration
string
    @return Content of the parameters in the configuration string or the default value
    */
    ULONG GetConfigurationNumber(LPCSTR _szParams, ULONG _ulDefault = 0);

    /** Retrieve a number in a folder of the configuration string
    @param _szFolder Name of the folder to find parameters to retrieve
    @param _szParam Name of the parameters to retrieve
    @param _ulDefault Default value to return if the param is not present in the configuration
string
    @return Content of the parameters in the configuration string or the default value
    */
    ULONG GetConfigurationNumber(LPCSTR _szFolder, LPCSTR _szParams, ULONG _ulDefault = 0);

    /** Provide an output buffer for the specified length
        @param _ulGraphTimeStamp Time stamp of the data to provide
        @param _ulLength Length in ms of data to provide
        @param _pInputPin Pointer to the input pin that call the function (or NULL for the graph)
        @return Pointer to the buffer with data or NULL if no data to provide
        */
        virtual CEMStreamBufferImpl *OnProcess(ULONG _ulGraphTimeStamp, ULONG _ulLength,
CEMStreamPinInputImpl *_pInputPin);

    /** Mute or unmute the filter output
    @param _bEnable Optional flag that indicate the action. True to mute the filter output, False to
unmute the fllter output. The default value is True.
    */
    void Mute(bool _bEnable = true);

    /** Retrieve if the filter output is muted
    @return Flag that indicate if the filter output is muted. True if muted, False if not muted
    */
    bool IsMute();

    /** Create an input pin
        @return Pointer to the input pin created
        */
        virtual CEMStreamPinInputImpl *PinInputCreate();

        /** Destroy an input pin
        @param Pointer to the input pin to destroy or null if no more input pin is available
        */
        virtual void PinInputDestroy(CEMStreamPinInputImpl *_pInputPin);

        /** Retrieve input pin of the filter connected to the specified output filter
        @param _pOutputFilter Pointer to the output filter connected to this filter. NULL is allowed
if one one input pin exists.
        @return Pointer to the input pin connected to the specified output filter
        */
        CEMStreamPinInputImpl *GetConnectedPin(CEMStreamFilterImpl *_pOutputFilter = NULL);

        /** Start the filter
        @return One of the EEMStreamErrors.
        */
        virtual EEMStreamErrors Start();

        /** Stop the filter
        @return One of the EEMStreamErrors.
        */
        virtual EEMStreamErrors Stop();

        /** Pause the filter. State to be used as a lock on filter processing
        @return One of the EEMStreamErrors.
        */
        virtual EEMStreamErrors Pause();

        /** Resume the filter from Paused state
        @return One of the EEMStreamErrors.
        */
        virtual EEMStreamErrors Resume();

        /** Retrieve the current state of the filter
        @return State of the filter
        */
        EEMStreamFilterState GetState();

        /** Retrieve the type of the filter
        @return Type of the filter
        */
        EEMStreamFilterType GetType();

        /** Notify that an input pin is connected to the filter
        @param _pInputPin Pointer to the input pin that connected to the filter
        @return True if success.
        */
        virtual bool OnConnect(CEMStreamPinInputImpl *_pInputPin);

        /** Notify that an input pin is disconnected from the filter
        @param _pInputPin Pointer to the input pin that disconnected from the filter
        @return True if success.
        */
        virtual bool OnDisconnect(CEMStreamPinInputImpl *_pInputPin);

        /** Retrieve the name of the filter
        @return Name of the filter
        */
        virtual EMString GetName();

        /** Retrieve the pipe that contains the filter
        @return Pointer to the pipe that contains this filter
        */
        CEMStreamPipeImpl *GetPipe();
```

```
        /** Enable or disable the filter
        @param _bEnable True to enable the filter. False to disable it.
        */
        virtual void SetEnable(bool _bEnable);

        /** Retrieve if filter is enable or disable
        @return True if filter is enabled. False if it is disabled.
        */
        virtual bool IsEnable();

        /** Set the flag termination on this filter
        @param _bTermination True if the filter is a termination one. False if the filter is not a
termination one.
        */
        void SetTermination(bool _bTermination);

        /** Retrieve if the termination flag is set
        @return True if the termination flag is set.
        */
        bool IsTermination();

       /** Event received from a filter
       @param _pFilter Pointer to the filter that fire the event
       @param _Event One of the EEMStreamFilterEvent values specified which event is fired.
       @param _ulParam1 First parameter of the event depending of the event.
       @param _ulParam2 Second parameter of the event depending of the event.
       */
       virtual void OnEvent(CEMStreamFilterImpl *_pFilter, EEMStreamFilterEvent _Event, ULONG
_ulParam1 = 0, ULONG _ulParam2 = 0);

       /** Set the sampling rate of the filter
       @param _ulSamplingRate New sampling rate
       @return True if sampling rate changed, False if already set or failed
       */
       virtual bool SetSamplingRate(ULONG _ulSamplingRate);


       /** Set the frame size of the filter (if necessary). This is normally received from the
incoming buffer. So need a frame resize if this does not match !
       @param _ ulWidth New frame width
       @param _ ulHeight New frame height
       @return True if frame size changed, False if already set or failed
       */
       virtual bool SetFrameSize(ULONG, _ulWidth, ULONG _ulHeight);

protected:
       /// Configuration string of the filter
       EMString m_sConfiguration;
        /// Pointer to the graph that contains the filter
        CEMStreamGraphImpl *m_pGraph;
        /// Pointer to the pipe that contains the filter
        CEMStreamPipeImpl *m_pPipe;
        /// Type of the filter
        EEMStreamFilterType m_Type;

        /// Status of the filter
        EEMStreamFilterState m_State;
        /// Flag that indicate if the filter is enable
        bool m_bEnable;
        /// Flag that indicate if the filter is a termination one
        bool m_bTerminationFilter;
        /// Protection of processing
        CEMLock m_Lock;
       /// Sampling rate of working buffers
       ULONG m_ulSamplingRate;
       /// Length of last buffer for debug purpose
       ULONG m_ulLastDataLength;

       /// Mute state
       bool m_bMute;
```

```
        /// Debug state
        bool m_bDebug;
        /// Flag that indicates if silence are written in the debug file
        bool m_bDebugLogSilence;
        /// Debug file name
        EMString m_sDebugFile;
        /// Pointer to the debug file
        CEMFile *m_pDebugFile;
        /// Last graph time stamp written in the debug file
        ULONG m_ulDebugLastGraphTimeStamp;

        /** Enable or disable the debug mode on the input pin
        @param _bEnable Optional flag that indicate the action. True to enable the debug mode, False
to disable the debug mode. The default value is True.
        @param _sFilename Optional name of debug file. Use only if _bEnable is True. The default
value is empty.
        @param _bLogSilence Optional flag that indicate if silence are wirtten in the debug file.
Use only if _bEnable is True. The default value is True.
        */
        void Debug(bool _bEnable = true, EMString _sFilename = "", bool _bLogSilence = true);

        /** Retrieve if the filter is in debug mode
        @return Flag that indicate if the filter is in debug mode. True if debug mode is on, False
if debug mode is off.
        */
        bool IsDebug();

        /// Input Pin list type
        typedef deque<CEMStreamPinInputImpl *> listPinInput;
        /// Input Pin list
        listPinInput m_PinInputList;
        /// Maximum number of allowed input pins.
        ULONG m_ulPinInputMax;

        /// connection list type
        typedef deque<CEMStreamPinInputImpl *> listConnection;
        /// listConnection to the connected input pin
        listConnection m_ConnectedPins;

        /// delivered list type
        typedef map<CEMStreamPinInputImpl *, ULONG> listDelivered;
        /// Pointer to the delivered input pin
        listDelivered m_DeliveredPins;

        /** Notify the pipe of an event
        @param _Event One of the EEMStreamFilterEvent values specified which event is fired.
        @param _ulParam1 First parameter of the event depending of the event.
        @param _ulParam2 Second parameter of the event depending of the event.
        */
        void NotifyPipe(EEMStreamFilterEvent _Event, ULONG _ulParam1 = 0, ULONG _ulParam2 = 0);

        /** Provide an output buffer for the specified length
        @param _ulGraphTimeStamp Time stamp of the data to provide
        @param _ulLength Length in ms of data to provide
        @return Pointer to the buffer with data or NULL if no data to provide
        */
        virtual CEMStreamBufferImpl *OnProcess(ULONG _ulGraphTimeStamp, ULONG _ulLength);

        /** Call the OnProcess method and manage the debug mode
        @param _ulGraphTimeStamp Time stamp of the data to provide
        @param _ulLength Length in ms of data to provide
        @return Pointer to the buffer with data or NULL if no data to provide
        */
        virtual CEMStreamBufferImpl *Process(ULONG _ulGraphTimeStamp, ULONG _ulLength);

private:
        /// Pointer to the last buffer returned by OnProcess
        CEMStreamBufferImpl *m_pLastBuffer;
};

class CEMStreamFilterAudioImpl : public CEMStreamFilterImpl
```

```
{
protected:
        /** Constructor
        @param _Type Type of the filter
        @param _pGraph Pointer to the graph that contains the filter
        @param _pPipe Pointer to the pipe that contains the filter
        */
        CEMStreamFilterAudioImpl(EEMStreamFilterType _Type, CEMStreamGraphImpl *_pGraph,
CEMStreamPipeImpl *_pPipe);

        /** Destructor
        */
        virtual ~CEMStreamFilterAudioImpl();

        /** Retrieve the name of the filter
        @return Name of the filter
        */
        virtual EMString GetName();
};

#endif // EM_STREAMFILTERIMPL_H
```