# FxEngine Framework
# User guide

Release 1.20

**SMProcess**

## FxEngine contents

**FxEngine methods**

**FxEngine structures**

**FxEngine constants**

**Figures**

**Tables**

# 1. SOFTWARE LICENSE AGREEMENT

FxEngine Framework Copyright (C) 1999-2012 Sylvain Machel, owner of SMProcess and creator of this document.

Licenses for files are:

      FxEngine Framework library: GNU LGPL
      FxEngine Editor program:  GNU GPL
      TraceTool program:  GNU GPL

The GNU GPL and GNU LGPL restriction means that you must follow both GNU GPL and GNU LGPL rules restriction rules.

*Note:*

*You can use the FxEngine Framework on any computer, including a computer in a commercial organization.*

*You don't need to register or pay for FxEngine Framework.*

## 1.1. GNU LGPL INFORMATION

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library. if not,  If not, see <http://www.gnu.org/licenses/>.

## 1.2. GNU GPL INFORMATION

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

# 2. INTRODUCTION

The FxEngine is a Framework for data flow processing and the design of dynamic systems using plugins.

The Framework can be used in several application areas such:

> Signal Processing
> Image Processing
> Test and Measurement
> Financial Processing
> Control systems Design

This FxEngine Framework does not define a new standard for plugin architecture; it simply helps users to build them and to use them for their data flow processing.

## 2.1. CONCEPT

The FxEngine Framework is based on the C/C++ language, STL (Standard Template Library) and boost libraries (www.boost.org). It doesn't use COM, MFC or other proprietary components.

The FxEngine Framework provides two main components:

> A set of Programming Interfaces to build plugins,
> An Application Programming Interface (API) to use plugins in your application.

In the following sections, we use the FxEngine Framework terminology:

> Plugins are called FXs,
> The Fx manager, which allows to control FXs, is called FxEngine.

### 2.1.1. FX

FXs are composed of Input Pins, Output Pins and Fx Input/Output parameters. These components are showed in the following figure.
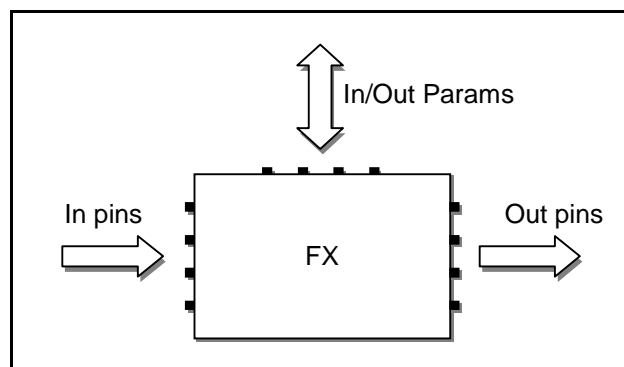


Figure 2-1 Fx components

FXs receive input data, perform a task and produce output data.

An Output Pin can be connected to **several** Input Pins.

Each input pin receives a flow of media data from the previous Fx and each output pin delivers a flow data to next Fx.

Media data are put inside a Media buffer called FxMedia (see IFxMedia interface).

In/Out parameters are used to publish the Fx's parameters. Fx's parameters could be read and/or modified by the application.

The FxEngine Framework allows to the user to define the number of input/output pins, their types and to define or not the Fx parameters.

## 2.1.2. FXENGINE ARCHITECTURE

In the FxEngine Framework, the FxEngine architecture is a chain of FXs. The output from one Fx becomes the input for another. The following figure shows an example of Fx architecture:
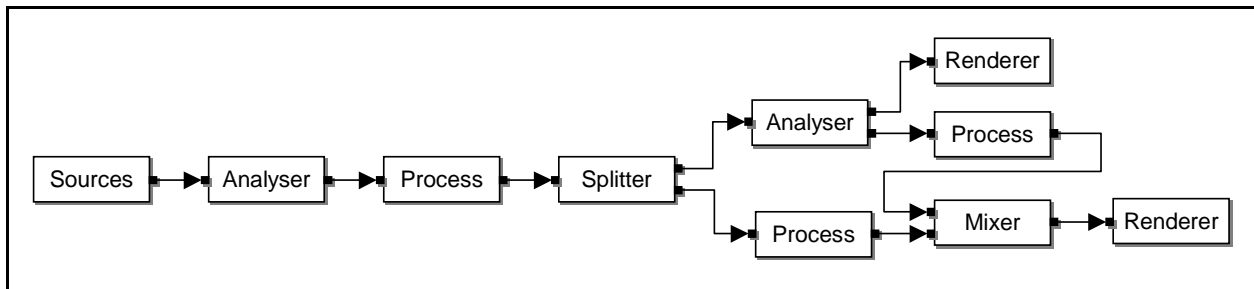


Figure 2-2 FxEngine architecture

As we see in the previous figure, FXs can be grouped into several types:

Fx **Source** allows to introduce data in the architecture from file, network or anywhere else.
Fx **Analyser** allows to analyze data.
Fx **Process** allows to perform a user task on an input data. Video and Audio CODECs are examples of Fx Process.
Fx **Splitter** splits an input data into two or more outputs. Audio channels separation is an example of Fx splitter.
Fx **Mixer** allows mixing multiple inputs and producing a single output.
Fx **Renderer** allows to present data to the user. Fx Renderer might use audio sound card, video display or hard disk to write file.
Fx **User** allows to the user to specify its own Fx type.

The Fx types are used to group FXs but there are not absolute rules, in this way, an Fx analyzer could be also used to process or to split data.

## 2.1.3. APPLICATIONS

The FxEngine is used to manage FXs. Applications use the FxEngine API to perform any task by connecting chains of FXs together.

Applications use the following steps to build plugin architecture:

The application creates an instance of the FxEngine (via the FxEngine API).
The application uses the FxEngine API to add and to connect FXs.
The application uses the FxEngine API to control any FXs, to get Fx Interfaces.
The application has to hang Fx states and Fx notes.

The following figure shows an example of application:
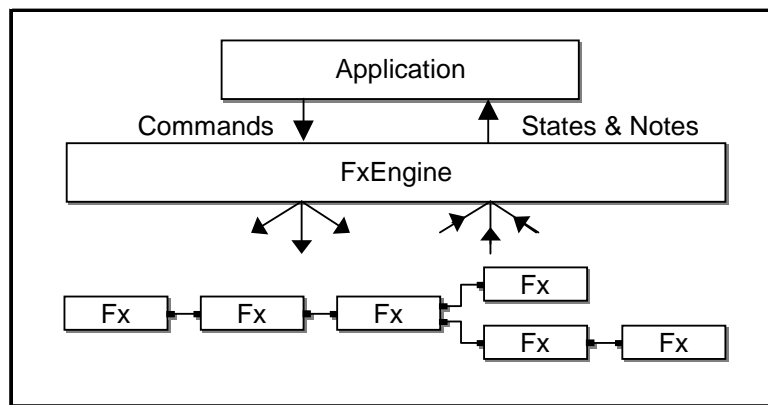
Figure 2-3 Example of simple application

Applications can use several FxEngine instances allowing to build more sophisticated architectures. The following figure shows an example:
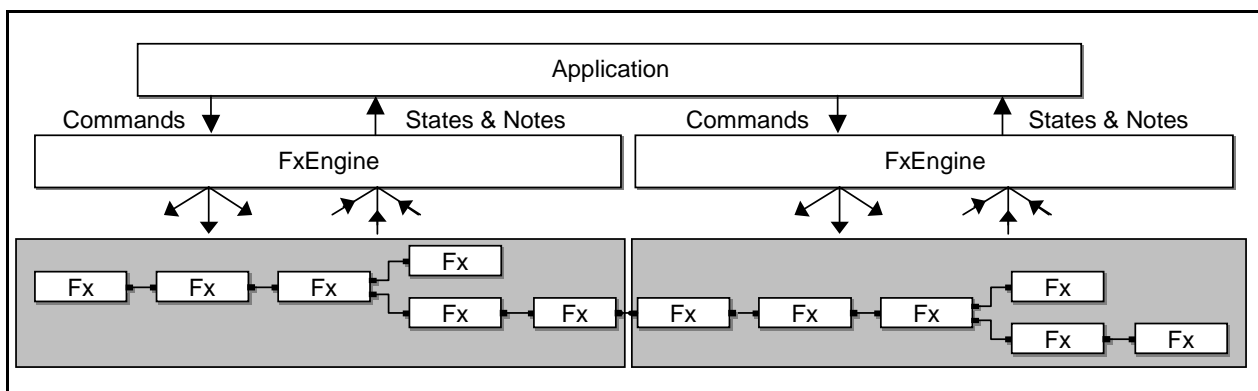


Figure 2-4 Example of sophisticated application

## 2.1.4.  FX STATES

### 2.1.4.1.  Fx running states

Fx running states depend on the FxEngine events START / PAUSE / STOP. Those events try respectively to start, to pause and to stop any Fx. The Fx running states are defined in FX_RUNNING_STATE.
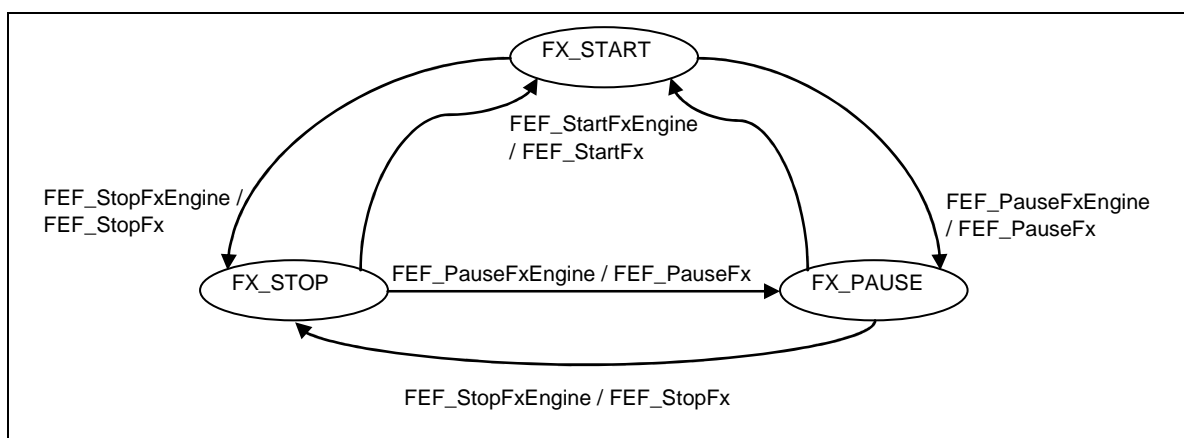
### 2.1.4.2.  Fx internal states

Fx internal States are some constants values which are used to inform application or any other observers. Fxs are in charge to send the right states through the interface IFxState using the IFxState::FxPublishState method. All pre-defined Fx states are contained into FX_STATE.

### 2.1.4.3.  Fx Notes

Fx notes are some message (charactere chain) which are sent to the application host or any other observers. Fxs can send any sort of messages using the interface IFxNote using the IFxNote::FxPublishState method.

### 2.1.5.  SUB FXENGINE

Each Fx can contain a Sub FxEngine with several FXs inside.

## 2.2.  SUPPORTED MEDIA

The FxEngine Framework supports several media types. Each Fx uses formatted media object to manage data. The pin exposes media types, which are described by the FX_MEDIA_TYPE structure.

The following table shows the supported medias.

| Main media type | Sub media type | Description |
|---|---|---|
| Audio | PCM | Generic Audio pulse code modulation (see IFxPcmFormat) |
| | PCMU | G.711 u-Law. The PCMU audio codec is described in rfc 3551 |
| | PCMA | G.711 a-Law. The PCMA audio codec is described in RFC 3551 |
| | G721, G726 | ADPCM - Subsumed by G.726. The packetization of the G721 audio codec is described in RFC 3551 |
| | G722 | G.722. The packetization of the G722 audio codec is described in RFC 3551 |
| | G723 | G.723.1 at 6.3kbps or 5.3 kbps. The packetization of the G723 audio codec is described in RFC 3551 |
| | G728 | G.728 16kbps CELP. The packetization of the G728 audio codec is described in RFC 3551 |
| | G729 | G.729 8kbps. The packetization of the G729 audio codec is described in RFC 3551 |
| | GSM | GSM 06.10. The packetization of the GSM audio codec is described in RFC 3551 |
| | LPC | LPC-10 Linear Predictive CELP, The packetization of the LPC audio codec is described in RFC 3551 |
| | QCELP | The Electronic Industries Association (EIA) & Telecommunications Industry Association (TIA) standard IS-733. The packetization of the QCELP audio codec is described in RFC 2658 |
| | DVI4 8k | DVI4 at 8kHz sample rate. The packetization of the DVI_8K audio codec is described in RFC 3551 |
| | DVI4 11k | DVI4 at 11kHz sample rate. The packetization of the DVI_11K audio codec is described in RFC 3551 |
| | DVI4 16k | DVI4 at 16kHz sample rate. The packetization of the DVI4_16K audio codec is described in RFC 3551 |
| | DVI4 22k | DVI4 at 22kHz sample rate. The packetization of the DVI4_22K audio codec is described in RFC 3551 |
| | MPEGA | MPEGA denotes the ISO standard packet of MPEG-1 or MPEG-2 audio. The encoding is defined in ISO standards ISO/IEC 11172-3 and 13818-3 |
| | MPA | MPA denotes MPEG-1 or MPEG-2 audio encapsulated as elementary streams.  The encoding is defined in ISO standards ISO/IEC 11172-3 and 13818-3.  The encapsulation is specified in RFC 2250 |
| | DOLBY AC3 | Dolby® AC3 audio |

| Main media type | Sub media type | Description |
|---|---|---|
| | AAC | MPEG-2 AAC audio standard. The encoding is defined in ISO standards ISO/IEC 13818-7 |
| | WMA | Windows® Media Audio standard packet |
| Video / Image | MPV | MPV designates the use of MPEG-1 and MPEG-2 video encoding elementary streams as specified in ISO Standards ISO/IEC 11172 and 13818-2, respectively.  The RTP payload format is as specified in RFC 2250 |
| | CELB | The CELL-B encoding is a proprietary encoding proposed by Sun Microsystems. The byte stream format is described in RFC 2029 |
| | JPEG | The encoding is specified in ISO Standards 10918-1 and 10918-2. The RTP payload format is as specified in RFC 2435 |
| | BGR | 24 bits RGB Format. 8 bits Blue / 8 bits Green / 8bits Red |
| | RGB | 24 bits RGB Format. 8 bits Red / 8 bits Green / 8bits Blue |
| | R_COLOR | 8 bits Red only |
| | G_COLOR | 8 bits Green only |
| | B_COLOR | 8 bits Blue only |
| | UYVY | 16 bits YUV Format (packed 4:2:2). Each macropixel is 4 bytes and contains 2 pixels (U0 Y0 V0 Y1) |
| | YUY2 | 16 bits YUV Format (packed 4:2:2). Each macropixel is 4 bytes and contains 2 pixels (Y0 U0 Y1 V0) |
| | IYUV | 12 bits YUV Format. 8-bpp Y plane, followed by 8-bpp 2×2 U and V planes |
| | YV12 | 12 bits YUV Format. YV12 is identical to IYUV but the order of the U and V planes is switched, so the V plane comes before the U plane |
| | NV12 | 12 bits YUV Format. 8-bpp Y plane, followed by 8-bpp 2×2 interlaced U and V planes |
| | NV21 | 12 bits YUV Format. 8-bpp Y plane, followed by 8-bpp 2×2 interlaced V and U planes |
| | Y800 | 8 bits YUV Format |
| | H261 | The encoding is specified in ITU-T Recommendation H.261, "Video codec for audiovisual services at p x 64 kbit/s". The packetization and RTP-specific properties are described in RFC 2032 |
| | H263 | The encoding is specified in the 1996 version of ITU-T Recommendation H.263, "Video coding for low bit rate communication". The packetization and RTP-specific properties are described in RFC 2190 |
| | H263_1998 | The encoding is specified in the 1998 version of ITU-T RecommendationH.263, "Video coding for low bit rate communication". The packetization and RTP-specific properties are described in RFC 2429 |
| | MP2T | MP2T designates the use of MPEG-2 transport streams, for either audio or video. The RTP payload format is described in RFC 2250 |
| | WMV | Windows® Media Video standard |
| | DIVX | DIVX codec packet |
| | XVID | XVID codec packet |
| Text | UTF_8 | Unicode UTF-8 encoding represents Unicode characters as sequences of 8-bit integers |
| | UTF_16 | Unicode UTF-16 encoding represents Unicode characters as sequences of 16-bit integers |
| | ASCII | ASCII encoding represents the Latin alphabet as single 7-bit ASCII characters |
| Data | Vector | Generic Vector format (see IFxVectorFormat) |
| | Matrix | Generic Matrix format (see IFxMatrixFormat) |
| User | User define | User define |

Table 1 FxEngine Framework media types

**Please contact SMProcess for additional media types and sub-types.**

Main Media types and sub-types are defined in FxMediaTypes.h file.

RFCs can be found on www.rfc.net.

## 2.3. FX CONNECTION

Fx pins are connected through the FxEngine API. Output pins can be connected to several input pins.

There is no Fx pin order connection, in other words, you can connect output pin before input pin or the inverse.

There is no Fx order connection, in other words, you can begin to connect any FXs in chain.

Each pin can be disconnected and reconnected in any Fx states.

Because a pin can accept several media types, you have to create pin with a preferred media types order. The connection negotiation task between two FXs gets the first media type of the input pin and checks that one of the output pin media types is acceptable. If not, the FxEngine tries with the next input pin media types.

In case where an output pin is already connected, a new connection will try to connect with the media type of the previous connection only.

During the pins connection, both pin callbacks are called to negotiate the media buffer properties (size and number). See IFxPinCallback::FxMedia method.

When the connection is established, the Fx IFxPinCallback::FxPinState callback method is called.

## 2.4. FX SYNCHRONISATION

The FxEngine Framework provides two ways to synchronize FXs:
Using a master clock signal,
Using a request media signal.

The first way is to create a master clock using the IFxRefClock interface or API clock functions. Application and FXs can create several clocks with a unique identifier. All clocks can be retrieved by any FXs and applications by this ID.

The second way is to use the IFxPin::WaitForIFxMedia to request media data from the previous FXs in chain. For example, Fx Renderer, which uses the audio sound card clock, might use this method to receive new media data from the previous Fx Source(s) in chain (see IFxPinCallback interface).

## 2.5. BUILD ENVIRONMENT

The FxEngine Framework uses the C/C++ language. To build FXs or applications you need to use Microsoft® Visual C++®, and Linux IDE (e.g. Code::Block, KDevelop) or Makefile.

### 2.5.1. HEADER FILES

In the FxEngine Framework directory installation, you will find the Includes directory that contains all header files (.h) to build FXs and applications.

The following table shows the header files.

| Header File | Description | Required for |
|---|---|---|
| FxDef.h | This file includes the main definitions of the FxEngine Framework. | Required for all FXs and applications. |
| FxErr.h | This file includes the main FxEngine Framework errors. | All FXs and applications if needed. |

| Header File | Description | Required for |
|---|---|---|
| FxTypes.h | This file contains the main types used in the FxEngine FrameWork. | Required for all FXs and applications. |
| FxMediaTypes.h | This file contains the Main and Sub Media Types | Required for all FXs and applications if needed. |
| FxEngine.h | This file contains the definitions of the FxEngine FrameWork API | Required for applications. |
| IFxBase.h | This file defines the Fx export interface. Every Fx plugin have to inherit from it | Required for all FXs. |
| IFx.h | This is the main interface for the FXs. | Required for all FXs. |
| IFxState.h | Fx state interface. | Recommended for all FXs. |
| IFxNote.h | Fx note interface. | Recommended for all FXs. |
| IFxPinManager.h | Pins Manager interface.<br>This file contains the definitions and methods to manage Fx's pins. | Required for all FXs. |
| IFxPin.h | Pins interface.<br>This file contains the definitions and methods of Fx's pins. | Required for all FXs.<br>Required for all applications if needed. |
| IFxPinCallback.h | Pins callback interface.<br>This file contains the callback definitions for pins. | Required for all FXs. |
| IFxParam.h | Parameters interface.<br>This file contains the definitions and methods to control Fx's parameters. | Required for all FXs and applications if needed. |
| IFxRefClock.h | Reference clock interface.<br>This file contains the definitions and methods to control FxEngine Framework clock. | Required for all FXs and applications if needed. |
| IFxMedia.h | Media interface.<br>This file contains the definitions and methods to control Fx's Media. | Required for all FXs.<br>Required for all applications if needed. |
| IFxPcmFormat.h | PCM media interface.<br>This file contains the definitions and methods to manage audio pcm Media format. | Required for all FXs if needed.<br>Required for all applications if needed. |
| IFxVideoImgFormat.h | Video media interface.<br>This file contains the definitions and methods to manage video Media format. | Required for all FXs if needed.<br>Required for all applications if needed. |
| IFxVectorFormat.h | Vector media interface.<br>This file contains the definitions and methods to manage vector Media format. | Required for all FXs if needed.<br>Required for all applications if needed. |

| Header File | Description | Required for |
|---|---|---|
| IFxMatrixFormat.h | Matrix media interface.<br>This file contains the definitions and methods to manage matrix Media format. | Required for all FXs if needed.<br>Required for all applications if needed. |

Table 2 FxEngine Framework header files

### 2.5.2. LIBRARY FILES

In the FxEngine Framework directory installation, you will find the Lib and Bin directories containing all static and dynamic libraries (.so, .lib and .dll).

The following table shows the Windows library files where **X** represents the windows Run-time Libraries version (e.g. 7, 8, 9).

| Library File | Description | Required for |
|---|---|---|
| FxEngine-Vc**X**-md.lib | The FxEngine static library using the multithread- and DLL-specific versions of the run-time routines (MSVCRT.lib).<br>This library needs MSVCR**X**0.DLL and MSVCP**X**0.DLL. | Required for all applications linking.<br>Required for all FXs linking. |
| FxEngined-Vc**X**-md.lib | The FxEngine static library using the multithread- and DLL-specific versions of the run-time routines (MSVCRTD.lib).<br>This library needs MSVCR**X**0D.DLL and MSVCP**X**0D.DLL. | Required for all applications **debug** linking.<br>Required for all FXs **debug** linking. |
| FxEngine-Vc**X**-md.dll | The FxEngine dynamic library using FxEngined-Vc**X**-md.lib. | Required for all applications running. |
| FxEngined-Vc**X**-md.dll | The FxEngine dynamic library using FxEngined-Vc**X**-md.lib. | Required for all applications running (**debug only**). |

Table 3 FxEngine Framework Windows library files

The following table shows the Linux library file.

| Library File | Description | Required for |
|---|---|---|
| FxEngine.XX.XX.so | The FxEngine dynamic library. | Required for all applications linking.<br>Required for all FXs linking. |

Table 4 FxEngine Framework Linux library file

### 2.5.3. BUILDING FX

Fx building requires the "**FEF**" namespace to use the FxEngine Framework. Add "using namespace FEF;" or use "FEF::" in your code.

### 2.5.3.1.   Windows

To build Fx, perform the following steps:

> Create a new empty dynamic link library,
> Include the header file *IFxBase.h*,
> Add the following to the DLL .def file in the exports section:
> ```
>     FEF_GetFxBase        PRIVATE
> ```
> Set the Run-Time Library to Multi-threaded DLL (MD),
> Set the Calling Convention to __cdecl.
> Link to the Fx run-time library (FxEngine-Vc**X**-md.lib or FxEngined-Vc**X**-md.lib to debug).
> Inherit your Fx from the IFxBase interface and overwrite pure virtual methods,
> Write the **FEF_GetFxBase()** method.
> Use the Fx Interfaces to build your Fx.

FXs require the FxEngine binaries (FxEngine-Vc**X**-md.dll or FxEngined-Vc**X**-md.dll) to run properly. You may distribute these files in your Fx package.

### 2.5.3.2.   Linux

To build Fx, perform the following steps:

> Create a new C++ class file,
> Include the header file *IFxBase.h*,
> Inherit your Fx from the IFxBase interface and overwrite pure virtual methods,
> Write the **FEF_GetFxBase()** method.
> Use the Fx Interfaces to build your Fx.
> Link to the Fx run-time library (FxEngine.XX.XX.so) in your make file.

FXs require the FxEngine library (FxEngine.XX.XX.so) to run properly. You may distribute this file in your Fx package.

### 2.5.4.   BUILDING APPLICATION

Application building requires the "**FEF"** namespace to use the FxEngine Framework. Add "using namespace FEF;" or use "FEF::" in your code.

### 2.5.4.1.   Windows

To build application, perform the following steps:

> Create a new application,
> Include the header file FxEngine.h ,
> Link to the run-time library (FxEngine-Vc**X**-md.lib or FxEngined-Vc**X**-md.lib to debug).
> Use the FxEngine API.

Applications require the FxEngine binaries (FxEngine-Vc**X**-md.dll or FxEngined-Vc**X**-md.dll) to run properly. You may distribute these files in your application package.

### 2.5.4.2.   Linux

To build application, perform the following steps:

> Create a new application,

Include the header file FxEngine.h ,
Link to the run-time library (FxEngine.XX.XX.so).
Use the FxEngine API.

Applications require the FxEngine binaries (FxEngine.XX.XX.so) to run properly. You may distribute this file in your application package.

# 3. FX INTERFACES

The following sections describe the Fx interfaces.

## 3.1. IFXBASE

The IFxBase interface is the Fx DLL interface. Each Fx has to inherit from it. User has to write the IFxBase methods.
The IFxBase methods are called by the FxEngine API to control the FXs.

**FEF_GetFxBase**

This function exports the IFxBase interface of the Fx from your DLL.

**Syntax:**

FXENGINE_EXP IFxBase* FXENGINE_API FEF_GetFxBase ();

**Parameters:**

None.

**Return Values:**

The IFxBase interface of the Fx instance.

**Remarks:**

None.

**Requirements:**

Header: IFxBase.h.

**IFxBase::GetFxInfo**

This method is called to get the main definitions of the Fx.

**Syntax:**

virtual Int32 GetFxInfo(
    const FX_DESCRIPTOR** ppFxDescriptor
)PURE;

**Parameters:**

*ppFxDescriptor*
        [out] Address of a variable that receives a pointer to the Fx descriptor structure (see
        FX_DESCRIPTOR structure).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxBase.h.


`IFxBase::InitFx`

The InitFx method is called to initialize the Fx. This method has to create an IFx instance, defines Fx pins (In or/and Out), Fx parameters and all other Fx features.

**Syntax:**

```
virtual Int32 InitFx(
    IFx **ppFx
)PURE;
```

**Parameters:**

*ppFx*
> [out] Pointer to a variable that receives the address of the IFx interface.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX  error code.

**Remarks:**

This method can't be a worker thread !!.

At the end of this method, it's recommended to set the Fx state to FX_INIT_STATE (see IFxState::FxPublishState method).

**Requirements:**

Header: IFxBase.h.


`IFxBase::DisplayFxPropertyPage`

This method is called to show the Fx property page.

**Syntax:**

```
virtual Int32 DisplayFxPropertyPage(
    Pvoid pvWndParent
);
```

**Parameters:**

*pvWndParent*
> [in] Handle to the parent window (Can be null).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX  error code.

**Remarks:**

This method is optional.

**Requirements:**

Header: IFxBase.h.


`IFxBase::GetFxFrame`

This method is called to return the current Fx Frame. Fx frame is a XPM image format (see http://koala.ilog.fr/lehors/xpm.html) and allows to a Framework front-end to render the Fx with a picture.

Fx can update at any moment its frame and informs Fx observer with the FX_FRAME_UPDATE state sending.

**Syntax:**

```
virtual Int32 GetFxFrame(
    const Char** ppbFxFrame
);
```

**Parameters:**

*ppbFxFrame*
        [out] Address of a variable that receives the XPM data (Can be null).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX  error code.

**Remarks:**

This method is optional.

**Requirements:**

Header: IFxBase.h.


`IFxBase::UpdateFxParam`

The UpdateFxParam is called to update the public parameters. It means that at least one Fx parameter is been modified by an application and that  the FEF_UpdateFxParam function was called.

**Syntax:**

```
Virtual Int32 UpdateFxParam (
    const std::string strParamName,
    FX_PARAMETER FxParameter
);
```

**Parameters:**

*strParamName*
        [in] Variable that contains the parameter name.
*FxParameter*
        [in] Variable that contains the updating mode.
**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX  error code.

**Remarks:**

This method is optional, see the FEF_UpdateFxParam function.

**Requirements:**

Header: IFxBase.h.

`IFxBase::GetFxUserInterface`

The GetFxUserInterface method is called to get an user interface if it exists.

**Syntax:**

```
virtual Int32 GetFxUserInterface(
    Pvoid* ppvUserInterface
);
```

**Parameters:**

*ppvUserInterface*
      [out] Address of a variable that receives a pointer to the user interface.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX  error code.

**Remarks:**

See the FxEngine API GetFxUserInterface function.

This method is optional.

**Requirements:**

Header: IFxBase.h.

---

`IFxBase::GetFxSubFxEngine`

The GetFxSubFxEngine is called to get a FxEngine handle if it exists.

Fx can contain a sub FxEngine system with several FXs.

**Syntax:**

```
virtual Int32 GetFxSubFxEngine(
    FX_HANDLE* phFxEngine
);
```

**Parameters:**

*phFxEngine*
      [out] Pointer to a variable that receives the FxEngine handle.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX  error code.

**Remarks:**

See the FxEngine API GetFxSubFxEngine function.

This method is optional.

**Requirements:**

Header: IFxBase.h.

---

`IFxBase::StartFx`

This method is called to start the Fx.

**Syntax:**

```
virtual Int32 StartFx(
)PURE;
```

**Parameters:**

None.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX  error code.

**Remarks:**

At the end of this method, it's recommended to set the Fx state to FX_START_STATE (see IFxState::FxPublishState method).

**Requirements:**

Header: IFxBase.h.

**IFxBase::RunFx**

This method is called to run the Fx.

**Syntax:**

virtual Int32 RunFx(
);

**Parameters:**

None.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

At the end of this method, it's recommended to set the Fx state to FX_RUN_STATE (see IFxState::FxPublishState method).

**Requirements:**

Header: IFxBase.h.

**IFxBase::PauseFx**

This method is called to pause the Fx.

**Syntax:**

virtual Int32 PauseFx(
);

**Parameters:**

None.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

This method is optional.

At the end of this method, it's recommended to set the Fx state to FX_PAUSE_STATE (see IFxState::FxPublishState method).

**Requirements:**

Header: IFxBase.h.

**IFxBase::StopFx**

This method is called to stop the Fx.

**Syntax:**

virtual Int32 StopFx(

```
)PURE;
```

**Parameters:**

None.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX  error code.

**Remarks:**

At the end of this method, it's recommended to set the Fx state to FX_STOP_STATE (see IFxState::FxPublishState method).

**Requirements:**

Header: IFxBase.h.

---

`IFxBase::ReleaseFx`

This method is called to release the Fx. This method has to release all Fx components (interfaces and internal values) and probably the current IFxBase instance..

**Syntax:**

```
virtual Int32 ReleaseFx(
)PURE;
```

**Parameters:**

None.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX  error code.

**Remarks:**

At the end of this method, it's recommended to set the Fx state to FX_RELEASE_STATE (see IFxState::FxPublishState method).

**Requirements:**

Header: IFxBase.h.

## 3.2. IFX

The IFx is the main Fx interface, which contains methods to create and manage Fx.

---

`FEF_CreateFx`

The FEF_CreateFx method obtains an IFx interface.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_CreateFx(
    IFx ** ppFx,
    const std::string strFxName,
);
```

**Parameters:**

*ppFx*

[out] Address of a variable that receives a pointer to the IFx interface created.

*strFxName*
[in] Variable that contains the Fx short name.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX  error code.

**Remarks:**

Call the IFxState::Release method to release the IFx interface.

**Requirements:**

Header: IFx.h.

`IFx::GetIFxVersion`

The GetIFxVersion method gets the IFx interface version.

**Syntax:**

```
virtual Int32 GetIFxVersion(
    Uint16* pwMajor,
    Uint16* pwMinor,
    Uint16* pwBuild,
    Uint16* pwRev
)PURE;
```

**Parameters:**

*pwMajor*
[out] Pointer to a variable that receives the Major of IFx version.
*pwMinor*
[out] Pointer to a variable that receives the Minor of IFx version.
*pwBuild*
[out] Pointer to a variable that receives the Build of IFx version.
*pwRev*
[out] Pointer to a variable that receives the Revision of IFx version.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX  error code.

**Remarks:**

None.

**Requirements:**

Header: IFx.h.

`IFx::FxGetInterface`

The FxGetInterface method retrieves a specific interface of Fx. This method increases the interface reference count by 1.

**Syntax:**

```
virtual Int32 FxGetInterface(
    FX_INTERFACE FxInterfaceType,
    Void** ppFxInterface
)PURE;
```

**Parameters:**

*FxInterfaceType*
> [in] Variable that contains the Fx Interface to get (see FX_INTERFACE).

*ppFxInterface*
> [out] Address of a variable that receives a pointer to the interface.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

Call the ReleaseInterface method to release the interface.

**Requirements:**

Header: IFx.h.

---

**IFx::Release**

The Release method releases the IFx interface returned by the FEF_CreateFx function.

**Syntax:**

```
virtual Int32 Release(
)PURE;
```

**Parameters:**

None.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

Release all other interfaces (returned by the FxGetInterface method) before calling it.

**Requirements:**

Header: IFx.h.

---

**IFx::GetConstToString**

The GetConstToString method converts a FxEngine constant to a string.

**Syntax:**

```
virtual Int32 GetConstToString (
    FXENGINE_CONST_TYPE FxEngineConstType,
    Int32 sdwFxEngineConst,
    Std::string& strStateName
)PURE;
```

**Parameters:**

*FxEngineConstType*
> [in] Variable that contains the Type of the constant (see FXENGINE_CONST_TYPE).

*sdwFxEngineConst*
> [in] Variable that contains the constant to convert.

*strStateName*
> [out] Reference to a variable that receives the constant name.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFx.h.

## 3.3. IFXSTATE

The IFxState interface allows to publish the Fx states to observers (see AttachFxObserver, AttachFxObserverEx and DetachFxObserver from the FXEngine API).

Use the IFx::FxGetInterface method with IFX_STATE parameter to retrieve IFxState.

`IFxState::FxPublishState`

The FxPublishState method publishes a Fx state to the observers.

**Syntax:**

```
virtual Int32 FxPublishState(
    FX_STATE FxState
)PURE;
```

**Parameters:**

*FxState*
    [in]:Variable that contains the Fx State to publish.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

See the FX_STATE constants.

**Requirements:**

Header: IFxState.h.

`IFxState::FxRePublishState`

The FxRePublishState method re-publishes the latest Fx state to the observers.

**Syntax:**

```
    virtual Int32 FxRePublishState()PURE;
```

**Parameters:**

*None*

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxState.h.

**IFxState::FxGetState**

The FxGetState method retrieves the latest Fx state which has been sended to the observers.

**Syntax:**

```
virtual Int32 FxGetState(
    FX_STATE *pFxState
)PURE;
```

**Parameters:**

*pFxState*
>  [out]: Pointer on a variable that receives the Fx State.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

See the FX_STATE constants.

**Requirements:**

Header: IFxState.h.

**IFxState::FxReleaseInterface**

The FxReleaseInterface method releases the IFxState interface. This method decreases the reference count by 1.

**Syntax:**

```
virtual Int32 FxReleaseInterface(
)PURE;
```

**Parameters:**

None.

**Return Values:**

The new reference count.

**Remarks:**

See IFx::FxGetInterface method.

**Requirements:**

Header: IFxState.h.

## 3.4. IFXNOTE

The IFxNote interface allows to publish the "Notes" or "Messages" (Character chain) from Fx to any observers (see AttachFxObserver, AttachFxObserverEx and DetachFxObserver from the FXEngine API).

Use the IFx::FxGetInterface method with IFX_NOTE parameter to retrieve IFxNote.

**IFxNote::FxPublishNote**

The FxPublishNote method publishes "Notes" or "Messages" to the attached observers.

**Syntax:**

```
virtual Int32 FxPublishNote(
    const std::string strNote
```

```
)PURE;
```

**Parameters:**

*strNote*

> [in]:Variable that contains the note to publish.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Requirements:**

Header: IFxNote.h.

---

`IFxNote::FxReleaseInterface`

The FxReleaseInterface method releases the IFxNote interface. This method decreases the reference count by 1.

**Syntax:**

```
virtual Int32 FxReleaseInterface(
)PURE;
```

**Parameters:**

None.

**Return Values:**

The new reference count.

**Remarks:**

See IFx::FxGetInterface method.

**Requirements:**

Header: IFxNote.h.

## 3.5. IFXPINMANAGER

The IFxPinManager interface contains methods to create and to remove Fx pins. Use the IFx::FxGetInterface method with IFX_ PINMANGER parameter to retrieve IFxPinManager.

`IFxPinManager::Create`

The Create method creates a new Fx pin.

Fx can create at any moment a new pin. Fx has to inform Fx observers with the FX_PIN_UPDATE state sending.

All Fx pins are automaticly removed after the calling of IFxBase::Release method.

**Syntax:**

```
virtual Int32 Create(
    PFX_PIN pPinInfo,
    IFxPin** ppFxPin
)PURE;
```

**Parameters:**

*pPinInfo*

> [in] Pointer to a variable that contains the FX_PIN structure (see FX_PIN).

*ppFxPin*
>    [out] Address of a variable that receives a pointer to an IFxPin interface.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

You can add and connect Fx's pins at any time. In this case, it can be useful to publish the FX_PIN_UPDATE state.

**Requirements:**

Header: IFxPinManager.h.


`IFxPinManager::Remove`

The Remove method removes an existing Fx pin.

Fx can remove at any moment a pin. Fx has to inform Fx observers with the FX_PIN_UPDATE state sending.

**Syntax:**

```
virtual Int32 Remove(
    IFxPin* pFxPin
)PURE;
```

**Parameters:**

*pFxPin*
>    [in] Variable that contains the IFxPin interface of pin to release.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The given pin is automatically disconnected.

**Requirements:**

Header: IFxPinManager.h.


`IFxPinManager::FxReleaseInterface`

The FxReleaseInterface method releases the IFxPinManager interface. This method decreases the reference count by 1.

**Syntax:**

```
virtual Int32 FxReleaseInterface(
)PURE;
```

**Parameters:**

None.

**Return Values:**

The new reference count.

**Remarks:**

See IFx::FxGetInterface method.

**Requirements:**

Header: IFxPinManager.h.

## 3.6. IFXPIN

The IFxPin interface contains methods to control the Fx pins. Fx can contain one or several pin of any media types.

Use the IFxPinManager::Create method to create pin and retrieve IFxPin.

Use the QueryFxPinInterface function to retrieve IFxPinManager from the application.

All IFxPin methods may be used on In/Out pins (exept the **IFxPin::GetProcessTime**).

`IFxPin::GetPinName`

The GetPinName method gets the Fx pin name.

**Syntax:**

```
virtual Int32 GetPinName(
    std::string& strPinName
)PURE;
```

**Parameters:**

*strPinName*
        [out] Reference to a variable that receives the pin name.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxPin.h.

`IFxPin::GetPinType`

The GetPinType method gets the Fx pin type.

**Syntax:**

```
virtual Int32 GetPinType(
    FX_PIN_TYPE* pPinType
)PURE;
```

**Parameters:**

*pPinType*
        [out] Pointer to a variable that receives the pin type (see FX_PIN_TYPE).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxPin.h.


### IFxPin::GetPinState

The GetPinState method gets the Fx pin state.

**Syntax:**

```
virtual Int32 GetPinState(
    FX_PIN_STATE* pPinState
)PURE;
```

**Parameters:**

*pPinState*
> [out] Pointer to a variable that receives the pin type (see FX_PIN_STATE).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The pin state is also sent to the IFxPinCallback::FxPinState method.

**Requirements:**

Header: IFxPin.h.


### IFxPin::GetTxRxBytes

The GetTxRxBytes method gets the amount of bytes received or transmitted since the Fx pin is connected.

**Syntax:**

```
virtual Int32 GetTxRxBytes(
    Uint64* pqTxRxByte
)PURE;
```

**Parameters:**

*pqTxRxByte*
> [out] Pointer to a variable that receives the amount of bytes.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The amount of bytes is reset (i.e. 0) when pin is disconnected.

**Requirements:**

Header: IFxPin.h.


### IFxPin::GetMediaTypeCount

The GetMediaTypeCount method gets the Pin MediaType count.

**Syntax:**

```
virtual Int32 GetMediaTypeCount(
    Uint16* pwMediaTypeCount
```

)PURE;

**Parameters:**

*pwMediaTypeCount*
        [out] Pointer to a variable that receives the Pin MediaType count.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxPin.h.

---

`IFxPin::GetMediaType`

The GetMediaType method gets the Pin MediaType by its index.

**Syntax:**

```
virtual Int32 GetMediaType(
    PFX_MEDIA_TYPE pMediaType,
    Uint16 wMediaTypeIndex
)PURE;
```

**Parameters:**

*pMediaType*
        [out] Pointer on a FX_MEDIA_TYPE variable that receives media type.
*wMediaTypeIndex*
        [in] Index of the Pin MediaType to get. From 0 to N-1. N is given by the GetMediaTypeCount
        function.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxPin.h.

---

`IFxPin::GetConnectedPinNumber`

The GetConnectedPinNumber method returns the number of connected pin to it.

**Syntax:**

```
virtual Int32 GetConnectedPinNumber(
    Uint32 *pdwPinNumber
)PURE;
```

**Parameters:**

*pdwPinNumber*
        [out] Address of a variable that receives the number of connected pin.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

On a call on an input pin, this method always returns a number equal to 1.

**Requirements:**

Header: IFxPin.h.

**IFxPin::GetPinConnected**

The GetPinConnected method gets the Fx pin, which is connected to it.

**Syntax:**

```
virtual Int32 GetPinConnected(
    IFxPin** ppFxPin
)PURE;
```

**Parameters:**

*ppFxPin*
> [out] Address of a variable that receives a pointer to an IFxPin interface.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxPin.h.

**IFxPin::GetNextPinConnected**

The GetNextPinConnected method gets the next Fx pins, which are connected to it. To retrieve all connected pins, this method must be called in a loop until its output parameter is NULL.

**Syntax:**

```
virtual Int32 GetNextPinConnected(
    IFxPin** ppFxPin
)PURE;
```

**Parameters:**

*ppFxPin*
> [out] Address of a variable that receives a pointer to an IFxPin interface. The address is equal to
> NULL in case where there are no more connected pins.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The method GetPinConnected must call before to initialize the call iteration.

The call iteration is reset by the IFxPin::ConnectPin and IFxPin::DisconnectPin methods. It means that the GetPinConnected must call back to re-initialize the call interation.

On a call on an input pin, this method always returns a NULL pointer.

**Requirements:**

Header: IFxPin.h.

---

**IFxPin::GetConnectionMediaType**

The GetConnectionMediaType method gets the current MediaType of the connected pin. If the pin is not connected, defaults values are returned (see MAIN_TYPE_UNDEFINED and SUB_TYPE_UNDEFINED).

**Syntax:**

```
virtual Int32 GetConnectionMediaType(
    PFX_MEDIA_TYPE pMediaType
)PURE;
```

**Parameters:**

*pMediaType*
        [out] Pointer to a FX_MEDIA_TYPE structure that receives the connected media type.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxPin.h.

---

**IFxPin::DeliverMedia**

The DeliverMedia method delivers a FxMedia to the connected input/output pin.

**Syntax:**

```
virtual Int32 DeliverMedia(
    IFxMedia* pIFxMedia
)PURE;
```

**Parameters:**

*pIFxMedia*
        [out] Pointer to a FxMedia interface to deliver.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

Call IFxPin::GetDeliveryMedia method to obtain a free FxMedia.

When you call the IFxPin::DeliverMedia method, FxMedia object is automaticly released.

The Fx must never re-use the FxMedia object after it calls DeliverMedia method.

When DeliverMedia is called, the connected pin receives the FxMedia through the IFxPinCallback::FxPin method with STREAM_PROCESS state (see FX_STREAM_STATE).

**Requirements:**

Header: IFxPin.h.

---

**IFxPin::InitStream**

The InitStream method delivers a FxMedia to the connected output pin . In practice, this method is called at the beginning of stream to initialize FXs in chain even if the next FXs in chain are stopped. You can also use it when the stream format is changing.

**Syntax:**

```
virtual Int32 InitStream(
    IFxMedia* pIFxMedia
)PURE;
```

**Parameters:**

*pIFxMedia*
> [out] Pointer to a FxMedia interface to send.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

Call IFxPin::GetDeliveryMedia method to obtain a free FxMedia.

When you call the IFxPin::InitStream method, FxMedia object is automaticly released.

The Fx must never re-use the FxMedia object after it calls InitStream method.

When InitStream is called, the connected pin receives the FxMedia through IFxPinCallback::FxPin method with STREAM_INIT state (see FX_STREAM_STATE).

**Requirements:**

Header: IFxPin.h.

---

`IFxPin::GetFreeMediaNumber`

The GetFreeMediaNumber  method retrieves the free FxMedia number.

**Syntax:**

```
virtual Int32 GetFreeMediaNumber (
    Uint32* pdwFreeMediaNumber
)PURE;
```

**Parameters:**

*pdwFreeMediaNumber*
> [out] Address of a variable that receives the free media number.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

See IFxPinCallback::FxMedia to define the FxMedia number and their size.

**Requirements:**

Header: IFxPin.h.

---

`IFxPin::GetDeliveryMedia`

The GetDeliveryMedia method retrieves a free IFxMedia to fill with data. The GetDeliveryMedia method returns if the time-out interval elapses, or if a free media buffer is available.

**Syntax:**

```
virtual Int32 GetDeliveryMedia(
    IFxMedia** ppIFxMedia,
    Uint32 dwTimeOut
)PURE;
```

**Parameters:**

*ppIFxMedia*
> [out] Address of a variable that receives a pointer to an IFxMedia interface.

*dwTimeOut*
> [in] Variable that contains time-out interval in milliseconds. If dwTimeOut is equal to zero, the method tries to get a free media buffer and returns immediately.
> If dwTimeOut is equal to INFINITE_TIME, the method's time-out interval never elapses.

**Return Values:**

If the method succeeds, it returns FX_OK. if the time-Out is reached, it returns FX_TIMEOUT. Otherwise it returns an FX error code.

**Remarks:**

You can use FxMedia objects to an Fx internal use.

You must always call IFxMedia::Release or IFxPin::DeliverMedia to release the FxMedia object.

The FxMedia contains the main media type of the pin connection otherwhise the default pin media type.

Use the IFxMedia methods to manage the FxMedia object.

**Requirements:**

Header: IFxPin.h.

---

**IFxPin::WaitForIFxMedia**

The WaitForIFxMedia method sends an FxMedia data request to the previous FXs in chain. The method returns immediately (see IFxPinCallback interface).

**Syntax:**

```
virtual Int32 WaitForIFxMedia(
    Uint32 dwTimeStamp,
    FX_PTR dwUser
)PURE;
```

**Parameters:**

*dwTimeStamp*
> [in] Variable that contains the request TimeStamp (Can be NULL).

*dwUser*
> [in] Variable that contains the user extra parameter.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

Use this method on an Input pin only.

The TimeStamp value depends of the FxMedia data. For example, it can be a packet number or a sample number.

**Requirements:**

Header: IFxPin.h.

---

`IFxPin::InitDumpData`

The InitDumpData method initializes the dump of pin data in a file.

**Syntax:**

```
virtual Int32 InitDumpData(
    const std::string strFilePath
)PURE;
```

**Parameters:**

*strFilePath*
> [in] Variable that contains the dump file path.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxPin.h.


`IFxPin::StartDumpData`

The StartDumpData method starts the dump of pin data.

**Syntax:**

```
virtual Int32 StartDumptData(
)PURE;
```

**Parameters:**

None.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxPin.h.


`IFxPin::StopDumpData`

The StopDumpData method stops the dump of pin data.

**Syntax:**

```
virtual Int32 StopDumptData(
)PURE;
```

**Parameters:**

None.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxPin.h.

---

`IFxPin::Flush`

The Flush method propagates the flush notification. This method must be sent at the end of stream to flush the next FXs in chain. This method can be use even when the Fx is stopped.

**Syntax:**

```
virtual Int32 Flush(
)PURE;
```

**Parameters:**

None.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The STREAM_FLUSH state will be set in the IFxPinCallback::FxPin method (see FX_STREAM_STATE).

Requirements:

Header: IFxPin.h.

---

`IFxPin::GetProcessTime`

The GetProcessTime method gets the processing time of an input pin in ms.

**Syntax:**

```
virtual Int32 GetProcessTime(
    Uint32* pdwProcessingTime
)PURE;
```

**Parameters:**

*pdwProcessingTime*
        [in] Pointer to a variable that receives the processing time.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The GetProcessTime method is valid only when the input pin is about to working.

Requirements:

Header: IFxPin.h.

## 3.7. IFXPINCALLBACK

IFxPinCallback is a callback interface for Fx pins. User implements the IFxPinCallback methods if necessary and according to the pin type.

---

The IFxPinCallback interface is used in the IFxPinManager::Create.

Several Pins can use the same IFxPinCallback interface.

**IFxPinCallback::FxPin**

Implement the FxPin method if you want to receive new Fx Media on the input pin (see IFxPin::DeliverMedia method).

This callback can rejects the Fx Media, returns immediately and processes the Fx Media in a thread or processes the Fx media before returning.

**Syntax:**

```
virtual Int32 FxPin(
    IFxPin* pFxPin,
    IFxMedia* pIFxMedia,
    FX_STREAM_STATE StreamState
);
```

**Parameters:**

*pFxPin*
> [in] Address of the input pin that receives the new IFxMedia.

*pIFxMedia*
> [in] Address of an IFxMedia that contains the new Fx Media. This object does not have to be modified. Call GetDeliveryMedia method from the IFxPin interface to get free IFxMedia to fill.

*StreamState*
> [in] Variable that contains the stream process information. StreamState can take either STREAM_PROCESS, STREAM_INIT or STREAM_FLUSH value (see FX_STREAM_STATE).
> - STREAM_PROCESS: normal data computing.
> - STREAM_INIT: incoming new stream format.
> - STREAM_FLUSH: Fx must be flushed. In this case, the pIFxMedia is null

**Return Values:**

If the method succeeds, it returns FX_OK.

If the FxMedia must be repeated, it returns FX_REPEATFXMEDIA.

Otherwise it returns an FX error code.

**Remarks:**

This method doesn't have to be blocking !!.

After the processed incoming new stream format, you have to propagate the STREAM_INIT state (see CFxPin::InitStream) to the Fx in chain and it's recommended to set the Fx state to FX_STREAM_INIT_STATE (see IFxState::FxPublishState method).

After your Fx flush done, you have to propagate the STREAM_FLUSH state (see CFxPin::Flush) to the next Fx in chain.

When the flushing is performed, it's recommended to set the Fx state to FX_FLUSH_STATE (see IFxState::FxPublishState method).

On an Output pin, FxPin method is not implemented or shall return NOT_IMPLEMENTED.

**Requirements:**

Header: IFxPinCallback.h.

**IFxPinCallback::FxMedia**

Implement the FxMedia method if you want to decide the media buffer properties, otherwise the media buffer properties are defined by the default values: size = 0x400 (10Ko) and number = 0x14 (20).

The FxMedia buffer size depends of the media type using by the Fx. You have to choose the appropriate size.

The number of free FxMedia buffers depends of the Fx task.

**Syntax:**

```
virtual Void FxMedia(
    IFxPin* pFxPin,
    Uint32* pdwFxMediaSize,
    Uint32* pdwFxMediaNumber
);
```

**Parameters:**

*pFxPin*
    [in] Address of the out pin which decides the FxMedia properties.
*pdwFxMediaSize*
    [out] Pointer to a variable that receives the size of each media buffer in bytes.
*pdwFxMediaNumber*
    [out] Pointer to a variable that receives the Number of media buffers.

**Return Values:**

None.

**Remarks:**

During the pins connection, each pin gives the media buffer properties that they need. Only free media buffers are updated according to the maximum of both pin's properties.

Each pin calls the IFxPin::GetDeliverMedia to obtain a free media buffer to use.

The IFxMedia::GetSize method retrieves the FxMedia buffer size.

The IFxPin::GetFreeMediaNumber method retrieves the number of free FxMedia buffer.

**Requirements:**

Header: IFxPinCallback.h.


`IFxPinCallback::FxPinState`

Implement the FxPinState method to receive the state of an input/output pin.

**Syntax:**

```
virtual Int32 FxPinState(
    IFxPin* pFxPin ,
    FX_PIN_STATE PinState
);
```

**Parameters:**

*pFxPin*
    [in] Address of an IFxPin that receives the state.
*PinState*
    [in] The state of pFxPin (see FX_PIN_STATE).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

This method doesn't have to be blocking !!.

**Requirements:**

Header: IFxPinCallback.h.

---

**IFxPinCallback::FxWaitForIFxMedia**

Implement the FxWaitForIFxMedia method if you want to receive the FxMedia request order on your outputPin (see IFxPin::WaitForIFxMedia method).

The callback can return immediately and performs a FxMedia sending in a thread or performs a FxMedia sending before returning (see IFxPin::DeliverMedia method).

**Syntax:**

```
virtual Int32 FxWaitForIFxMedia(
    IFxPin* pFxPin,
    Uint32 dwTimeStamp,
    FX_PTR dwUser
);
```

**Parameters:**

*pFxPin*
> [in] Address of the out pin that receives the order.

*dwTimeStamp*
> [in] Variable that contains the request TimeStamp. If dwTimeStamp is 0, the user has to decide the data size to send.

*dwUser*
> [in] Variable that contains the user extra parameter.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

When this callback is called, you have to delivery the requested FxMedia data.

User has to propagate (if necessary) the request order on the input pins.

This method doesn't have to be blocking !!.

On an Input pin, FxWaitForIFxMedia method shall return NOT_IMPLEMENTED or is not implemented.

**Requirements:**

Header: IFxPinCallback.h.

## 3.8. IFXPARAM

The IFxParam interface allows to manage the public Fx parameters.

Use the IFx::FxGetInterface method with IFX_PARAM parameter to retrieve IFxParam from Fx.

Use the QueryFxParamInterface function to retrieve IFxPinManager from the application.

---

**IFxParam::AddFxParam**

The AddFxParam method adds a parameter to the Fx.

**Syntax:**

```
virtual Int32 AddFxParam(
    const PFX_PARAM pFxParam
)PURE;
```

**Parameters:**

*pFxParam*
>  [in] Pointer to a variable that contains the FX_PARAM structure (see FX_PARAM).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

You can add Fx's parameter at any time in the Fx's life. In this case, it can be useful to publish the FX_PARAM_UPDATE state to inform Fx observers or to call the FEF_UpdateFxParam function from application to inform Fx.

**Requirements:**

Header: IFxParam.h.

---

`IFxParam::AddFxParam`

The AddFxParam method adds a string parameter to the Fx.

**Syntax:**

```
virtual Int32 AddFxParam(
    const PFX_PARAM_STRING pFxParam
)PURE;
```

**Parameters:**

*pFxParam*
>  [in] Pointer to a variable that contains the FX_PARAM_STRING structure (see FX_PARAM_STRING).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

You can add Fx's parameter at any time in the Fx's life. In this case, it can be useful to publish the FX_PARAM_UPDATE state to inform Fx observers or to call the FEF_UpdateFxParam function from application to inform Fx.

**Requirements:**

Header: IFxParam.h.

---

`IFxParam::RemoveFxParam`

The RemoveFxParam method removes a Fx parameter.

**Syntax:**

```
virtual Int32 RemoveFxParam(
    const std::string strParamName,
)PURE;
```

**Parameters:**

*strParamName*
>  [in] Variable that contains the parameter name.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

You can remove Fx's parameter at any time in the Fx's life. In this case, it can be useful to publish the FX_PARAM_UPDATE state to inform Fx observers or to call the FEF_UpdateFxParam function from application to inform Fx.

**Requirements:**

Header: IFxParam.h.

---

**IFxParam::FxReleaseInterface**

The FxReleaseInterface method releases the IFxParam interface. This method decreases the reference count by 1.

**Syntax:**

```
virtual FxReleaseInterface(
)PURE;
```

**Parameters:**

None

**Return Values:**

The new reference count.

**Remarks:**

None.

**Requirements:**

Header: IFxParam.h.

---

**IFxParam::GetFxParamCount**

The GetFxParamCount method gets the number of Fx parameters.

**Syntax:**

```
virtual Int32 GetFxParamCount(
    Uint16* pwParamCount
)PURE;
```

**Parameters:**

*pwParamCount*
        [out] Pointer to a variable that receive the number of parameters.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxParam.h.

---

**IFxParam::GetFxParamStringCount**

The GetFxParamCount method gets the number of Fx string parameters.

---

**Syntax:**

```
virtual Int32 GetFxParamStringCount(
    Uint16* pwParamCount
)PURE;
```

**Parameters:**

*pwParamCount*
        [out] Pointer to a variable that receive the number of parameters.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxParam.h.


`IFxParam::GetFxParam`

The GetFxParam method gets a Fx parameter by its index.

**Syntax:**

```
virtual Int32 GetFxParam(
    const FX_PARAM** ppFxParam,
    const Uint16 wFxParamIndex
)PURE;
```

**Parameters:**

*ppFxParam*
        [out] Address of a variable that receives a pointer to an FX_PARAM structure.
*wFxParamIndex*
        [in] Index of the parameter to get. From 0 to parameter count - 1.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

Use the IFxParam::GetFxParamCount method to retrieve the parameter count.

**Requirements:**

Header: IFxParam.h.


`IFxParam::GetFxParam`

The GetFxParam method gets a Fx string parameter by its index.

**Syntax:**

```
virtual Int32 GetFxParam(
    const FX_PARAM_STRING** ppFxParam,
    const Uint16 wFxParamIndex
)PURE;
```

**Parameters:**

*ppFxParam*

>   [out] Address of a variable that receives a pointer to an FX_PARAM_STRING structure.

*wFxParamIndex*

>   [in] Index of the parameter to get. From 0 to parameter count - 1.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

Use the IFxParam::GetFxParamCount method to retrieve the parameter count.

**Requirements:**

Header: IFxParam.h.


**IFxParam::SetFxParamValue**

The SetFxParamValue method sets the value of a Fx parameter.

**Syntax:**

```
virtual Int32 SetFxParamValue(
    const std::string strParamName,
    const Void* pvParamValue
)PURE;
```

**Parameters:**

*strParamName*

>   [in] Variable that contains the parameter name.

*pvParamValue*

>   [in] Pointer to a variable that contains the new parameter value.


**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The new parameter value must be set according to the parameter type and the parameter number.

You can set Fx's parameter at any time in the Fx's life. In this case, it can be useful to publish the FX_PARAM_UPDATE state to inform Fx observers or to call the FEF_UpdateFxParam function from application to inform Fx

**Requirements:**

Header: IFxParam.h.


**IFxParam::SetFxParamValue**

The SetFxParamValue method sets the value of a Fx string parameter.

**Syntax:**

```
virtual Int32 SetFxParamValue(
    const std::string strParamName,
    const std::string strParamValue
)PURE;
```

**Parameters:**

*strParamName*

>   [in] Variable that contains the parameter name.

*strParamValue*
> [in] Pointer to a variable that contains the new parameter value.


**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

You can set Fx's parameter at any time in the Fx's life. In this case, it can be useful to publish the FX_PARAM_UPDATE state to inform Fx observers or to call the FEF_UpdateFxParam function from application to inform Fx

**Requirements:**

Header: IFxParam.h.


---

**IFxParam::GetFxParamValue**

The GetFxParamValue method gets the value of a Fx parameter.

**Syntax:**

```
virtual Int32 GetFxParamValue(
    const std::string strParamName,
    Void* pvParamValue
)PURE;
```

**Parameters:**

*strParamName*
> [in] Variable that contains the parameter name.

*pvParamValue*
> [out] Address to a variable that receives the parameter value.


**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The pvParamValue must be allocated according to the parameter type and the parameter number.

**Requirements:**

Header: IFxParam.h.


---

**IFxParam::GetFxParamValue**

The GetFxParamValue method gets the value of a Fx string parameter.

**Syntax:**

```
virtual Int32 GetFxParamValue(
    const std::string strParamName,
    std::string& strParamValue
)PURE;
```

**Parameters:**

*strParamName*
> [in] Variable that contains the parameter name.

*pvParamValue*
> [out] Reference to a variable that receives the parameter value.

---

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxParam.h.

## 3.9. IFXREFCLOCK

The IFxRefClock interface allows to a Fx to publish and/or receive a reference clock.

Like time stamp, the reference clock value depends of the IFxMedia data.

Use the IFx::FxGetInterface method with IFX_REFCLOCK parameter to retrieve IFxRefClock.

See GetFxEngineRefClock and SetFxEngineRefClock to manage the FxEngine reference clock.

`IFxRefClock::SetFxRefClock`

The SetFxRefClock method publishes a reference clock.

**Syntax:**

```
virtual Int32 SetFxRefClock(
    Uint64 qRefClock,
    Uint32 dwId
)PURE;
```

**Parameters:**

*qRefClock*
        [in] Variable that contains the reference clock to publish.
*dwId*
        [in] Variable that contains the Fx reference clock ID.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The reference clock ID must be a unique number to avoid conflicts.

You can use the IFxParam interface to publish the ID.

**Requirements:**

Header: IFxRefClock.h.

`IFxRefClock::GetFxRefClock`

The GetFxRefClock method gets the latest reference clock published.

**Syntax:**

```
virtual Int32 GetFxRefClock(
    Uint64* pqRefClock,
    Uint32 dwId
)PURE;
```

**Parameters:**

*pqRefClock*

[out] Variable that receives the reference clock. Can be null, if no reference clock exists.

*dwId*

[in] Variable that contains the Fx reference clock ID.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxRefClock.h.

---

`IFxRefClock::GetFxEngineRefClock`

The GetFxEngineRefClock method gets the latest reference clock published by the FxEngine.

See SetFxEngineRefClock and GetFxEngineRefClock of the FxEngine API.

**Syntax:**

```
virtual Int32 GetFxEngineRefClock(
    Uint64* pqRefClock
)PURE;
```

**Parameters:**

*pqRefClock*

[out] Variable that receives the reference clock. Can be null, if no reference clock exists.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxRefClock.h.

---

`IFxRefClock::FxReleaseInterface`

The FxReleaseInterface method releases the IFxRefClock interface. This method decreases the reference count by 1.

**Syntax:**

```
virtual Int32 FxReleaseInterface(
)PURE;
```

**Parameters:**

None.

**Return Values:**

The new reference count.

**Remarks:**

None.

**Requirements:**

Header: IFxRefClock.h.

## 3.10. IFXMEDIA

The IFxMedia interface contains methods to manage properties on FxMedia data. A FxEngine data is a memory object that contains a block of data.

Some FxMedia support the GetFormatInterface method to get format properties (see IFxPcmFormat, IFxVideoImgFormat, IFxVectorFormat and IFxMatrixFormat interfaces).

`IFxMedia::CheckMediaType`

The CheckMediaType method determines if the FxEngine data is matching to a specific FxMedia type.

**Syntax:**

```
virtual Int32 CheckMediaType(
    PFX_MEDIA_TYPE pMediaType
)PURE;
```

**Parameters:**

*pMediaType*
> [in] Pointer to a variable that contains the FX_MEDIA_TYPE structure to compare (see FX_MEDIA_TYPE).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxMedia.h.

`IFxMedia::SetMediaType`

The SetMediaType method sets a FxMedia type of the FxMedia data.

**Syntax:**

```
virtual Int32 SetMediaType(
    PFX_MEDIA_TYPE pMediaType
)PURE;
```

**Parameters:**

*pMediaType*
> [in] Pointer to a variable that contains the new FX_MEDIA_TYPE structure to set (see FX_MEDIA_TYPE).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxMedia.h.

---

`IFxMedia::GetMediaType`

The GetMediaType method gets a FxMedia type of the FxMedia data.

**Syntax:**

```
virtual Int32 GetMediaType(
    PFX_MEDIA_TYPE pMediaType
)PURE;
```

**Parameters:**

*pMediaType*
> [out] Pointer to a variable that receives the FX_MEDIA_TYPE structure (see FX_MEDIA_TYPE).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxMedia.h.

---

`IFxMedia::GetFormatInterface`

The GetFormatInterface method gets a specific format interface on a FxMedia data. (see IFxPcmFormat, IFxVideoImgFormat, IFxVectorFormat and IFxMatrixFormat interfaces).

**Syntax:**

```
virtual Int32 GetFormatInterface(
    FX_SUB_MEDIA_TYPE SubMediaType,
    Void** ppFormatInterface
)PURE;
```

**Parameters:**

*SubMediaType*
> [in] Variable that contains the FX_SUB_MEDIA_TYPE interface type to get.

*ppFormatInterface*
> [out] Address of a variable that receives a pointer to a Format interface (see
> FX_SUB_MEDIA_TYPE).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

FEF provides several format interfaces: IFxPcmFormat for audio PCM data (PCM sub-type), IFxVideoImgFormat for Video and Image data (BGR to Y800 sub-types), IFxVectorFormat for vector (VECTOR sub-type), and IfxMatrixFormat for Matrix (MATRIX sub-type).

Please contact SMProcess for additional interfaces or additional media sub-types.

**Requirements:**

Header: IFxMedia.h.

---

`IFxMedia::GetSize`

The GetSize method retrieves the size of the current FxMedia in byte.

**Syntax:**

```
virtual Int32 GetSize(
    Uint32* pdwSize
)PURE;
```

**Parameters:**

*pdwSize*
> [out] Pointer to a variable that receives the data size.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

See IFxPinCallback::FxMedia to define the FxMedia size.

**Requirements:**

Header: IFxMedia.h.


`IFxMedia::GetDataLength`

The GetDataLength method retrieves the length (in byte) of the valid data in the FxMedia data.

**Syntax:**

```
virtual Int32 GetDataLength(
    Uint32* pdwLength
)PURE;
```

**Parameters:**

*pdwLength*
> [out] Pointer to a variable that receives the data size.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxMedia.h.


`IFxMedia::SetSize`

The SetSize method allows to re-allocate the memory block. Use this methode when it's **necessary** to allocate bigger memory than allocation done in IFxPinCallback::FxMedia method.

**Syntax:**

```
virtual Int32 SetSize(
    Uint32 dwSize
)PURE;
```

**Parameters:**

*pdwSize*
>		[in] Variable that contains the block size in byte.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

See IFxPinCallback::FxMedia to define the FxMedia size on pin connection.

**Requirements:**

Header: IFxMedia.h.

---

`IFxMedia::SetDataLength`

The SetDataLength method sets the length (in byte) of the valid data in the FxMedia data.

**Syntax:**

```
virtual Int32 SetDataLength(
    Uint32 dwLength
)PURE;
```

**Parameters:**

*dwLength*
>		[in] Variable that contains the data length.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxMedia.h.

---

`IFxMedia::GetMediaPointer`

The GetMediaPointer method retrieves a read/write pointer to the current block of data.

**Syntax:**

```
virtual Int32 GetMediaPointer(
    Uint8** ppbMediaData
)PURE;
```

**Parameters:**

*ppbMediaData*
>		[out] Address of a variable that receives a pointer to a buffer's memory.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxMedia.h.

The Release method releases an FxMedia object given by the IFxPin::GetDeliveryMedia method and which will be not delivered.

**Syntax:**

```
virtual Int32 Release (
)PURE;
```

**Parameters:**

None.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

You must never call IFxPin::DeliverMedia and Release methods on the same FxMedia object.

The Fx must never re-use the FxMedia object after it calls Release method.

**Requirements:**

Header: IFxMedia.h.

`IFxMedia::GetTimeStamp`

The GetTimeStamp method retrieves the current TimeStamp of the FxMedia data.

**Syntax:**

```
virtual Int32 GetTimeStamp(
    Uint64* pqTimeStamp
)PURE;
```

**Parameters:**

*pqTimeStamp*
        [out] Pointer to a variable that receives the time stamp.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The TimeStamp value depends of the IFxMedia data. For example, it can be a packet number or a sample number.

**Requirements:**
Header: IFxMedia.h.

`IFxMedia::SetTimeStamp`

The SetTimeStamp method sets the current TimeStamp to the FxMedia data.

**Syntax:**

```
virtual Int32 SetTimeStamp(
    Uint64 qTimeStamp
)PURE;
```

**Parameters:**

*qTimeStamp*
      [in] Variable that contains the new time stamp to set.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The TimeStamp value depends of the IFxMedia data.

**Requirements:**

Header: IFxMedia.h.

`IFxMedia::GetMediaMarker`

The GetMediaMarker method gets the MediaMarker of the FxMedia data.

MediaMarker allows to mark a FxMedia date in the stream.

**Syntax:**

```
virtual Int32 GetMediaMarker(
    FX_MEDIA_MARKER* pMediaMarker
)PURE;
```

**Parameters:**

*pMediaMarker*
      [out] Pointer to a variable that receives the MediaMarker.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxMedia.h.

`IFxMedia::SetMediaMarker`

The SetMediaMarker method sets the MediaMarker to the FxMedia data (see MEDIA_MARKER).

**Syntax:**

```
virtual Int32 SetMediaMarker(
    FX_MEDIA_MARKER MediaMarker
)PURE;
```

**Parameters:**

*MediaMarker*
      [in] Variable that contains the MediaMarker to set.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxMedia.h.

**IFxMedia::SetUserParams**

The SetUserParams method sets an user parameters to the FxMedia data.

**Syntax:**

```
virtual Int32 SetUserParams(
    FX_PTR dwUserParam1,
    FX_PTR dwUserParam2
)PURE;
```

**Parameters:**

*dwUserParam1*
        [in] Variable that contains the first FX_PTR value to set.
*dwUserParam2*
        [in] Variable that contains the second FX_PTR value to set.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxMedia.h.

**IFxMedia::GetUserParams**

The GetUserParams method gets the User parameters of the FxMedia data.

**Syntax:**

```
virtual Int32 GetUserParams(
    FX_PTR* pdwUserParam1,
    FX_PTR* pdwUserParam2
)PURE;
```

**Parameters:**

*pdwUserParam1*
        [out] Pointer to a variable that receives the first FX_PTR value.
*pdwUserParam2*
        [out] Pointer to a variable that receives the second FX_PTR value.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxMedia.h.

---

**IFxMedia::GetFxMediaName**

The GetFxMediaName retrieves the FxMedia name. The FxName allows to give a name to the data inside theFxMeia.

**Syntax:**

```
virtual Int32 GetFxMediaName (
    std::string& strFxMediaName
)PURE;
```

**Parameters:**

*strFxMediaName*
>        [out] Reference on a variable that receives the FxMedia name.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxMedia.h.

---

**IFxMedia::SetFxMediaName**

The SetFxMediaName method sets the FxMedia name.

**Syntax:**

```
virtual Int32 SetFxMediaName(
    const std::string strFxMediaName
)PURE;
```

**Parameters:**

*strFxMediaName*
>        [in] Variable that contains the FxMedia name.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxMedia.h.

---

**IFxMedia::Copy**

The Copy method enables you to copy FxMedia data and properties.

---

**Syntax:**

```
virtual Void Copy(
    IFxMedia* const pIFxMedia
)PURE;
```

**Parameters:**

*pIFxMedia*
>   [in] Pointer to a variable that contains the FxMedia object.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

All FxMedia properties are copied including sub-type properties (i.e. IFxPcmFormat, IFxVideoImgFormat, IFxVectorFormat and IfxMatrixFormat.

**Requirements:**

Header: IFxMedia.h.

## 3.11.  IFXPCMFORMAT

The IFxPcmFormat interface contains methods to manage the PCM sub format properties.

Use the IFxMedia::GetFormatInterface method with PCM sub format  parameter to retrieve IFxPcmFormat.

`IFxPcmFormat::GetPcmFormat`

The GetPcmFormat method retrieves the PCM format of the FxMedia data.

**Syntax:**

```
virtual Int32 GetPcmFormat(
    PFX_PCM_FORMAT pPcmFormat
)PURE;
```

**Parameters:**

*pPcmFormat*
>   [out] Pointer to a variable that receives the PFX_PCM_FORMAT structure (see FX_PCM_FORMAT).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxPcmFormat.h.

`IFxPcmFormat::SetPcmFormat`

The SetPcmFormat method sets the PCM format of the FxMedia data.

**Syntax:**

```
virtual Int32 SetPcmFormat(
    PFX_PCM_FORMAT pPcmFormat
```

```
)PURE;
```

**Parameters:**

*pPcmFormat*
       [in] Pointer to a variable that receives the PFX_PCM_FORMAT structure (see FX_PCM_FORMAT).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxPcmFormat.h.

---

**IFxPcmFormat::GetBitsPerSample**

The GetBitsPerSample method returns the number of bits per sample for the format type specified by FormatTag.

**Syntax:**

```
virtual Int32 GetBitsPerSample (
    Uint16* pwBitsPerSample
)PURE;
```

**Parameters:**

*pwBitsPerSample*
       [out] Pointer to a variable that receives the number of bits per sample.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxPcmFormat.h.

---

## 3.12. IFXVIDEOIMGFORMAT

IFxVideoImgFormat interface contains methods to manage the Video and Image format definition.

Use the IFxMedia::GetFormatInterface method with BGR to Y800 sub formats parameter to retrieve IFxVideoImgFormat.

---

**IFxVideoImgFormat::GetVideoImgProperties**

The GetVideoImgProperties method retrieves the Video/Image properties (Width * Height).

**Syntax:**

```
virtual Int32 GetVideoImgProperties(
    Uint32* pdwWidth,
    Uint32* pdwHeight
)PURE;
```

---

**Parameters:**

*pdwWidth*
> [out] Pointer to a variable that receives the width of video/Img.

*pdwHeight*
> [out] Pointer to a variable that receives the height of video/Img.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxVideoImgFormat.h.

---

`IFxVideoImgFormat::SetVideoImgProperties`

The SetVideoImgProperties method sets the Video/Image properties (Width * Height).

**Syntax:**

```
virtual Int32 SetVideoImgProperties(
    Uint32 dwWidth,
    Uint32 dwHeight
)PURE;
```

**Parameters:**

*dwWidth*
> [in] Variable that contains the new width of the video/Img.

*dwHeight*
> [in] Variable that receives the new height of the video/Img.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The image properties must be accorded to the FxMedia data length (see IFxMedia::SetDataLength).

**Requirements:**

Header: IFxVideoImgFormat.h.

## 3.13. IFXVECTORFORMAT

The IFxVectorFormat interface contains methods to manage the vector format definition.

Use the IFxMedia::GetFormatInterface method with VECTOR sub formats parameter to retrieve IFxVectorFormat.

---

`IFxVectorFormat::GetUnitType`

The GetUnitType method retrieves the Unit type of the Vector Media (see FX_UNIT_TYPE).

**Syntax:**

```
virtual Int32 GetUnitType(
```

```
    FX_UNIT_TYPE* pUnitType
)PURE;
```

**Parameters:**

*pUnitType*
        [out] Pointer to a variable that receives the Type of the vector component.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: IFxVectorFormat.h.

---

**IFxVectorFormat::SetUnitType**

The SetUnitType method sets the Vector Unit type (see FX_UNIT_TYPE).

**Syntax:**

```
virtual Int32 SetUnitType(
    FX_UNIT_TYPE UnitType
)PURE;
```

**Parameters:**

*UnitType*
        [in] Variable that contains the type of the Vector component.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The vector properties (unit type and size) must be accorded to the FxMedia data length (see IFxMedia::SetDataLength).

**Requirements:**

Header: IFxVectorFormat.h.

---

**IFxVectorFormat::GetVectorProperty**

The GetVectorProperty method retrieves the vector component number (M).

**Syntax:**

```
virtual Int32 GetVectorProperty(
    Uint32* pdwM
)PURE;
```

**Parameters:**

*pdwM*
        [out] Pointer to a variable that receives the vector component number.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None

**Requirements:**

Header: IFxVectorFormat.h.

---

`IFxVectorFormat::SetVectorProperty`

The SetVectorProperty method sets the vector component number (M).

**Syntax:**

```
virtual Int32 SetVectorProperty(
    Uint32 dwM
) PURE;
```

**Parameters:**

*dwM*
        [in] Variable that contains the new vector component number.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The vector properties (unit type and size) must be accorded to the FxMedia data length (see IFxMedia::SetDataLength).

**Requirements:**

Header: IFxVectorFormat.h.

---

## 3.14. IFXMATRIXFORMAT

The IFxMatrixFormat interface contains methods to manage the matrix format properties.

Use the IFxMedia::GetFormatInterface method with MATRIX sub formats parameter to retrieve IFxMatrixFormat.

---

`IFxMatrixFormat::GetUnitType`

The GetUnitType method retrieves the Unit type of the Matrix Media (see FX_UNIT_TYPE).

**Syntax:**

```
virtual Int32 GetUnitType(
    FX_UNIT_TYPE* pUnitType
) PURE;
```

**Parameters:**

*pUnitType*
        [out] Pointer to a variable that receives the Type of the matrix component.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

---

**Remarks:**

None.

**Requirements:**

Header: IFxMatrixFormat.h.

---

`IFxMatrixFormat::SetUnitType`

The SetUnitType method sets the Matrix Unit type (see FX_UNIT_TYPE).

**Syntax:**

```
virtual Int32 SetUnitType(
    FX_UNIT_TYPE UnitType
) PURE;
```

**Parameters:**

*UnitType*
        [in] Variable that contains the type of the matrix component.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The matrix properties (unit type and size) must be accorded to the FxMedia data length (see IFxMedia::SetDataLength).

**Requirements:**

Header: IFxMatrixFormat.h.

---

`IFxMatrixFormat::GetMatrixProperties`

The GetMatrixProperties method retrieves the Matrix properties (N * M).

**Syntax:**

```
virtual Int32 GetMatrixProperties(
    Uint32* pdwN,
    Uint32* pdwM
) PURE;
```

**Parameters:**

*pdwN*
        [out] Pointer to a variable that receives the N parameter of the matrix.
*pdwM*
        [out] Pointer to a variable that receives the M parameter of the matrix.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None

**Requirements:**

Header: IFxMatrixFormat.h.

---

---

`IFxMatrixFormat::SetMatrixProperties`

The SetMatrixProperties method sets the matrix properties (N * M).

**Syntax:**

```
virtual Int32 SetMatrixProperties(
    Uint32 dwN,
    Uint32 dwM
) PURE;
```

**Parameters:**

*dwN*
>   [in] Variable that contains the new N value.

*dwM*
>   [in] Variable that contains the new M value.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The matrix properties (unit type and size) must be accorded to the FxMedia data length (see IFxMedia::SetDataLength).

.**Requirements:**

Header: IFxMatrixFormat.h.

# 4. FXENGINE API

The FxEngine API is a set of methods to build the FxEngine. Through the FxEngine, applications can build a plugin architecture. The next sections describe the methods of the FxEngine API.

`FEF_CreateFxEngine`

The FEF_CreateFxEngine create a FxEngine Instance. The FxEngine Instance is identified by its FX_HANDLE value.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_CreateFxEngine(
    FX_HANDLE* phFxEngine
);
```

**Parameters:**

*phFxEngine*
>   [out] Pointer to a variable that receives the FxEngine handle.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

---

Header: FxEngine.h.

---

**FEF_ReleaseFxEngine**

The FEF_ReleaseFxEngine function releases a FxEngine Instance that was created with the FEF_CreateFxEngine function.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_ReleaseFxEngine (
    FX_HANDLE hFxEngine
);
```

**Parameters:**

*hFxEngine*
> [in] Handle of the FxEngine instance to release.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: FxEngine.h.

---

**FEF_GetFxEngineVersion**

The FEF_GetFxEngineVersion function gets the FxEngine API version.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_GetFxEngineVersion (
    Uint16* pwMajor,
    Uint16* pwMinor,
    Uint16* pwBuild,
    Uint16* pwRev
);
```

**Parameters:**

*pwMajor*
> [out] Pointer to a variable that receives the Major of FxEngine API version.

*pwMinor*
> [out] Pointer to a variable that receives the Minor of FxEngine API version.

*pwBuild*
> [out] Pointer to a variable that receives the Build of FxEngine API version.

*pwRev*
> [out] Pointer to a variable that receives the Revision of FxEngine API version.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: FxEngine.h.

---

**FEF_AddFx**

The FEF_AddFx function allows to add a Fx in the FxEngine using the Fx DLL path.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_AddFx (
    FX_HANDLE hFxEngine,
    const std::string strFx,
    FX_HANDLE* phFx
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*strFx*
        [in] Variable that contains the Fx path.
*phFx*
        [out] Pointer to a variable that receives the Fx handle.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The Framework calls in order the IFxBase methods:

IFxBase::InitFx()  ➔  IFxBase::GetFxInfo().

**Requirements:**

Header: FxEngine.h. `GetFxInfo`

**FEF_AddFxEx**

The FEF_AddFxEx function allows to add a Fx in the FxEngine using an IFxBase interface.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_AddFxEx (
    FX_HANDLE hFxEngine,
    IFxBase* pIFxBase,
    FX_HANDLE* phFx
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*pIFxBase*
        [in] Pointer to a variable that contains the IFxBase interface of Fx to add
*phFx*
        [out] Pointer to a variable that receives the Fx handle.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The Framework calls in order the IFxBase methods:

IFxBase::InitFx()  ➔  IFxBase::GetFxInfo().

**Requirements:**

Header: FxEngine.h.

---

`FEF_RemoveFx`

The FEF_RemoveFx function allows to remove a Fx in the FxEngine.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_RemoveFx (
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*hFx*
        [in] Handle of the Fx to remove.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

Call the FEF_DetachFxObserver function before call this function if necessary.

The Framework calls in order the IFxBase methods:

IFxBase::StopFx()  ➔  IFxBase::ReleaseFx().

**Requirements:**

Header: FxEngine.h.

---

`FEF_GetFxCount`

The FEF_GetFxCount returns the number of FXs (N) in the FxEngine.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_GetFxCount (
    FX_HANDLE hFxEngine,
    Uint16* pwFxCount
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*pwFxCount*
        [out] Pointer to a variable that receives the number of Fx.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: FxEngine.h.

---

**FEF_GetFx**

The FEF_GetFx returns the Fx handle in the FxEngine by its index.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_GetFx (
    FX_HANDLE hFxEngine,
    FX_HANDLE* phFx,
    Uint16 wFxCount
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*phFx*
        [out] Pointer to a variable that receives the Fx handle.
*pwFxCount*
        [in] Index of the Fx to get from 0 to N-1. N is given by the GetFxCount function.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: FxEngine.h.


**FEF_StartFxEngine**

The FEF_StartFxEngine function starts the FXs in the FxEngine. The IFxBase::StartFx method of each Fx is called. After the IFxBase::StartFx functions calling, the IFxBase::RunFx method of each Fx is called.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_StartFxEngine(
    FX_HANDLE hFxEngine,
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

FXs are started from the first Fx added to the last Fx added.

For each Fx, if the Fx start succeeds then, the Fx running state is set to FX_START.

**Requirements:**

Header: FxEngine.h.


**FEF_StartFx**

The FEF_StartFx function starts a Fx individually. In order, the IFxBase::StartFx and IFxBase::RunFx functions of Fx are called.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_StartFx (
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*hFx*
        [in] Handle of the Fx plugin to start.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

Use this function to start the FXs in your preferred order.

If the method succeeds, the Fx running state is set to FX_START.

**Requirements:**

Header: FxEngine.h.

---

`FEF_PauseFxEngine`

The FEF_PauseFxEngine function pauses the FXs in the FxEngine. The IFxBase::PauseFx method of each Fx is called ( if it exists).

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_PauseFxEngine (
    FX_HANDLE hFxEngine
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

FXs are paused from the first Fx added to the last Fx added.

For each Fx, if the Fx pause succeeds then, the Fx running state is set to FX_PAUSE.

**Requirements:**

Header: FxEngine.h.

---

`FEF_PauseFx`

The FEF_PauseFx function pauses a Fx if the IFxBase::PauseFx method exists.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_PauseFx (
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx
```

```
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*hFx*
        [in] Handle of the Fx to pause.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

Use this function to pause the FXs in your preferred order.

If the method succeeds, the Fx running state is set to FX_PAUSE.

**Requirements:**

Header: FxEngine.h.

---

**FEF_StopFxEngine**

The FEF_StopFxEngine function stops the FXs in the FxEngine. The IFxBase::StopFx method of each Fx is called.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_StopFxEngine (
    FX_HANDLE hFxEngine
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

FXs are stopped from the first Fx added to the last Fx added.

For each Fx, if the Fx stop succeeds then, the Fx running state is set to FX_STOP.

**Requirements:**

Header: FxEngine.h.

---

**FEF_StopFx**

The FEF_StopFx function stops a Fx individually. The IFxBase::StopFx of Fx is called.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_StopFx (
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx
);
```

**Parameters:**

*hFxEngine*

      [in] Handle of the FxEngine instance.

*hFx*

      [in] Handle of the Fx to stop.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**


Use this function to stop the FXs in your preferred order.

If the method succeeds, the Fx running state is set to FX_STOP.

**Requirements:**

Header: FxEngine.h.


---

### FEF_GetFxInfo

The FEF_GetFxInfo function gets the mains Fx definitions.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_GetFxInfo (
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx,
    const FX_DESCRIPTOR** ppFxDescriptor
);
```

**Parameters:**

*hFxEngine*

      [in] Handle of the FxEngine instance.

*hFx*

      [in] Handle of the Fx.

*ppFxDescriptor*

      [out] Address of a variable that receives a pointer to the Fx descriptor structure (see
      FX_DESCRIPTOR structure).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The GetFxInfo function calls the IFxBase::GetFxInfo method.

**Requirements:**

Header: FxEngine.h.


---

### FEF_GetFxRunningState

The FEF_GetFxRunningState function gets the Fx running state.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_GetFxRunningState (
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx,
    FX_RUNNING_STATE* pFxRunningState
);
```

**Parameters:**

*hFxEngine*
> [in] Handle of the FxEngine instance.

*hFx*
> [in] Handle of the Fx.

*pFxRunningState*
> [out] Pointer to a variable that receives the Fx state (see FX_RUNNING_STATE).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: FxEngine.h.

---

### FEF_GetFxState

The FEF_GetFxState function gets the lastest Fx internal state which has been published by the IFxState::FxPublishState.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_GetFxState (
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx,
    FX_STATE* pFxState
);
```

**Parameters:**

*hFxEngine*
> [in] Handle of the FxEngine instance.

*hFx*
> [in] Handle of the Fx.

*pFxState*
> [out] Pointer to a variable that receives the Fx state (see FX_STATE).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

See the IFxState interface.

**Requirements:**

Header: FxEngine.h.

---

### FEF_GetConstToString

The FEF_GetConstToString function converts a FxEngine constant to a string.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_GetConstToString (
    FXENGINE_CONST_TYPE FxEngineConstType,
    Int32 sdwFxEngineConst,
    std::string& strStateName
);
```

**Parameters:**

*FxEngineConstType*
    [in] Variable that contains the Type of the constant (see FXENGINE_CONST_TYPE).
*sdwFxEngineConst*
    [in] Variable that contains the constant to convert.
*strStateName*
    [out] Reference on variable that receives the constant name.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: FxEngine.h.

---

**FEF_AttachFxObserver**

The FEF_AttachFxObserver function attaches an observer object for getting states and notes of all Fx.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_AttachFxObserver (
    FX_HANDLE hFxEngine,
    CFxStateCallback* pFxStateCallback,
    CFxNoteCallback* pFxNoteCallback
    FX_PTR dwParam,
    FX_HANDLE* phObserverId
);
```

**Parameters:**

*hFxEngine*
    [in] Handle of the FxEngine instance.
*pFxStateCallback*
    [in] Pointer to a CFxStateCallback object to be used during the Fx running to process Fx states.
*pFxNoteCallback*
    [in] Pointer to a CFxNoteCallback object to be used during the Fx running to process Fx notes.
*dwParam*
    [in] First User-supplied callback data.
*phObserverId*
    [out] Pointer to a Handle that receives the ObserverId.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The CFxStateCallback class declaration is defined in the IFxState.h include file.

The CFxNoteCallback class declaration is defined in the IFxNote.h include file.

See the IFxState and IFxNote interfaces.

Do not call the IFxState interface's methods in the CFxStateCallback callback method.

Do not call the IFxNote interface's methods in the CFxSNoteCallback callback method.

**Requirements:**

Header: FxEngine.h.

**FEF_AttachFxObserverEx**

The FEF_AttachFxObserver function attaches an observer for getting states and notes of all Fx.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_AttachFxObserverEx (
    FX_HANDLE hFxEngine,
    FXSTATECALLBACK* pFxStateCallback,
    FXNOTECALLBACK* pFxNoteCallback,
    FX_PTR dwParam,
    FX_HANDLE* phObserverId
);
```

**Parameters:**

*hFxEngine*
    [in] Handle of the FxEngine instance.
*pFxStateCallback*
    [in] Pointer to a fixed callback function to be used during the Fx running to process Fx states.
*pFxNoteCallback*
    [in] Pointer to a fixed callback function to be used during the Fx running to process Fx notes.
*dwParam1*
    [in] First User-supplied callback data.
*phObserverId*
    [out] Pointer to a Handle that receives the ObserverId.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The FXSTATECALLBACK declaration is defined in the IFxState.h include file.

The FXNOTECALLBACK declaration is defined in the IFxNote.h include file.

See the IFxState and IFxNote interfaces.

Do not call the IFxState interface's methods in the callback function.

Do not call the IFxNote interface's methods in the callback function.

**Requirements:**

Header: FxEngine.h.

**FEF_DetachFxObserver**

The FEF_DetachFxObserver function detaches a FxState observer.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_DetachFxObserver (
    FX_HANDLE hFxEngine,
    FX_HANDLE hObserverId
);
```

**Parameters:**

*hFxEngine*
    [in] Handle of the FxEngine instance.
*hObserverId*
    [in] Handle that contains the ObserverId to detach.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The Observer Id is returned by the FEF_AttachFxObserver functions.

**Requirements:**

Header: FxEngine.h.

**FEF_GetFxPinCount**

The FEF_GetFxPinCount function gets the number (N) of pin of a Fx.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_GetFxPinCount (
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx,
    Uint16* pwPinCount
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*hFx*
        [in] Handle of the Fx.
*pwPinCount*
        [out] Pointer to a variable that receives the number of Fx pin.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: FxEngine.h.

**FEF_QueryFxPinInterface**

The FEF_QueryFxPinInterface function retrieves the IFXPin interface of a Fx pin.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_QueryFxPinInterface (
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx,
    IFxPin** ppIFxPin,
    Uint16 wPinIndex
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*hFx*
        [in] Handle of the Fx.
*ppIFxPin*
        [out] Address of a variable that receives a pointer to an IFxPin interface.
*wPinIndex*
        [in] Index of the Fx pin to get from 0 to N-1. N is given by the FEF_GetFxPinCount function.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: FxEngine.h.

---

**FEF_QueryFxParamInterface**

The FEF_QueryFxParamInterface function retrieves the IFXParam interface of a Fx pin.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_QueryFxParamInterface (
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx,
    IFxParam** ppIFxParam
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*hFx*
        [in] Handle of the Fx.
*ppIFxParam*
        [out] Address of a variable that receives a pointer to an IFxParam interface.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: FxEngine.h.

---

**FEF_UpdateFxParam**

The FEF_UpdateFxParam function calls the IFxBase::UpdateFxParam method.

It allows to a Fx to reload the public parameters.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_UpdateFxParam (
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx,
    const std::string strParamName,
    FX_PARAMETER FxParameter
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*hFx*
        [in] Handle of the Fx.
*strParamName*

[in] Variable that contains the parameter name.

*FxParameter*

[in] Variable that contains the updating mode.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

If FxParameter is equal to FX_PARAM_ALL, strParamName is ignored.

**Requirements:**

Header: FxEngine.h.

---

**FEF_ConnectFxPin**

The FEF_ConnectFxPin function allows to connect two pins.

Because a pin can accept several media types, the connection negotiation task gets the first media type of the input pin and checks that one of the output pin media types is acceptable. If not, the FxEngine tries with the next input pin media types.

In case where an output pin is already connected, a new connection will try to connect with the media type of the previous connection only.

Both pins can receive and deliver new FxMedia according to the Fx states (i.e. START, STOP, PAUSE).

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_ConnectFxPin (
    FX_HANDLE hFxEngine,
    IFxPin* pFxPinIn,
    IFxPin* pFxPinOut
);
```

**Parameters:**

*hFxEngine*

[in] Handle of the FxEngine instance.

*pFxPinIn*

[in] Pointer to the input IFxPin.

*pFxPinOut*

[in] Pointer to the output IFxPin.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The IFxPin interfaces are retrieved by the FEF_QueryFxPinInterface  function.

See FEF_ConnectFxPinEx function to connect pins with a specified media type.

**Requirements:**

Header: FxEngine.h.

---

**FEF_ConnectFxPinEx**

The FEF_ConnectFxPinEx function allows to connect two pins with a specified media type.

In case where an output pin is already connected, the given media type must be compatible with the media type of the previous connection.

Both pins can receive and deliver new FxMedia according to the Fx states (i.e. START, STOP, PAUSE).

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_ConnectFxPinEx (
    FX_HANDLE hFxEngine,
    IFxPin* pFxPinIn,
    IFxPin* pFxPinOut,
    PFX_MEDIA_TYPE pMediaType
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*pFxPinIn*
        [in] Pointer to the input IFxPin.
*pFxPinOut*
        [in] Pointer to the output IFxPin.
*PMediaType*
        [in] Pointer to a variable that contains the media type to use.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The IFxPin interfaces are retrieved by the FEF_QueryFxPinInterface  function.

**Requirements:**

Header: FxEngine.h.

---

**FEF_DisconnectFxPin**

The FEF_DisconnectFxPin function allows to disconnect pin.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_DisconnectFxPin (
    FX_HANDLE hFxEngine,
    IFxPin* pFxPin
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*pFxPinIn*
        [in] Pointer to the IFxPin to disconnect.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The IFxPin interfaces are retrieved by the FEF_QueryFxPinInterface  function.

The attached Pin is disconnected too.

**Requirements:**

Header: FxEngine.h.

---

**FEF_SetFxEngineRefClock**

The FEF_SetFxEngineRefClock function sets a reference clock to the Fx engine.

---

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_SetFxEngineRefClock (
    FX_HANDLE hFxEngine,
    Uint64 qRefClock
);
```

**Parameters:**

*hFxEngine*
> [in] Handle of the FxEngine instance.

*qRefClock*
> [in] Reference clock to set.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

Fx uses the IFxRefClock::GetFxEngineRefClock method to retrieve it.

**Requirements:**

Header: FxEngine.h.

---

**FEF_GetFxEngineRefClock**

The FEF_GetFxEngineRefClock function retrieves a reference clock of the Fx engine.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_GetFxEngineRefClock (
    FX_HANDLE hFxEngine,
    Uint64* pqRefClock
);
```

**Parameters:**

*hFxEngine*
> [in] Handle of the FxEngine instance.

*pqRefClock*
> [out] Pointer to a variable that receives the reference clock.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

Fx uses the IFxRefClock::GetFxEngineRefClock method to retrieve it.

**Requirements:**

Header: FxEngine.h.

---

**FEF_GetFxRefClock**

The FEF_GetFxRefClock function retrieves the latest reference clock of a Fx.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_GetFxRefClock (
    FX_HANDLE hFxEngine,
    Uint64* pqRefClock,
    Uint32 dwId
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*pqRefClock*
        [out] Pointer to a variable that receives the reference clock. Can be null, if no reference clock exists.
*dwId*
        [in] Variable that contains the Fx reference clock ID.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

See IFxRefClock interface.

**Requirements:**

Header: FxEngine.h.

---

**FEF_DisplayFxPropertyPage**

The FEF_DisplayFxPropertyPage function allows to display the property page of a Fx, if it exits.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_DisplayFxPropertyPage (
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx,
    Pvoid pvWndParent
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*hFx*
        [in] Handle of the Fx.
*pvWndParent*
        [in] Handle to the parent window (Can be null).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: FxEngine.h.

---

**FEF_GetFxFrame**

The FEF_GetFxFrame function returns the Fx frame if it exists. Fx frame is a XPM image format (see http://koala.ilog.fr/lehors/xpm.html) and allows to any Framework front-end to render the Fx with a picture.

Fx can update at any moment its frame and informs Fx observer with the  FX_FRAME_UPDATE state sending.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_GetFxFrame (
    FX_HANDLE hFxEngine,
```

```
    FX_HANDLE hFx,
    const Char** ppbFxFrame
);
```

**Parameters:**

*hFxEngine*
>       [in] Handle of the FxEngine instance.

*hFx*
>       [in] Handle of the Fx.

*ppbFxFrame*
>       [out] Address of a variable that receives the XPM data (Can be null).

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

None.

**Requirements:**

Header: FxEngine.h.

---

**FEF_GetFxUserInterface**

The FEF_GetFxUserInterface function allows to get an user interface if it exists.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_GetFxUserInterface (
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx,
    Pvoid* ppvUserInterface
);
```

**Parameters:**

*hFxEngine*
>       [in] Handle of the FxEngine instance.

*hFx*
>       [in] Handle of the Fx.

*ppvUserInterface*
>       [out] Address of a variable that receives a pointer to the user interface.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

See the IFxBase::GetFxUserInterface method.

**Requirements:**

Header: FxEngine.h.

---

**FEF_GetFxSubFxEngine**

The FEF_GetFxSubFxEngine function allows to get a FxEngine Handle if it exists.

Fx can contain a sub FxEngine system with several FXs.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_GetFxSubFxEngine (
```

---

```
    FX_HANDLE hFxEngine,
    FX_HANDLE hFx,
    FX_HANDLE* phFxEngine
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*hFx*
        [in] Handle of the Fx.
*phFxEngine*
        [out] Pointer to a variable that receives the FxEngine handle.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

See the IFxBase::GetFxSubFxEngine method.

**Requirements:**

Header: FxEngine.h.


**FEF_SaveFxEngine**

The FEF_SaveFxEngine function allows to save the FxEngine configuration.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_SaveFxEngine (
    FX_HANDLE hFxEngine,
    const std::string strFilePath
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*strFilePath*
        [in] Variable that contains the configuration file path to save.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The FxEngine configuration includes FXs, which are added with FEF_AddFx function only, and their pins connections.

The configuration is saved in a "pseudo" xml form.

See the FEF_LoadFxEngine method.

**Requirements:**

Header: FxEngine.h.


**FEF_LoadFxEngine**

The FEF_LoadFxEngine function allows to load a FxEngine configuration.

**Syntax:**

```
FXENGINE_EXP Int32 FXENGINE_API FEF_LoadFxEngine (
    FX_HANDLE hFxEngine,
    const std::string strFilePath
);
```

**Parameters:**

*hFxEngine*
        [in] Handle of the FxEngine instance.
*strFilePath*
        [in] Variable that contains the configuration file path to load.

**Return Values:**

If the method succeeds, it returns FX_OK. Otherwise it returns an FX error code.

**Remarks:**

The FxEngine configuration includes FXs, which are added with FEF_AddFx function only, their parameters and their pins connections.

See the FEF_SaveFxEngine method.

**Requirements:**

Header: FxEngine.h.


# 5. FRAMEWORK STRUCTURES


`FX_DESCRIPTOR`

Specifies the Fx.

**Syntax:**

```
typedef struct _FX_DESCRIPTOR {
    std::string strName;
    std::string strVersion;
    std::string strAuthor;
    std::string strCopyright;
    FX_TYPE FxType;
    FX_SCOPE FxScope;
} FX_DESCRIPTOR, *PFX_DESCRIPTOR;
```

**Attributes:**

*strName*
        Specifies the Fx name.
*strVersion*
        Specifies the Fx version.
*strAuthor*
        Specifies the Fx author.
*strCopyright*
        Specifies the Fx copyright.
*FxType*
        Specifies the Fx Type (see FX_TYPE).
*FxScope*
        Specifies the Fx Scope (see FX_SCOPE).

**Requirements:**

Header: FxDef.h.

## FX_PIN

Specifies the Fx pin.

**Syntax:**

```
typedef struct _FX_PIN {
    std::string strPinName;
    FX_PIN_TYPE PinType;
    PFX_MEDIA_TYPE pMediaTypes;
    Uint16 wMediaTypeCount;
    IFxPinCallback* pIFxPinCallBack;
} FX_PIN, *PFX_PIN;
```

**Attributes:**

*strPinName*
        Specifies the name of pin.
*PinType*
        Specifies the type of pin.
*pMediaTypes*
        Specifies the array of medias.
*wMediaTypeCount*
        Specifies the number Fx medias that contains pMediaTypes supported by the pin.
*pIFxPinCallBack*
        Specifies the IFxPinCallback interface of pin.

**Requirements:**

Header: IFxPinManager.h.

## FX_MEDIA_TYPE

Specifies the pin media types.

**Syntax:**

```
typedef struct _FX_MEDIA_TYPE {
    MAIN_FX_MEDIA_TYPE  MainMediaType;
    SUB_FX_MEDIA_TYPE  SubMediaType;
} FX_MEDIA_TYPE, *PFX_MEDIA_TYPE;
```

**Attributes:**

*MainMediaType*
        Specifies the Main type of the FxMedia data.
*SubMediaType*
        Specifies the Sub type of the FxMedia data.

**Requirements:**

Header: FxDef.h.

## FX_PARAM

Specifies the Fx parameters.

**Syntax:**

```
typedef struct _FX_PARAM {
    std::string strParamName;
    std::string strParamUnitName;
```

```
    Void* pvDefaultValue;
    Void* pvMinValue;
    Void* pvMaxValue;
    FX_UNIT_TYPE ParamType;
    Uint32 dwParamNumber;
} FX_PARAM, *PFX_PARAM;
```

**Attributes:**

*strParamName*
        Specifies the name of the parameter.
*strParamUnitName*
        Specifies the unit name of the parameter.
*pvDefaultValue*
        Specifies the default values according to its type and number (can be null).
*pvMinValue*
        Specifies the min parameter value (can be null).
*pvMaxValue*
        Specifies the max parameter value (can be null).
*ParamType*
        Specifies the parameter's type.
*dwParamNumber*
        Specifies the number of parameter.

**Requirements:**

Header: FxDef.h.

---

**FX_PARAM_STRING**

Specifies the Fx parameters.

**Syntax:**

```
typedef struct _FX_PARAM_STRING {
    std::string strParamName;
    std::string strDefaultValue;
} FX_PARAM_STRING, *PFX_PARAM_STRING;
```

**Attributes:**

*strParamName*
        Specifies the name of the parameter.
*strDefaultValue*
        Specifies the default value.

**Requirements:**

Header: FxDef.h.

---

**FX_PCM_FORMAT**

Specifies the PCM sub type.

**Syntax:**

```
typedef struct _FX_PCM_FORMAT {
    Uint16 wChannels;
    FX_UNIT_TYPE FormatTag;
    Uint32 dwSamplingRate;
} FX_PCM_FORMAT, *PFX_PCM_FORMAT;
```

**Attributes:**

*wChannels*
>    Specifies the PCM channel number.

*FormatTag*
>    Specifies the PCM sample format (see FX_ UNIT_TYPE).

*dwSamplingRate*
>    Specifies the PCM sampling rate.

**Requirements:**

Header: FxDef.h.


# 6. FRAMEWORK CONSTANTS


**FXENGINE_EXP**

Specifies the FxEngine extended attribute syntax (Windows only).

**Syntax:**

```
#define FXENGINE_EXP __declspec(dllexport)
```

**Requirements:**

Header: FxDef.h.


**FXENGINE_API**

Specifies the FxEngine calling convention (Windows only).

**Syntax:**

```
#define FXENGINE_API __stdcall
```

**Requirements:**

Header: FxDef.h.


**FX ERRORS**

Specifies the Fx general errors.

**Syntax:**

```
#define FX_NOERROR                0
#define FX_OK                     (FX_NOERROR)
#define FX_ERROR                  (FX_NOERROR      - 1)
#define FX_INVALPARAM             (FX_NOERROR      - 2)
#define FX_INVALHANDLE            (FX_NOERROR      - 3)
#define FX_NOMEM                  (FX_NOERROR      - 4)
#define FX_MEDIANOTSUPPORTED      (FX_NOERROR      - 5)
#define FX_SUBMEDIANOTSUPPORTED   (FX_NOERROR      - 6)
#define FX_FMTNOTSUPPORTED        (FX_NOERROR      - 7)
#define FX_ERRORSTATE             (FX_NOERROR      - 8)
#define FX_NOINTERFACE            (FX_NOERROR      - 9)
#define FX_INVALPINTYPE           (FX_NOERROR      - 10)
#define FX_TIMEOUT                (FX_NOERROR      - 11)
#define FX_REPEATFXMEDIA          (FX_NOERROR      + 1)
```

**Details:**

*FX_NOERROR*
> Error Base.

*FX_OK*
> No error.

FX_ERROR
> Unspecified error.

*FX_INVALPARAM*
> Invalid parameter.

*FX_INVALHANDLE*
> Invalid handle.

*FX_NOMEM*
> Memory allocation error.

*FX_MEDIANOTSUPPORTED*
> Media is not supported.

*FX_SUBMEDIANOTSUPPORTED*
> SubMedia is not supported

*FX_FMTNOTSUPPORTED*
> Format is not supported.

*FX_ERRORSTATE*
> Fx state error.

*FX_NOINTERFACE*
> FX interface not found.

*FX_INVALPINTYPE*
> Invalid pin type.

*FX_TIMEOUT*
> TimeOut occurs

*FX_REPEATFXMEDIA*
> FxMedia must be repeated (see IFxPinCallBack interface).

**Requirements:**

Header: FxErr.h.

---

`FX_INTERFACE`

Specifies the Fx interfaces.

**Syntax:**

```
typedef enum _FX_INTERFACE {
    IFX_PINMANGER = 0,
    IFX_PARAM,
    IFX_REFCLOCK,
    IFX_STATE
} FX_INTERFACE;
```

**Attributes:**

*IFX_PINMANGER*
> Fx Pin Manager interface (see IFxPinManager).

*IFX_PARAM*
> Fx parameter interface (see IFxParam).

IFX_REFCLOCK
> Fx reference clock (see IFxRefClock).

*IFX_STATE*
> Fx state interface (see IFxRefClock).

**Requirements:**

Header: FxDef.h.

### FX_PIN_TYPE

Specifies the pin types.

**Syntax:**

```
typedef enum _FX_PIN_TYPE {
    PIN_IN = 0,
    PIN_OUT
} FX_PIN_TYPE;
```

**Attributes:**

*PIN_IN*
        Input pin type
*PIN_OUT*
        Output pin type

**Requirements:**

Header: FxDef.h.

### FX_PIN_STATE

Specifies the pin types.

**Syntax:**

```
typedef enum _FX_PIN_STATE {
    PIN_NOT_CONNECTED = 0,
    PIN_CONNECTED,
    PIN_ERROR
}FX_PIN_STATE;
```

**Attributes:**

*PIN_NOT_CONNECTED*
        Fx pin is not connected
*PIN_CONNECTED*
        Fx pin is connected
*PIN_ERROR*
        Error on the Fx pin

**Requirements:**

Header: FxDef.h.

### FX_STREAM_STATE

Specifies the stream states.

**Syntax:**

```
typedef enum _FX_STREAM_STATE {
    STREAM_PROCESS = 0,
    STREAM_INIT,
    STREAM_FLUSH
}FX_STREAM_STATE;
```

**Attributes:**

*STREAM_PROCESS*
        Normal stream processing
*STREAM_INIT*

New stream format
*STREAM_FLUSH*
End of stream

## Requirements:

Header: FxDef.h.

---

**FX_MAIN_MEDIA_TYPE**

Specifies the Main media types. Main types are used to define media of Fx pins and data flow. Each main type has several sub types.

**Please contact SMProcess for additional media types.**

The sub types are defined in the FxMediaTypes.h header file

### Syntax:

```
typedef enum FX_MAIN_MEDIA_TYPE {
    MAIN_TYPE_UNDEFINED = 0,
    AUDIO_TYPE,
    VIDEO_TYPE,
    TEXT_TYPE,
    DATA_TYPE,
    USER_TYPE
} FX_MAIN_MEDIA_TYPE;
```

### Attributes:

*AUDIO_TYPE*
Audio waveform type
*VIDEO_TYPE*
Video / Image type
*TEXT_TYPE*
Text type
*DATA_TYPE*
Raw type (Vector/Matrix)
*USER_TYPE*
User type

### Requirements:

Header: FxDef.h.

---

**FX_MEDIA_MARKER**

Specifies the FxMedia markers. FxMedia markers are used (optional) to tag the FxMedia.

### Syntax:

```
typedef enum _FX_MEDIA_MARKER{
    UNDEFINED_MARKER = 0,
    DISCONTINUITY_MARKER,
    USER1_MARKER = 100,
    USER2_MARKER,
    USER3_MARKER,
    USER4_MARKER,
    USER5_MARKER,
    USER6_MARKER,
    USER7_MARKER,
    USER8_MARKER
} FX_MEDIA_MARKER;
```

**Attributes:**

*UNDEFINED_MARKER*
        Undefined marker.
*DISCONTINUITY_MARKER*
        Stream discontinuity marker.
*USER1_MARKER to USER8_MARKER*
        User markers.

**Requirements:**

Header: FxDef.h.

---

**FX_TYPE**

Specifies the Fx types.

**Syntax:**

```
typedef enum _FX_TYPE{
    FX_NOT_DEFINED = 0,
    FX_SOURCE,
    FX_RENDERER,
    FX_ANALYSER,
    FX_PROCESS,
    FX_SPLITTER,
    FX_MIXER,
    FX_USER
}FX_TYPE;
```

**Attributes:**

*FX_NOT_DEFINED*
        Undefined Fx type (Default).
*FX_SOURCE*
        Source Fx.
*FX_RENDERER*
        Renderer Fx.
*FX_ANALYSER*
        Analyser Fx.
*FX_PROCESS*
        Process Fx
*FX_SPLITTER*
        Splitter Fx.
*FX_MIXER*
        Mixer Fx.
*FX_USER*
        Free user type.

**Requirements:**

Header: FxDef.h.

---

**FX_SCOPE**

Specifies the Fx scopes.

**Syntax:**

```
typedef enum _FX_SCOPE{
```

---

```
    FX_SCOPE_NOT_DEFINED = 0,
    FX_SCOPE_AUDIO,
    FX_SCOPE_VIDEO,
    FX_SCOPE_TEXT,
    FX_SCOPE_DATA,
    FX_SCOPE_NETWORK,
    FX_SCOPE_ALL = 1000,
    FX_SCOPE_USER = 2000
}FX_SCOPE;
```

**Attributes:**

*FX_SCOPE_NOT_DEFINED*
        Undefined Fx scope (Default).
*FX_SCOPE_AUDIO*
        Audio waveform Fx.
*FX_SCOPE_VIDEO*
        Video / Image Fx.
*FX_SCOPE_TEXT*
        Text Fx.
*FX_SCOPE_DATA*
        Raw Fx (Vector/Matrix).
*FX_SCOPE_NETWORK*
        Network Fx.
*FX_SCOPE_ALL*
        All fx scopes.
*FX_SCOPE_USER*
        Private user scope.

**Requirements:**

Header: FxDef.h.

---

**FX_RUNNING_STATE**

Specifies the Fx running states.

**Syntax:**

```
typedef enum _FX_RUNNING_STATE{
    FX_STOP = 0,
    FX_PAUSE,
    FX_START
}FX_RUNNING_STATE;
```

**Attributes:**

*FX_STOP*
        The Fx is stopped.
*FX_PAUSE*
        The Fx is paused.
*FX_START*
        The Fx is started.

**Requirements:**

Header: FxDef.h.

---

**FX_STATE**

Specifies the Fx states. User has to choose the good Fx state and to publish it. See IFxState::FxPublishState.

The GetFxState and AttachFxObserver methods from the FXEngine API allow to retrieve states of any Fx.

**Syntax:**

```
typedef enum _FX_STATE {
    FX_UNDEFINED_STATE = 0,
    FX_LOADING_STATE,
    FX_RELEASE_STATE,
    FX_INIT_STATE,
    FX_CONNECT_STATE,
    FX_DISCONNECT_STATE,
    FX_STOP_STATE,
    FX_PAUSE_STATE,
    FX_START_STATE,
    FX_RUN_STATE,
    FX_IDLE_STATE,
    FX_FLUSH_STATE,
    FX_PARAM_UPDATE,
    FX_PIN_UPDATE,
    FX_STREAM_INIT_STATE,

    FX_UNDERRUN_STATE = 100,
    FX_OVERRUN_STATE,
    FX_TIMEOUT_STATE,

    FX_ERROR_RELEASE_STATE = 200,
    FX_ERROR_INIT_STATE,
    FX_ERROR_CONNECT_STATE,
    FX_ERROR_DISCONNECT_STATE,
    FX_ERROR_STATE,
    FX_ERROR_PIN_STATE,
    FX_ERROR_MEDIA_PIN_STATE,
    FX_ERROR_STOP_STATE,
    FX_ERROR_PAUSE_STATE,
    FX_ERROR_START_STATE,
    FX_ERROR_RUN_STATE,
    FX_ERROR_INVALID_PARAM,
    FX_ERROR_SUBMEDIA_PIN_STATE,
    FX_ERROR_MEMORY_STATE,

    FX_USER_STATE = 300
} FX_STATE;
```

**Attributes:**

*FX_UNDEFINED_STATE*
        Undefined state.
*FX_LOADING_STATE*
        Fx is loading.
*FX_INIT_STATE*
        Fx is initialized.
*FX_ERROR_INIT_STATE*
        Fx initialization error.
*FX_CONNECT_STATE*
        Fx is connected.
*FX_ERROR_CONNECT_STATE*
        Fx connection error.
*FX_DISCONNECT_STATE*
        Fx is disconnected.
*FX_ERROR_DISCONNECT _STATE*

Fx disconnection error.
*FX_STOP_STATE*
Fx is stopped.
*FX_ERROR_STOP_STATE*
Error on stop command.
*FX_PAUSE_STATE*
Fx is paused.
*FX_ERROR_PAUSE_STATE*
Error on pause command.
*FX_START_STATE*
Fx is started.
*FX_ERROR_START_STATE*
Error on start command.
*FX_RUN_STATE*
Fx is running.
*FX_ERROR_RUN_STATE*
Error on run command.
*FX_IDLE_STATE*
Fx is idle.
*FX_RELEASE_STATE*
Fx is released.
*FX_ERROR_RELEASE_STATE*
Fx release error.
*FX_UNDERRUN_STATE*
Underrun data.
*FX_OVERRUN_STATE*
Overrun data.
*FX_TIMEOUT_STATE*
Timeout state.
*FX_FLUSH_STATE*
Fx flushing is done.
*FX_STREAM_INIT_STATE*
Fx received a new stream format.
*FX_ERROR_STATE*
General error on Fx.
*FX_ERROR_PIN_STATE*
General error on Fx pin.
*FX_ERROR_MEDIA_PIN_STATE*
Invalid media format on Fx pin.
*FX_ERROR_SUBMEDIA_PIN_STATE*
Invalid sub media format on Fx pin.
*FX_PARAM_UPDATE*
Fx parameters are updated.
*FX_ERROR_INVALID_PARAM,*
At least one Fx parameter is invalid
*FX_PIN_UPDATE*
Fx Pins are updated.
*FX_ERROR_MEMORY_STATE*
Error on memory allocation or release.
*FX_USER_STATE*
User state.

**Requirements:**

Header: FxDef.h.


`FX_UNIT_TYPE`

Specifies the FxEngine unit types.

**Syntax:**

```
typedef enum _FX_UNIT_TYPE{
    NOT_DEFINED_TYPE = 0,
    UINT8_TYPE,
    INT8_TYPE,
    UINT16_TYPE,
    INT16_TYPE,
    UINT32_TYPE,
    INT32_TYPE,
    INT64_TYPE,
    UINT64_TYPE,
    FLOAT32_TYPE,
    FLOAT64_TYPE,
    COMPLEX_TYPE
} FX_UNIT_TYPE;
```

**Attributes:**

*UINT8_TYPE*
        Unsigned Integer 8bits.
*INT8_TYPE*
        Signed Integer 8bits.
*UINT16_TYPE*
        Unsigned Integer 16bits.
*INT16_TYPE*
        Signed Integer 16bits.
*UINT32_TYPE*
        Unsigned Integer 32bits
*INT32_TYPE*
        Signed Integer 32bits.
*INT64_TYPE*
        Signed Integer 64bits.
*UINT64_TYPE*
        Unsigned Integer 64bits.
*FLOAT32_TYPE*
        Signed Flotting 32bits.
*FLOAT64_TYPE*
        Signed Flotting 64bits.
*COMPLEX_TYPE*
        Two words of Signed Flotting 32 bits (Real, Imag).

**Requirements:**

Header: FxDef.h.

**FXENGINE_CONST_TYPE**

Specifies the FxEngine constant types.

**Syntax:**

```
typedef enum _FXENGINE_CONST_TYPE{
    FX_STATE_CONST = 0,
    FX_TYPE_CONST,
    UNITTYPE_CONST,
    FXENGINE_ERROR_CONST,
    FX_MAINMEDIATYPE_CONST,
    FX_SUBMEDIATYPE_CONST,
    FX_PINTYPE_CONST,
    FX_SCOPE_CONST
}FXENGINE_CONST_TYPE;
```

**Attributes:**

*FX_STATE_CONST*
>    States of Fx.

*FX_TYPE_CONST*
>    Types of Fx.

UNITTYPE_CONST
>    Unit Type of Fx parameters.

*FXENGINE_ERROR_CONST*
>    Errors of FxEngine.

*FX_MAINMEDIATYPE_CONST*
>    Main Media Types of Fx Media.

*FX_SUBMEDIATYPE_CONST*
>    Sub Media Types of Fx Media.

*FX_PINTYPE_CONST*
>    Pin Types of Fx.

*FX_SCOPE_CONST*
>    Scopes of Fx

**Requirements:**

Header FxDef.h.

---

**FX_PARAMETER**

Specifies the Fx parameters updating mode (see FEF_UpdateFxParam).

**Syntax:**

```
typedef enum _FX_PARAMETER{
    FX_PARAM_ONE = 0,
    FX_PARAM_ALL,
} FX_PARAMETER;
```

**Attributes:**

*FX_PARAM_ONE*
>    Specifies one Fx parameter to update only.

*FX_PARAM_ALL*
>    Specifies all Fx parameters to update.

**Requirements:**

Header FxDef.h.

# 7. CONTACTS

http://www.smprocess.org

Support:

>    support@smprocess.org

SMProcess information:

>    info@smprocess.org