FreeFrame SDK Reference

Copyright Notice

License agreement

Table of Contents

The FreeFrame SDK

The FreeFrame SDK aims at helping developers of FreeFrame plugins by providing a default implementation of most of the functions defined in the FreeFrame specification. Developers can thus concentrate their efforts on the few most important functions determining the external behavior of the plugin, namely *processFrame* and *processFrameCopy* (for more information about the FreeFrame functions see the FreeFrame specification at the end of this document and at www.freeframe.org).

The FreeFrame SDK has been designed and implemented on Windows platforms using Microsoft Visual Studio. It is written in C++ and it is available for both Microsoft Visual Studio 6 and Microsoft Visual Studio .NET. However, it can be easily ported to other development environments and other operating systems (e.g., Linux).

The FreeFrame SDK is distributed as a collection of header files to be included by the header files of the FreeFrame plugins using it, and as a library file to be statically linked in the DLLs implementing the FreeFrame plugins.

The FreeFrame SDK consists of three main C++ classes:

(i) CFFPluginInfo (header file FFPluginInfo.h) which is responsible of managing information about the plugin (e.g., its name, type, version);

(ii) CFFPluginManager (header file FFPluginManager.h) which is responsible of managing the plugin and encapsulate the information needed for implementing the default versions of most of the FreeFrame functions;

(iii) CFreeFramePlugin (header file FFPluginSDK.h), the base class from which all the plugins developed with the FreeFrame SDK should derive.

These three classes will be described in details in the following.

A Wizard is available with the FreeFrame SDK. It provides a template for developing a FreeFrame plugin using the FreeFrame SDK. For developing a plugin you will only need to fill with your own code the methods generated by the Wizard. Currently, the Wizard is available for Microsoft Visual Studio 6 only. The development of a Wizard for Microsoft Visual Studio .NET is planned. However, projects generated by the Wizard can also be compiled in Microsoft Visual Studio .NET.

A sample plugin developed with the FreeFrame SDK is also described.

1 How to install

Here follow some instructions about how installing the FreeFrame SDK and the FreeFrame Wizard. From now on Microsoft Windows and Microsoft Visual Studio 6 or Microsoft Visual Studio .NET are adopted as reference platforms for the discussion.

1.1 Installing the FreeFrame SDK

For installing the FreeFrame SDK download the FreeFrameSDK_SetUp.exe file (available at the InfoMus website: www.infomus.dist.unige.it) and double click on it. Then follows the instructions provided by the installer software. At the end of the installation process have a look at the location where you installed the FreeFrame SDK. You should find the following folders:

- *Binaries*: it contains the compiled version of the FreeFrame Wizard and its help file.
- *Documentation*: it contains this reference manual
- *Include*: it contains the header files of the FreeFrame SDK
- *Library*: it contains the library files of the FreeFrame SDK. Versions for both Microsoft Visual Studio 6 (MSVC6 subfolder) and Microsoft Visual Studio .NET (MSVC7 folder) are provided.
- *Tutorial*: it contains a sample FreeFrame plugin developed with the FreeFrame SDK (both source and compiled code).

In order to use the FreeFrame SDK you have to configure Microsoft Visual Studio properly. Here follow some instructions for doing it both in Microsoft Visual Studio 6 and in Microsoft Visual Studio .NET.

Microsoft Visual Studio 6

- Open Microsoft Visual Studio 6
- In the *Tools* menu select *Options*
- Choose the *Directories* property page
- Click on the *New* button and you will be prompted to select a directory: find the Include subdirectory of the FreeFrame SDK directory and select it (see Figure 1.1a). In this way the FreeFrame SDK header files will be available in all your projects.
- Change the *Show directories for* combo box from “Include files” to “Library files” and then proceed as before. Select at this step the MSVC6 subdirectory in the Library subdirectory of the FreeFrame SDK directory (see Figure 1.1b).

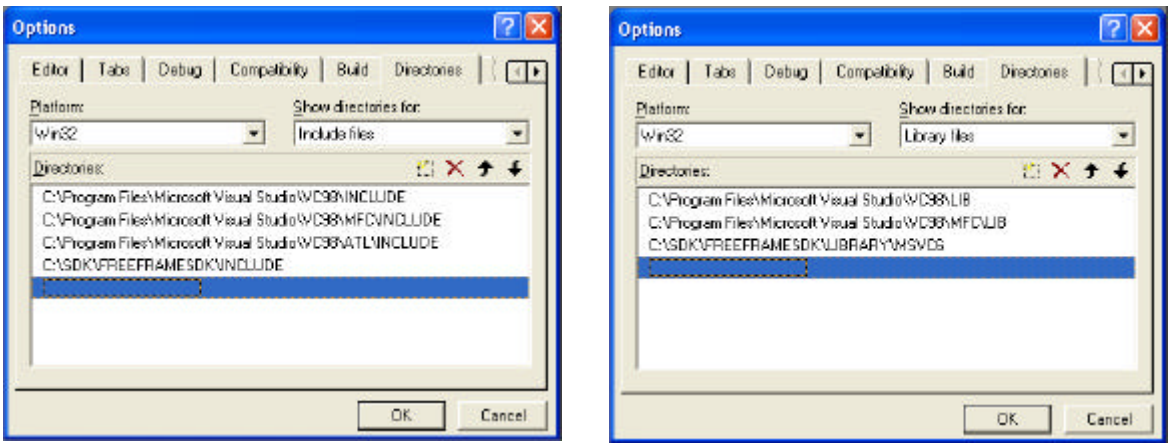
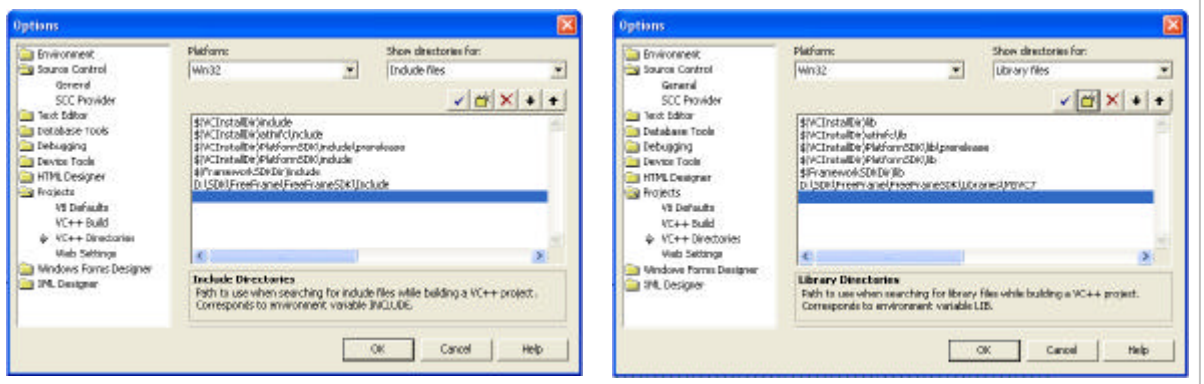


Figure 1.1: The *Directories* property page in Microsoft Visual Studio 6 (a) when “Include files” is selected in the *Show directories for* combo box and (b) when “Library files” is selected in the same combo box.

Microsoft Visual Studio .NET

- Open Microsoft Visual Studio .NET
- In the *Tools* menu select *Options*
- In the folders displayed on the left select *Projects* and choose VC++ *Directories*.
- In the *Show directories for* combo select “Include files”

- Click on the *New* button and you will be prompted to select a directory: find the Include subdirectory of the FreeFrame SDK directory and select it (see Figure 1.2a). In this way the FreeFrame SDK header files will be available in all your projects.
- Change the *Show directories for* combo box from “Include files” to “Library files” and then proceed as before. Select at this step the MSVC7 subdirectory in the Library subdirectory of the FreeFrame SDK directory (see Figure 1.2b).



(a) (b)

Figure 1.2: The *VC++ Directories* property page in Microsoft Visual Studio .NET (a) when “Include files” is selected in the *Show directories for* combo box and (b) when “Library files” is selected in the same combo box.

The Microsoft Visual Studio environment will save these modifications upon exiting the program, so quit the IDE and open it again if you want to be sure that your changes will have been saved for all your future projects.

1.2 Installing the FreeFrame Wizard

The FreeFrame Wizard is currently available only for Microsoft Visual Studio 6. In order to install it on your computer you have to copy the FreeFrameWizard.awx file and its help file to the C:\Program files\Microsoft Visual Studio\Common\MsDev98\Template folder of a standard installation of Microsoft Visual Studio (or in the corresponding folder in case of a custom installation). You can find the FreeFrameWizard.awx file and its help file in the Binaries folder of the FreeFrame SDK installation.

2 The FreeFrame Wizard

The FreeFrame Wizard provides a template for developing a FreeFrame plugin using the FreeFrame SDK. For developing a plugin you will only need to fill with your own code the methods generated by the Wizard. Currently, the Wizard is available for Microsoft Visual Studio 6 only. The development of a Wizard for Microsoft Visual Studio .NET is planned. However, projects generated by the Wizard can also be compiled in Microsoft Visual Studio .NET.
For running the FreeFrame Wizard select *New* in the *File* menu of Microsoft Visual Studio 6. Then choose the *Projects* page. If the Wizard has been correctly installed it should appear as "FreeFramePlugin AppWizard". Select it, type the name of the new project, select the directory for the new project, and press *OK*. The FreeFrame Wizard consists of three or four steps depending on the kind of plugin you are going to develop. The Wizard Steps and its output are discussed in the following.

2.1 Step 1

The first step in making a new FreeFrame plugin consists in indicating the plugin name and its type, i.e., whether it is a source or an effect plugin.

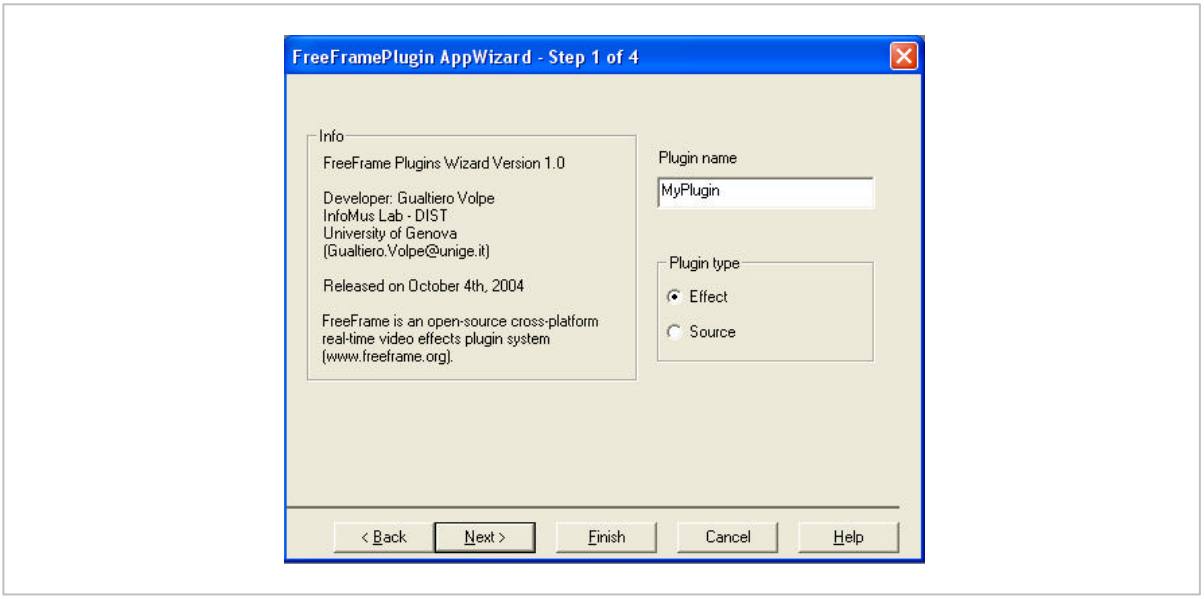


Figure 2.1: The FreeFrame Wizard, Step 1

Figure 2.1 here above displays the dialog box for the first step of the FreeFrame Wizard. Use it:

- To set the name of your plugin (*Plugin name*)
- To indicate whether your plugin is a source or an effect plugin (*Plugin type*).

Plugin name: write here the name of your FreeFrame plugin. Notice that it should be at most 16 characters long. Longer names are truncated at the 16th character.

Plugin type: according to the FreeFrame standard, two types of plugins exist: source and effect plugins. "Plugins of PluginType effect are passed frames of video, which they then modify. Source plugins are simply passed a pointer where they paint frames of video. One example of a source plugin would by a visual synthesizer which uses the parameters to synthesize video" (FreeFrame specification, version 1.0 - 03). Indicate here whether your plugin is a source or an effect plugin.

2.2 Step 2

The second step of the FreeFrame Wizard allows you to specify the properties of the plugin you are going to develop. These include general information about the plugin, its version, its developer, and possible copyright information, the image formats the plugin will support, the supported processing modes and optimizations.
Figure 2.2 shows the dialog box for the second step of the FreeFrame Wizard.

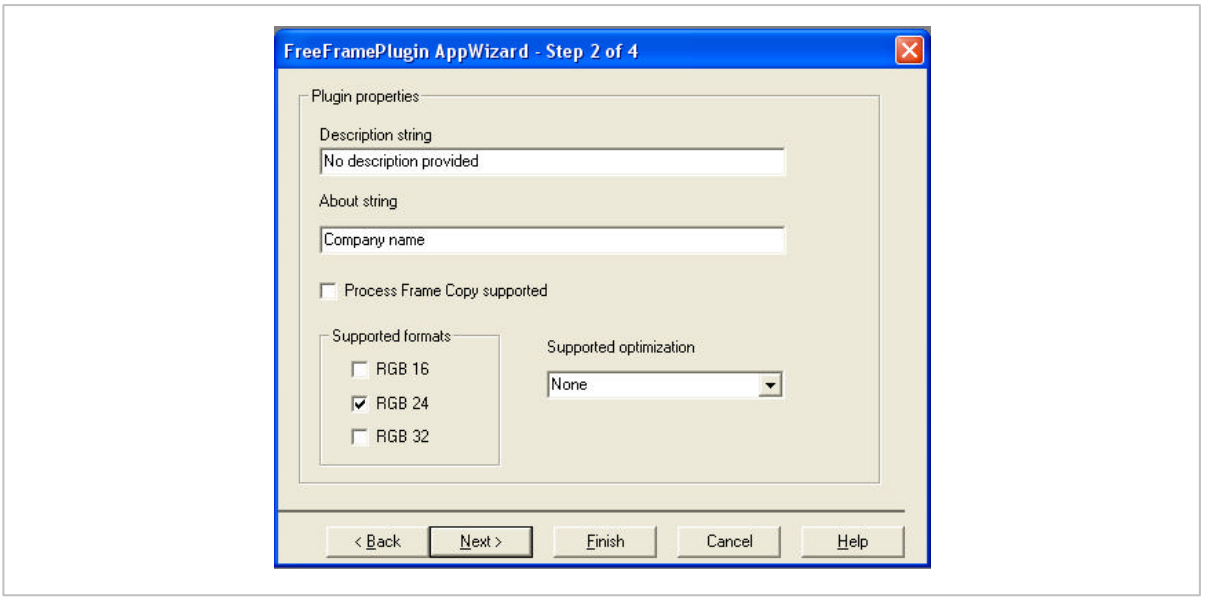


Figure 2.2: The FreeFrame Wizard, Step 2

Use this dialog box to specify the main properties of you plugin. In particular:

- To set a description string for your plugin (*Description string*)
- To set an about string (*About string*)
- To indicate whether your plugin supports Process Frame Copy working mode (*Process Frame Copy supported*)
- To specify which image formats are supported by your plugin (*Supported formats*)
- To specify whether your plugin is optimized for a specific working mode (*Supported optimization*).

Description string: a brief description of your plugin. It should say in a few understandable words what your plugin does.

About string: provide here information about who is making the plugin (e.g., the names of the developers and their company if needed, possible copyright information).

Process Frame Copy supported: check it if your plugin can work in Process Frame Copy mode. By default FreeFrame plugins work “in place”, i.e., they directly operate on the input frame and return the same modified frame as output. When a plugin works in Process Frame Copy mode instead, input and output frames are two separate buffers and parts of the input frame may need to be copied into the output frame. Moreover, Process Frame Copy mode allows multiple inputs. Note that effect plugins must support in place processing, i.e., if you decide to support Process Frame Copy mode, you will have to implement both the in place and the Process Frame Copy processing. Source plugins should not support Process Frame Copy mode since they should not have any input: therefore, if your plugin is a source plugin this option is automatically disabled.

Supported formats: indicate here which image formats your plugin supports. Three options are currently available in the FreeFrame Specification: RGB16 (16-bit, 5-6-5 packed), RGB24 (24-bit, packed), RGB32 (32-bit, also suitable for 32-bit unsigned integer aligned 24-bit). The byte order of color values depends on the target platform (see FreeFrame Specification). You may support more than one format, but you must support at least one of them.

Supported optimization: indicate here whether your plugin is optimized for a specific kind of processing, i.e., whether it is optimized for in place processing (“Process Frame” option), or for Process Frame Copy processing (“Process Frame Copy” option), or for both (“Both” option), or none (“None” option) of them. If *Process Frame Copy supported* is not checked only two options (“None” and “Process Frame”) are available.

2.3 Step 3

Step 3 is needed only if your plugin will support Process Frame Copy mode, i.e., if at Step 2 you checked the *Process Frame Copy supported* checkbox. In case your plugin only supports in place processing, you will directly move to Step 4. Notice, however, that in such case the dialog box here referred as "Step 4" will appear in MS Visual Studio as "Step 3 of 3" since Microsoft Visual Studio provides a progressive numbering of the dialog boxes that are actually displayed.
Step 3 is responsible of gathering information about the inputs of your plugin. Its dialog box is displayed in Figure 2.3. You will use it:

- To specify how many inputs your plugin can accept (*Minimum number of inputs* and *Maximum number of inputs*)
- To indicate whether the Wizard should provide a template for a custom implementation of the getInputStatus function (*Provide a custom implementation of the getInputStatus function*)

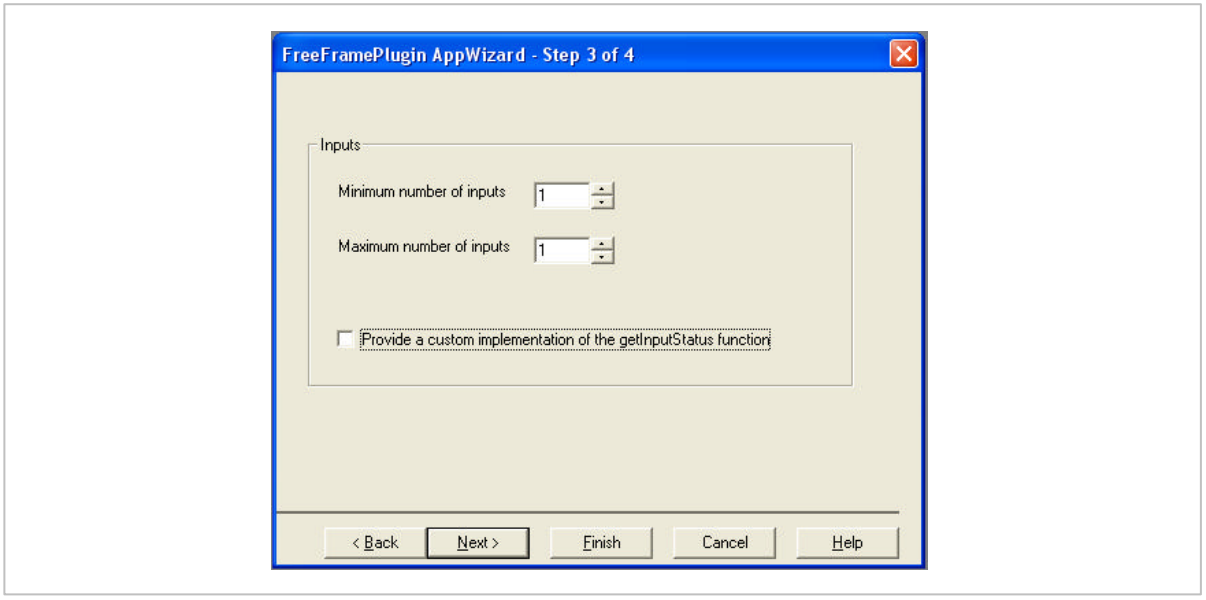


Figure 2.3: The FreeFrame Wizard, Step 3

Minimum number of inputs: indicate here the minimum number of inputs your plugin requires (i.e., the host must provide). These inputs are supposed to be always in use.

Maximum number of inputs: indicate here the maximum number of inputs your plugin can handle. Of course, it must be *Minimum number of inputs* <= *Maximum number of inputs*.

Provide a custom implementation of the getInputStatus function: the FreeFrame SDK provides a default implementation of the getInputStatus function. In such default implementation, all the inputs are always considered to be in use. However, the inputs in the range [*Minimum input frames*, *Maximum input frames*) may be not in use and you may need to let the host know this. In this case you have to provide a custom implementation of the getInputStatus function. If you check this option the Wizard will produce a template for your custom getInputStatus function.

In case of effect plugins supporting only in place processing both *Minimum number of inputs* and *Maximum number of inputs* are automatically set to 1. No custom implementation of getInputStatus is needed.

2.4 Step 4

The last Step of the FreeFrame Wizard concerns plugin parameters and their properties. Note that in case your plugin only supports in place processing this step will be indicated in Microsoft Visual Studio as "Step 3 of 3".
The dialog box for Step 4 is displayed in Figure 2.4. Use this dialog:

- To specify the parameters of your plugin, their type, and their default values

- To indicate whether the Wizard should provide a template for a custom implementation of the getParameterDisplay function (*Provide a custom implementation of the getParameterDisplay function*).

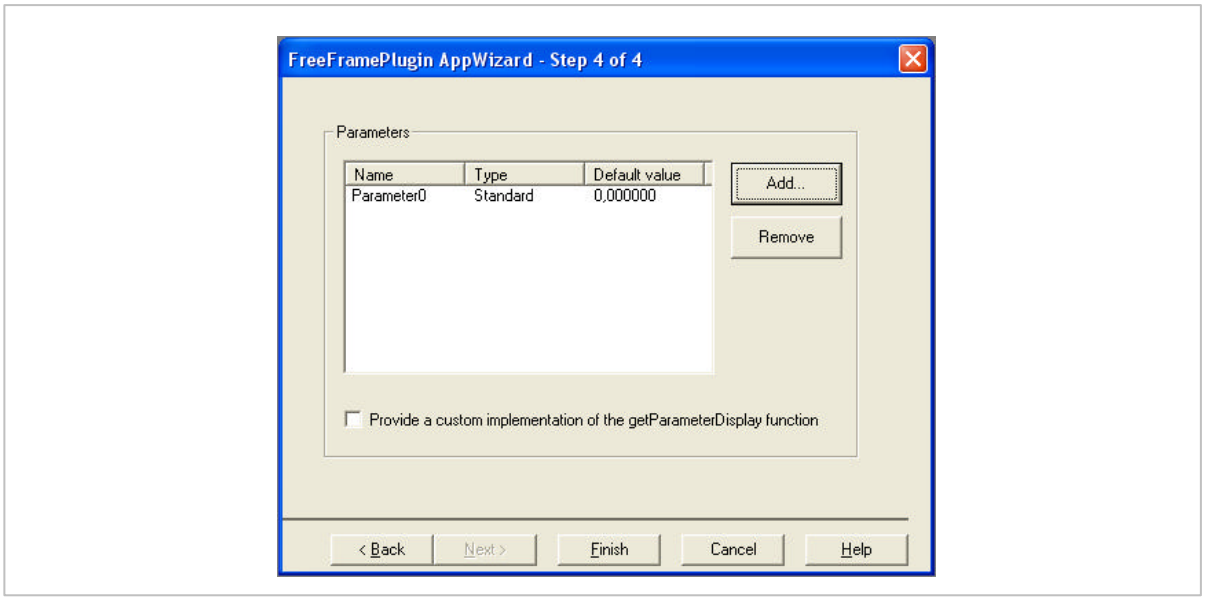


Figure 2.4: The FreeFrame Wizard, Step 4

For adding a new parameter press the add button. The Wizard will open a dialog box including the following fields: *Parameter name*, *Parameter type*, *Default value*, and *Text value*. Such dialog box is shown in Figure 2.5.

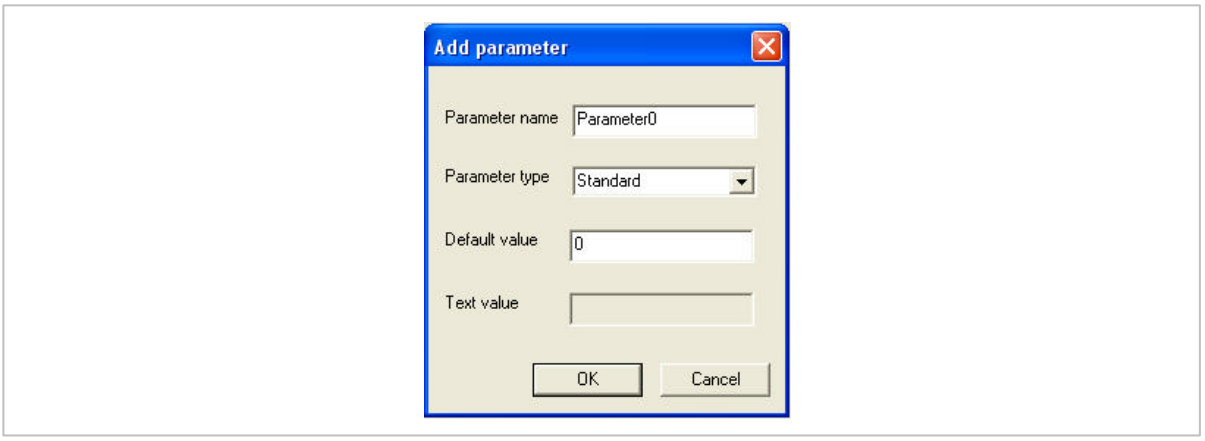


Figure 2.5: The FreeFrame Wizard, Step 4: the "Add Parameter" dialog box

Parameter name: type here the name of the new parameter. Notice that it should be at most 16 characters long. Longer names are truncated at the 16th character. This name will be displayed in the host UI. A private member variable with this name will be added to the class implementing your plugin.

Parameter type: specify here the parameter type. The following options are currently available in the FreeFrame specification: Boolean, Event, Red, Green, Blue, XPos, YPos, Standard, and Text (see FreeFrame Specification 1.0 – 03). Except for text parameters, parameters are always represented as 32-bit floating-point values ranging in [0, 1]. The FreeFrame SDK keeps this convention with the only exception of Boolean and Event parameters that are internally represented as Booleans (and then converted to float when needed). For text parameters the 32 bits represent a pointer to a null terminated string. The FreeFrame SDK internally represents them as char*.

Default value: provide here a default value for the parameter. It must be a floating-point value in the range [0, 1]. If the parameter type is Boolean or Event, whatever value is accepted with the (usual) following convention: 0 corresponds to False or Off, any other value corresponds to True or On. The FreeFrame SDK will convert

True to 1.0 and False to 0.0 when needed. In case the parameter type is Text this field is disabled and you have to use the *Text value* field instead.

Text value: this field is enabled only if the parameter type is set to text. Type here the string that has to be used as default value for the text parameter.

Provide a custom implementation of the getParameterDisplay function: the FreeFrame SDK provides a default implementation of the getParameterDisplay function. In such default implementation, floating-point values of parameters are converted for displaying needs in a string representing the same numbers (i.e., the digits of a parameter value are considered as characters composing its display string). However, you may need to modify this default behavior by providing your own display strings. In this case, you have to provide a custom implementation of the getParameterDisplay function. If you check this option the Wizard will produce a template for your custom getParameterDisplay function.

You can remove an already included parameter by selecting it in the list and pressing the Remove button.

2.5 FreeFrame Wizard Output

At the end of the three or four steps described above, the FreeFrame Wizard creates a project for Microsoft Visual Studio 6 that you can use for making a new FreeFrame plugin using the FreeFrame SDK. The output of the Wizard is just the starting point for making a new FreeFrame plugin. Now you have to fill with your own code the methods the Wizard have prepared for you.
You will find the following files in the newly crated project:

- A .dsw file having the same name of the Microsoft Visual Studio project (the one you have selected in the *New* dialog of the *File* menu). This file (the project workspace file) contains information on the contents and organization of the project workspace. You don't have to modify it.
- A .dsp file having the same name of the project too. This file (the project file) contains information at the project level and is used to build a single project or subproject. You don't have to modify it.
- A .opt file. This file (the workspace options file) contains the workspace settings that you specify in the Project Settings dialog. These settings specify the appearance of the project workspace using your hardware and configuration. This binary file is automatically generated when you open the .dsw or .dsp file in the IDE. You should not share the .opt file, because it contains information specific to your computer.
- A .ncb file. It provides information on the NCB (No Compile Browse) parser, the mechanism that updates ClassView and WizardBar. This is a binary file that is generated automatically and should not be shared. Both the .opt and the .ncb files have the same name of the project.
- A .h file having the same name of the project. This file is the main header file for your plugin. It includes the declaration of the class encapsulating your plugin. Such class is a subclass of the CFreeFramePlugin class of the FreeFrame SDK (you can find more information about it in the next Chapter of this manual).
- A .cpp file having the same name of the project. This file is the main source file. It contains the implementation of the methods of the plugin subclass declared in the .h file. You will have to fill such methods with your own code in order to implement your plugin. Browse the file and look for the TO DO comments. Usually you will have to fill either the ProcessFrame method or both the ProcessFrame and ProcessFrameCopy methods.
- A .def file specifying the methods exported by the DLL implementing your plugin, namely the plugMain method. Usually, you don't have to modify it.
- The FFPluginInfoData.cpp file. This file includes the DllMain function of the DLL implementing your plugin. It also declares a global variable used by the FreeFrame SDK for keeping information about the main features of your plugin. Usually, you don't have to modify it.

In Chapter 4 an example is provided showing how to use the FreeFrame Wizard and the FreeFrame SDK to make a sample FreeFrame plugin.

3 FreeFrame SDK Class Reference

3.1 CFFPluginInfo

CFFPluginInfo manages static information concerning a plugin name, version, and description.

```
#include <FFPluginInfo.h>
```

```
class CFFPluginInfo {

public:

	CFFPluginInfo(
		FPCREATEINSTANCE* pCreateInstance,
		const char* pchUniqueID,
		const char* pchPluginName,
		DWORD dwAPIMajorVersion,
		DWORD dwAPIMinorVersion,
		DWORD dwPluginMajorVersion,
		DWORD dwPluginMinorVersion,
		DWORD dwPluginType,
		const char* pchDescription,
		const char* pchAbout,
		DWORD dwFreeFrameExtendedDataSize = 0,
		const void* pFreeFrameExtendedDataBlock = NULL
	);

	~CFFPluginInfo();

	const PluginInfoStruct* GetPluginInfo() const;

	const PluginExtendedInfoStruct* GetPluginExtendedInfo() const;

	FPCREATEINSTANCE* GetFactoryMethod() const;

private:

	// Structures containing information about the plugin
	PluginInfoStruct m_PluginInfo;
	PluginExtendedInfoStruct m_PluginExtendedInfo;

	// Pointer to the factory method of the plugin subclass
	FPCREATEINSTANCE* m_pCreateInstance;
};
```

The CFFPluginInfo class manages static information related to a FreeFrame plugin. Examples are the name of the plugin, its unique identifier, its type (either source or effect), the current version, the version of the FreeFrame API the plugin refers to, a short description of the plugin, information about the developer(s) and possible copyright. In other words, this class stores the information required by the FreeFrame getInfo and getExtendedInfo global functions.
The CFFPluginInfo class is also involved in the process of creating an instance of the subclass implementing a plugin: it stores a pointer to the factory method of the plugin subclass, which is called when the plugin object needs to be instantiated. The FreeFrame SDK keeps a prototype instance of the plugin in order to be able to access information on the plugin at any time. The effectively working instance is created at the time the plugin is instantiated by the host.

3.1.1 Public Member Functions

3.1.1.1 CFFPluginInfo::CFFPluginInfo

```
CFFPluginInfo(
        FPCREATEINSTANCE* pCreateInstance,
        const char* pchUniqueID,
        const char* pchPluginName,
        DWORD dwAPIMajorVersion,
        DWORD dwAPIMinorVersion,
        DWORD dwPluginMajorVersion,
        DWORD dwPluginMinorVersion,
        DWORD dwPluginType,
        const char* pchDescription,
        const char* pchAbout,
        DWORD dwFreeFrameExtendedDataSize = 0,
        const void* pFreeFrameExtendedDataBlock = NULL
);
```

Parameters

pCreateInstance	A pointer to the factory method of the subclass implementing the plugin.
pchUniqueID	A string representing the unique identificator of the plugin. According to the FreeFrame specification, it must be a not null terminated string of 4 1-byte ASCII characters. Longer strings will be truncated at the 4th character.
pchPluginName	A string containing the name of the plugin. According to the FreeFrame specification, it must be a not null terminated string of 16 1-byte ASCII characters. Longer strings will be truncated at the 16th character.
dwAPIMajorVersion	The major version number of the FreeFrame API employed by the plugin. It is the number before the decimal point in the API version number.
dwAPIMinorVersion	The minor version number of the FreeFrame API employed by the plugin. It is the number after the decimal point in the API version number.
dwPluginMajorVersion	The major version number of the plugin. It is the number before the decimal point in the plugin version number.
dwPluginMinorVersion	The minor version number of the plugin. It is the number after the decimal point in the plugin version number.
dwPluginType	The type of the plugin. According to the FreeFrame specification, it should be 0 in case of effect plugins and 1 in case of source plugins.
pchDescription	A string providing a short description of what the plugin does.
pchAbout	A string providing information on the developer(s) of the plugin, their possible company, and possible copyright information.
dwFreeFrameExtendedDataSize	Size in bytes of the FreeFrame ExtendedDataBlock, or 0 if not provided by plugin. Extended Data Blocks are not yet exploited in the current version of

	the FreeFrame specification (1.0). Therefore, at the moment the default value (0) should be used for this parameter.
pFreeFrameExtendedDataBlock	32-bit pointer to a FreeFrame ExtendedDataBlock, Extended Data Bloks are not yet exploited by the FreeFrame specification version 1.0. Therefore, at the moment the default value (NULL) should be used for this parameter.

Remarks

The constructor of CFFPluginInfo receives through its parameters the information that needs to be stored.

3.1.1.2 CFFPluginInfo:: ~CFFPluginInfo

```
~CFFPluginInfo();
```

The standard destructor of CFFPluginInfo.

3.1.1.3 CFFPluginInfo::GetPluginInfo

```
const PluginInfoStruct* GetPluginInfo() const;
```

Parameters

None

Return value

A pointer to a PluginInfoStruct containing information on the plugin. For further information on the definition of PluginInfoStruct see the header file FreeFrame.h and the FreeFrame specification version 1.0.

Remarks

This method returns a pointer to a PluginInfoStruct as defined in FreeFrame.h. Such structure contains information on the plugin name and type, its unique identifier, and the version of the FreeFrame API it uses.

3.1.1.4 CFFPluginInfo::GetPluginExtendedInfo

```
const PluginExtendedInfoStruct* GetPluginExtendedInfo() const;
```

Parameters

None

Return value

A pointer to a PluginExtendedInfoStruct containing information on the plugin. . For further information on the definition PluginExtendedInfoStruct see the header file FreeFrame.h and the FreeFrame specification version 1.0.

Remarks
This method returns a pointer to a PluginExtendedInfoStruct (for further information see FreeFrame.h and the FreeFrame specification). A PluginExtendedInfoStruct contains information on the plugin version, a short description of the plugin, and information about the developer(s) and possible copyright issues.

3.1.1.5 CFFPluginInfo::GetFactoryMethod

```
typedef DWORD __stdcall FPCREATEINSTANCE(void**);

FPCREATEINSTANCE* GetFactoryMethod() const;
```

Parameters

None

Return value

A pointer to the factory method of the plugin subclass.

Remarks

This method returns a pointer to the factory method of the subclass implementing the plugin. It is called by the FreeFrame SDK when creating a new instance of the plugin.

3.2 CFFPluginManager

CFFPluginManager manages information concerning a plugin inputs, parameters, and capabilities.

```
#include <FFPluginManager.h>
```

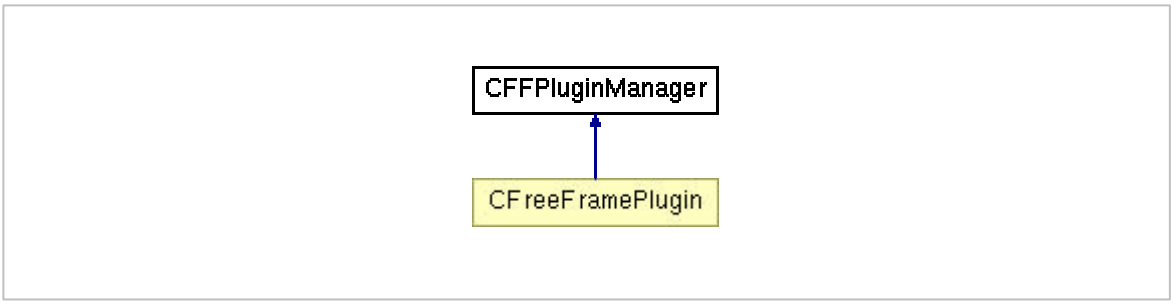


Figure 3.1: Inheritance diagram for CFFPluginManager

```
class CFFPluginManager {

public:

	enum FormatsFlags {
		FF_RGB_16 = 0x1,
		FF_RGB_24 = 0x2,
		FF_RGB_32 = 0x4
	 };

	enum OptimizationFlags {
```

```
        FF_OPT_NONE             = 0x1,
        FF_OPT_INPLACE          = 0x2,
        FF_OPT_COPY             = 0x4,
        FF_OPT_BOTH             = 0x8
    };

    virtual ~CFFPluginManager();

    bool IsProcessFrameCopySupported() const;

    DWORD GetSupportedFormat() const;

    DWORD GetSupportedOptimization() const;

    int GetMinInputs() const;

    int GetMaxInputs() const;

    int GetNumParams() const;

    char* GetParamName(DWORD dwIndex) const;

    DWORD GetParamType(DWORD dwIndex) const;

    void* GetParamDefault(DWORD dwIndex) const;

    void SetVideoInfo(const VideoInfoStruct* pVideoInfo);

protected:

    CFFPluginManager();

    void SetProcessFrameCopySupported(bool bIsSupported);

    void SetSupportedFormats(DWORD dwFlags);

    void SetSupportedOptimizations(DWORD dwFlags);

    void SetMinInputs(int iMinInputs);

    void SetMaxInputs(int iMaxInputs);

    void SetParamInfo(  DWORD dwIndex,
                        const char* pchName,
                        DWORD dwType,
```

```
                                float fDefaultValue );

        void SetParamInfo(      DWORD dwIndex,
                                const char* pchName,
                                DWORD dwType,
                                bool bDefaultValue);

        void SetParamInfo(      DWORD dwIndex,
                                const char* pchName,
                                DWORD dwType,
                                const char* pchDefaultValue);

        int GetFrameWidth() const;

        int GetFrameHeight() const;

        DWORD GetFrameDepth() const;

        DWORD GetFrameOrientation() const;

private:

        // Structure for keeping information about each plugin parameter
        typedef struct ParamInfoStruct{
                DWORD ID;
                char Name[16];
                DWORD dwType;
                float DefaultValue;
                char* StrDefaultValue;
                ParamInfoStruct* pNext;
        } ParamInfo;

        // Information on paramters and pointers to ParamInfo list
        int m_NParams;
        ParamInfo* m_pFirst;
        ParamInfo* m_pLast;

        // Plugin capabilities
        bool m_bIsProcessFrameCopySupported;
        DWORD m_dwSupportedFormats;
        DWORD m_dwSupportedOptimizations;

        // Inputs
       int m_iMinInputs;
       int m_iMaxInputs;
```

```
        // Information on the incoming images
        VideoInfoStruct m_VideoInfo;
};
```

The CFFPluginManager class is the base class for FreeFrame plugins developed with the FreeFrame SDK since it provides them with methods for automatically manage information concerning plugin inputs, paramaters, and capabilities. Examples of information managed by this class are the number of inputs and parameters of a plugin; the name, type and default value of each parameter; the image formats a plugin supports; the supported optimizations.
Plugins developed with the FreeFrame SDK (and thus having this class as base class) should call the protected methods of this class in order to specify the information related to their inputs, parameters and capabilities. These calls are usually done while constructing the plugin subclass. Plugins subclasses should also call methods of this class in order to get information about the images they are going to process (i.e., their width, height, depth, orientation).
The defualt implementations of the FreeFrame gloabal functions call the public methods of this class in order to return to the host information about a plugin inputs, parameters, and capabilities.

3.2.1 Public Types

3.2.1.1 Enum CFFPluginManager::FormatsFlags

```
    enum FormatsFlags {
            FF_RGB_16 = 0x1,
            FF_RGB_24 = 0x2,
            FF_RGB_32 = 0x4
     };
```

CFFPluginManager::FormatsFlags enumerates flags indicating which image formats a plugin supports. According to the FreeFrame specification three image formats are allowed: RGB16, RGB24, RGB32. The values of this enum are used to exchange information with the host on the formats a plugin supports. A plugin may support more than one image format.

Enumeration values

FF_RGB_16	The plugin supports RGB16 image format.
FF_RGB_24	The plugin supports RGB24 image format.
FF_RGB_32	The plugin supports RGB32 image format.

3.2.1.2 Enum CFFPluginManager::OptimizationFlags

```
    enum OptimizationFlags {
            FF_OPT_NONE         = 0x1,
            FF_OPT_INPLACE      = 0x2,
            FF_OPT_COPY         = 0x4,
            FF_OPT_BOTH         = 0x8
     };
```

CFFPluginManager::OptimizationFlags enumerates flags indicating which optimizations a plugin supports. According to the FreeFrame specification a plugin may be optimized either for in place processing, or for ProcessFrameCopy processing, or for both or none of them. Values of this enum are used to exchange information with the host on the optimizations a plugin supports.

Enumeration values

FF_OPT_NONE	No particular optimization is supported by this plugin.
FF_OPT_INPLACE	This plugin is optimized for in place processing.
FF_OPT_COPY	This plugin is optimized for ProcessFrameCopy processing.
FF_OPT_BOTH	This plugin is optimized for both kinds of processing.

3.2.2 Public Member Functions

3.2.2.1 CFFPluginManager:: ~CFFPluginManager

```
virtual ~CFFPluginManager();
```

The standard destructor of CFFPluginManager.

3.2.2.2 CFFPluginManager::IsProcessFrameCopySupported

```
bool IsProcessFrameCopySupported() const;
```

Parameters

None

Return value

True if the plugin supports ProcessFrameCopy mode, false if only in place working mode is supported.

Remarks

This method is called to know if the plugin supports ProcessFrameCopy working mode. It is usually called by the default implementations of the FreeFrame global functions.

3.2.2.3 CFFPluginManager::GetSupportedFormat

```
DWORD GetSupportedFormat() const;
```

Parameters

None

Return value

A combination of flags as defined by the CFFPluginManager::FormatsFlags enumeration.

Remarks

This method is called to know which formats the plugin supports. It is usually called by the default implementations of the FreeFrame global functions. Its return value should be checked against the flags defined by the CFFPluginManager::FormatsFlags enumeration.

3.2.2.4 CFFPluginManager::GetSupportedOptimization

```
DWORD GetSupportedOptimization() const;
```

Parameters

None

Return value

A combination of flags as defined by the CFFPluginManager::OptimizationFlags enumeration.

Remarks

This method is called to know which optimizations the plugin supports. It is usually called by the default implementations of the FreeFrame global functions. Its return value should be cheked agianst the flags defined by the CFFPluginManager::OptimizationFlags enumeration.

3.2.2.5 CFFPluginManager::GetMinInputs

```
int GetMinInputs() const;
```

Parameters

None

Return value

The minimum number of inputs the host must provide.

Remarks

This method returns the minimum number of inputs the host must provide. It is usually called by the default implementations of the FreeFrame global functions.

3.2.2.6 CFFPluginManager::GetMaxInputs

```
int GetMaxInputs() const;
```

Parameters

None

Return value

The maximum number of inputs the plugin can receive.

Remarks

This method returns the maximum number of inputs the plugin can receive. It is usually called by the default implementations of the FreeFrame global functions.

3.2.2.7 CFFPluginManager::GetNumParams

```
int GetNumParams() const;
```

Parameters

None

Return value

The number of parameters of the plugin.

Remarks

This method returns how may parameters the plugin has. It is usually called by the default implementations of the FreeFrame global functions.

3.2.2.8 CFFPluginManager::GetParamName

```
char* GetParamName(DWORD dwIndex) const;
```

Parameters

dwIndex The index of the plugin parameter whose name is queried. It should be in the range [0, Number of plugin parameters).

Return value

The name of the plugin parameter whose index is passed to the method. The return value is a pointer to an array of 16 1-byte ASCII characters, not null terminated (see FreeFrame specification). NULL is returned on error.

Remarks

This method returns the name of the plugin parameter whose index is passed as parameter to the method. It is usually called by the default implementations of the FreeFrame global functions.

3.2.2.9 CFFPluginManager::GetParamType

```
DWORD GetParamType(DWORD dwIndex) const;
```

Parameters

dwIndex The index of the plugin parameter whose type is queried. It should be in the range [0, Number of plugin parameters).

Return value

The type of the plugin parameter whose index is passed as parameter to the method. Codes for allowed parameter types are defined in FreeFrame.h. In case of error, FF_FAIL is returned.

Remarks

This method is called to know the type of the plugin parameter whose index is passed as parameter to the method. It is usually called by the default implementations of the FreeFrame global functions.

3.2.2.10 CFFPluginManager::GetParamDefault

```
void* GetParamDefault(DWORD dwIndex) const;
```

Parameters

dwIndex The index of the plugin parameter whose default value is queried. It should be in the range [0, Number of plugin parameters).

Return value

The default value of the plugin parameter whose index is passed as parameter to the method. The return value should be cast either to a char* in case of text parameters or to a float* in any other case. In case of error, NULL is returned.

Remarks

This method is called to get the default value of the plugin parameter whose index is passed as parameter to the method. It is usually called by the default implementations of the FreeFrame global functions.

3.2.2.11 CFFPluginManager::SetVideoInfo

```
void SetVideoInfo(const VideoInfoStruct* pVideoInfo);
```

Parameters

pVideoInfo A pointer to a VideoInfoStruct (see definition in FreeFrame.h) containing information about the width, height, depth, and orientation of the images the plugin is going to receive.

Return value

None

Remarks

This method is called to provide the plugin with information about the images it is going to process. It is usually called by the default implementations of the FreeFrame global functions once the host communicated the format of the images that have to be processed.

3.2.3 Protected Member Functions

3.2.3.1 CFFPluginManager::CFFPluginManager

```
CFFPluginManager();
```

The standard constructor of CFFPluginManager. Notice that the CFFPluginManager constructor is a protected member function, i.e., nor CFFPluginManager objects nor CFreeFramePlugin objects should be created directly, but only objects of the subclasses implementing specific plugins should be instantiated.

3.2.3.2 CFFPluginManager::SetProcessFrameCopySupported

```
void SetProcessFrameCopySupported(bool bIsSupported);
```

Parameters

bIsSupported The plugin subclass should set it either to true if ProcessFrameCopy mode is supported, or to false if only in place processing is supported.

Return value

None

Remarks

This method is called by a plugin subclass, derived from this class, to indicate whether ProcessFrameCopy mode is supported. This method is usually called when a plugin object is instantiated (i.e., in the plugin subclass constructor).

3.2.3.3 CFFPluginManager::SetSupportedFormats

```
void SetSupportedFormats(DWORD dwFlags);
```

Parameters

dwFlags The plugin subclass should set this parameter to a combination of the flags defined by the CFFPluginManager::FormatsFlags enum. More than one image format may be supported.

Return value

None

Remarks

This method is called by a plugin subclass to indicate which image formats the plugin supports. This method is usually called when a plugin object is instantiated (i.e., in the plugin subclass constructor).

3.2.3.4 CFFPluginManager::SetSupportedOptimizations

```
void SetSupportedOptimizations(DWORD dwFlags);
```

Parameters

dwFlags The plugin subclass should set this parameter to a combination of the flags defined by the CFFPluginManager::OptimizationFlags enumeration.

Return value

None

Remarks

This method is called by a plugin subclass, derived from this class, to indicate which optimizations the plugin supports. This method is usually called when a plugin object is instantiated (i.e., in the plugin subclass constructor).

3.2.3.5 CFFPluginManager::SetMinInputs

```
void SetMinInputs(int iMinInputs);
```

Parameters

iMinInputs The plugin subclass should set it to the minimum number of inputs the host must provide.

Return value

None

Remarks

This method is called by a plugin subclass, derived from this class, to indicate the minimum number of inputs the host must provide. This method is usually called when a plugin object is instantiated (i.e., in the plugin subclass constructor).

3.2.3.6 CFFPluginManager::SetMaxInputs

```
void SetMaxInputs(int iMaxInputs);
```

Parameters

iMaxInputs The plugin subclass should set it to the maximum number of inputs the plugin can receive.

Return value

None

Remarks

This method is called by a plugin subclass, derived from this class, to indicate the maximum number of inputs the plugin can receive. This method is usually called when a plugin object is instantiated (i.e., in the plugin subclass constructor).

3.2.3.7 CFFPluginManager::SetParamInfo (float version)

```
void SetParamInfo(  DWORD dwIndex,
                    const char* pchName,
                    DWORD dwType,
                    float fDefaultValue );
```

Parameters

dwIndex	Index of the plugin parameter whose data are specified. It should be in the range [0, Number of plugin parameters).
pchName	A string containing the name of the plugin parameter. According to the FreeFrame specification it should be at most 16 1-byte ASCII characters long. Longer strings will be truncated at the 16th character.
dwType	The type of the plugin parameter. Codes for allowed types are defined in FreeFrame.h.
fDefaultValue	The default value of the plugin parameter. According to the FreeFrame specification it must be a float in the range [0, 1].

Return value

None

Remarks

This method is called by a plugin subclass, derived from this class, to specify name, type, and default value of the plugin parameter whose index is passed as parameter to the method. This method is usually called when a plugin object is instantiated (i.e., in the plugin subclass contructor). This version of the SetParamInfo function (DefaultValue of type float) should be called for all types of plugin parameters except for text, boolean, and event parameters.

3.2.3.8 CFFPluginManager::SetParamInfo (bool version)

```
void SetParamInfo(  DWORD dwIndex,
                    const char* pchName,
                    DWORD dwType,
                    bool bDefaultValue);
```

Parameters

dwIndex	Index of the plugin parameter whose data are specified. It should be in the range [0, Number of plugin parameters).
pchName	A string containing the name of the plugin parameter. According to the FreeFrame specification it should be at most 16 1-byte ASCII characters long. Longer strings will be cut at the 16th character.
dwType	The type of the plugin parameter. Codes for allowed types are defined in FreeFrame.h.
bDefaultValue	The boolean default value of the plugin parameter.

Return value

None

Remarks

This method is called by a plugin subclass, derived from this class, to specify name, type, and default value of the plugin parameter whose index is passed as parameter to the method. This method is usually called when a plugin object is instantiated (i.e., in the plugin subclass contructor). This version of the SetParamInfo function (DefaultValue of type bool) should be called for plugin parameters of type boolean or event.

3.2.3.9 CFFPluginManager::SetParamInfo (char* version)

```
void SetParamInfo(  DWORD dwIndex,
                    const char* pchName,
                    DWORD dwType,
                    const char* pchDefaultValue);
```

Parameters

dwIndex	Index of the plugin parameter whose data are specified. It should be in the range [0, Number of plugin parameters).
pchName	A string containing the name of the plugin parameter. According to the FreeFrame specification it should be at most 16 1-byte ASCII characters long. Longer strings will be cut at the 16th character.
dwType	The type of the plugin parameter. Codes for allowed types are defined in FreeFrame.h.
pchDefaultValue	A string to be used as the default value of the plugin parameter.

Return value

None

Remarks

This method is called by a plugin subclass, derived from this class, to specify name, type, and default value of the plugin parameter whose index is passed as parameter to the method. This method is usually called when a plugin object is instantiated (i.e., in the plugin subclass contructor). This version of the SetParamInfo function (DefaultValue of type char*) should be called for plugin parameters of type text.

3.2.3.10 CFFPluginManager::GetFrameWidth

```
int GetFrameWidth() const;
```

Parameters

None

Return value

The width of the images the plugin will have to process

Remarks

This method is called by a plugin subclass, derived from this class, to know the width of the images that it will have to process. This method is usually called before starting processing frames.

3.2.3.11 CFFPluginManager::GetFrameHeight

```
int GetFrameHeight() const;
```

Parameters

None

Return value

The height of the images the plugin will have to process.

Remarks

This method is called by a plugin subclass, derived from this class, to know the height of the images that it will have to process. This method is usually called before starting processing frames.

3.2.3.12 CFFPluginManager::GetFrameDepth

```
DWORD GetFrameDepth() const;
```

Parameters

None

Return value

The depth of the images the plugin will have to process, where 0 is 16 bits, 1 is 24 bits, 2 is 32 bits (see FreeFrame specification).

Remarks

This method is called by a plugin subclass, derived from this class, to know the depth of the images that it will have to process. This method is usually called before starting processing frames.

3.2.3.13 CFFPluginManager::GetFrameOrientation

```
DWORD GetFrameOrientation() const;
```

Parameters

None

Return value

The orientation of the images the plugin will have to process, where 1 is top left and 2 is bottom left (see FreeFrame specification).

Remarks

This method is called by a plugin subclass, derived from this class, to know the orientation of the images that it will have to process. This method is usually called before starting processing frames.

3.3 CFreeFramePlugin

CFreeFramePlugin is the base class for all FreeFrame plugins developed with the FreeFrame SDK.

```
#include <FFPluginSDK.h>
```

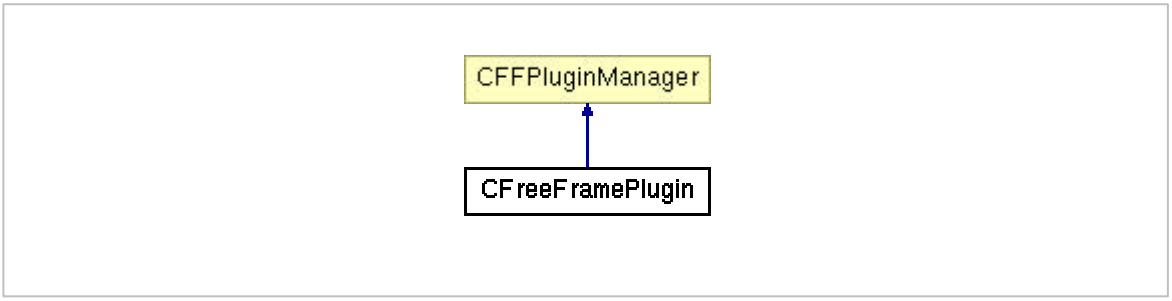


Figure 3.2: Inheritance diagram for CFreeFramePlugin

```
class CFreeFramePlugin : public CFFPluginManager {

public:

        virtual ~CFreeFramePlugin();

        virtual char* GetParameterDisplay(DWORD dwIndex);

        virtual DWORD SetParameter(const SetParameterStruct* pParam);

        virtual DWORD GetParameter(DWORD dwIndex);

        virtual DWORD ProcessFrame(void* pFrame);

        virtual DWORD ProcessFrameCopy(ProcessFrameCopyStruct* pFrameData);

        virtual DWORD GetInputStatus(DWORD dwIndex);

        CFreeFramePlugin* m_pPlugin;

protected:

        CFreeFramePlugin();
```

```
};
```

The CFreeFramePlugin class is the base class for every FreeFrame plugins developed with the FreeFrame SDK. It is derived from CFFPluginManager, so that most of the plugin management and communication with the host can be transparently handled through the default implementations of the methods of CFFPluginManager.
While CFFPluginManager is used by the global FreeFrame methods, CFreeFramePlugin provides a default implementation of the instance specific FreeFrame functions. Note that CFreeFramePlugin methods are virtual methods: any given FreeFrame plugin developed with the FreeFrame SDK will be a derived class of CFreeFramePlugin and will have to provide a custom implementation of most of such methods. Except for CFreeFramePlugin::GetParameterDisplay and CFreeFramePlugin::GetInputStatus, all the default methods of CFreeFramePlugin just return FF_FAIL: every derived plugin is responsible of providing its specific implementation of such default methods.

3.3.1 Public Member Functions

3.3.1.1 CFreeFramePlugin:: ~CFreeFramePlugin

```
virtual ~CFreeFramePlugin();
```

The standard destructor of CFreeFramePlugin.

3.3.1.2 CFreeFramePlugin::GetParameterDisplay

```
virtual char* GetParameterDisplay(DWORD dwIndex);
```

Parameters

dwIndex The index of the parameter whose display value is queried. It should be in the range [0, Number of plugin parameters).

Return value

The display value of the plugin parameter or NULL in case of error.

Remarks

Default implementation of the FreeFrame getParameterDisplay instance specific function. It provides a string to display as the value of the plugin parameter whose index is passed as parameter to the method. This default implementation just returns the string representation of the float value of the plugin parameter. A custom implementation may be provided by every specific plugin.

3.3.1.3 CFreeFramePlugin::SetParameter

```
virtual DWORD SetParameter(const SetParameterStruct* pParam);
```

Parameters

pParam A pointer to a SetParameterStruct (see FreeFrame.h and FreeFrame specification for further information) containing the index and the new value of the plugin parameter whose value is going to be set. The parameter index should be in the range [0, Number of plugin parameters).

Return value

The default implementation always returns FF_FAIL. A custom implementation must be provided.

Remarks

Default implementation of the FreeFrame setParameter instance specific function. It allows setting the current value of the plugin parameter whose index is passed as parameter to the method. This default implementation always returns FF_FAIL. A custom implementation must be provided by every specific plugin.

3.3.1.4 CFreeFramePlugin::GetParameter

```
virtual DWORD GetParameter(DWORD dwIndex);
```

Parameters

dwIndex The index of the parameter whose current value is queried. It should be in the range [0, Number of plugin parameters).

Return value

The default implementation always returns FF_FAIL. A custom implementation must be provided by every specific plugin.

Remarks

Default implementation of the FreeFrame getParameter instance specific function. It allows getting the current value of the plugin parameter whose index is passed as parameter to the method. This default implementation always returns FF_FAIL. A custom implementation must be provided by every specific plugin.

3.3.1.5 CFreeFramePlugin::ProcessFrame

```
virtual DWORD ProcessFrame(void* pFrame);
```

Parameters

pFrame Pointer to the frame that has to be processed by the plugin.

Return value

The default implementation always returns FF_FAIL. A custom implementation must be provided by every specific plugin.

Remarks

Default implementation of the FreeFrame processFrame instance specific function. This function performs in place processing of the frame passed as parameter. This default implementation always returns FF_FAIL. A custom implementation must be provided by every specific plugin.

3.3.1.6 CFreeFramePlugin::ProcessFrameCopy

```
virtual DWORD ProcessFrameCopy(ProcessFrameCopyStruct* pFrameData);
```

Parameters

pFrameData Pointer to a ProcessFrameCopyStruct structure (see the definition in FreeFrame.h and the description in the FreeFrame specification).

Return value

The default implementation always returns FF_FAIL. A custom implementation may be provided.

Remarks

Default implementation of the FreeFrame processFrameCopy instance specific function. This function processes the input frame(s) by possibly performing copy operations. This default implementation always returns FF_FAIL. A custom implementation must be provided by every specific plugin supporting processFrameCopy processing mode. In case only in place processing is supported the empty default implementation will suit.

3.3.1.7 CFreeFramePlugin::GetInputStatus

```
virtual DWORD GetInputStatus(DWORD dwIndex);
```

Parameters

dwIndex The index of the input whose status is queried. It should be in the range [Minimum number of inputs, Maximum number of inputs).

Return value

The default implementation always returns FF_FF_INPUT_INUSE or FF_FAIL if the index is out of range. A custom implementation may be provided by every specific plugin.

Remarks

Default implementation of the FreeFrame getInputStatus instance specific function. This function is called to know whether a given input is currently in use. For the default implementation every input is always in use. A custom implementation may be provided by every specific plugin.

3.3.2 Public Data Fields

```
CFreeFramePlugin* m_pPlugin;
```

The only public data field CFreeFramePlugin contains is m_pPlugin, a pointer to the plugin instance. Subclasses may use this pointer for self-referencing (e.g., a plugin may pass this pointer to external modules, so that they can use it for calling the plugin methods).

3.3.3 Protected Member Functions

```
CFreeFramePlugin();
```

The only protected function of CFreeFramePlugin is its constructor. In fact, nor CFFPluginManager objects nor CFreeFramePlugin objects should be created directly, but only objects of the subclasses implementing specific plugins should be instantiated. Moreover, subclasses should define and provide a factory method to be used by the FreeFrame SDK for instantiating plugin objects. An example of factory method is shown in the following:

```
DWORD CMyPlugin::CreateInstance(void** ppInstance)
{
        *ppInstance = new CMyPlugin();
        if (*ppInstance != NULL) return FF_SUCCESS;
        return FF_FAIL;
}
```

For further information on how to make FreeFrame plugins using the classes of the FreeFrame SDK, see the tutorial later in this manual.

4 Developing a sample FreeFrame plugin

This Chapter should be considered as a tutorial showing how to develop a new FreeFrame plugin using the FreeFrame SDK and the FreeFrame Wizard. Since the use of the FreeFrame Wizard the example will be discussed with reference to Microsoft Visual Studio 6. It however compiles on Microsoft Visual Studio .NET too. Both the source and the compiled code (i.e., the DLL) of this example are fully available in the FreeFrame SDK distribution in the Tutorial folder.
In this example we are going to create a simple effect plugin that only adds a constant value passed as parameter to the RGB values of all the pixels in the input image.

4.1 Step 1: Using the FreeFrame Wizard

The first step in developing a new FreeFrame plugin with the FreeFrame SDK and the FreeFrame Wizard consists in using the Wizard for creating a project for an empty plugin. The Wizard provides a template that you have to fill with your own code in order to make the plugin.
Let's thus start by opening Microsoft Visual Studio 6, selecting *New* in the *File* menu, and then the *Projects* page. If the Wizard has been installed properly (see Chapter 1 for installation instructions) it should appear in the list as "FreeFramePlugin AppWizard". Select it, type the project name (e.g., FFSDKTutorial), select the directory in which the Wizard will create the new project, and press *OK*. You will be introduced to Step 1 of the FreeFrame Wizard that asks for the name of the new plugin and its type. As for the name FFSDKTutarial will suit, for the type select "Effect". Click on the *Next* button to move to Step 2.
At Step 2 we have to provide same basic information about the new plugin. In the *Description string* box we can write a short sentence describing what the new plugin will do. In our case something like "A simple plugin showing how to make a FreeFrame plugin using the FreeFrame SDK" can be enough. In the *About string* box you can put your name and possible affiliation. Our plugin is very simple and it can work in place, so we are not going to support Process Frame Copy working mode: therefore, we leave the *Process Frame Copy supported* box unchecked. We behave similarly as for supported image formats: in order to keep things simple we support only RGB24 images and therefore we check only the *RGB 24* checkbox in the *Supported formats* group. Our sample plugin will not have any particular optimization, so we select "None" in the *Supported optimization* combo box. Now we are ready to click on *Next* and perform the last step of the FreeFrame Wizard.
Step 3 asks us to provide information about possible parameters of the new plugin. Our sample plugin will have just one standard parameter: the value to be added to the RGB values of each pixel of the incoming image. In order to add it to the list of plugin's parameters, click on the *Add* button. A dialog box will appear asking for the name of the new parameter, its type, and its default value. "Value" will suit as name. As for the type select "Standard" (i.e., a floating point value in [0, 1]). The default value can be 0.5. We do not need a custom implementation of the getParameterDisplay function, so we can leave the corresponding checkbox unchecked and click on the *Finish* button. The FreeFrame Wizard will inform us that it is going to create a new project and it will actually do the work after a click on the *OK* button.

4.2 Step 2: Looking at the code produced by the Wizard

At this point, the FreeFrame Wizard will have produced a new Microsoft Visual Studio 6 workspace and project for the new plugin. Looking at the folder where the project has been created you will find the files listed at the end of Chapter 2. Let's have a look at the two most important ones: the .h and .cpp files of the new plugin. If you chose FFSDKTutorial as name for the new project, their names will be FFSDKTutorial.h and FFSDKTutorial.cpp respectively. You can find them in the *File View* window of Microsoft Visual Studio 6. Note, however, that now you may also import the project in Microsoft Visual Studio .NET and proceed with it.
FFSDKTutorial.h should look like the following:

```
//
// FFSDKTutorial.h
//
// Plugin header file
//

#ifndef FFSDKTUTORIAL_STANDARD
#define FFSDKTUTORIAL_STANDARD
```

```
#include "FFPluginSDK.h"

class CFFSDKTutorial : public CFreeFramePlugin {

public:

        ~CFFSDKTutorial();

        //////////////////////////////////////////////////
        // FreeFrame plugin methods
       //////////////////////////////////////////////////

        DWORD   SetParameter(const SetParameterStruct* pParam);
        DWORD   GetParameter(DWORD dwIndex);
        DWORD   ProcessFrame(void* pFrame);

        //////////////////////////////////////////////////
        // Factory method
        //////////////////////////////////////////////////

        static DWORD __stdcall CreateInstance(void** ppInstance);

private:

        CFFSDKTutorial();

        // Parameters
        float m_Value;
};

#endif
```

The file declares the CFFSDKTutorial class as public subclass of CFreeFramePlugin. This class is the main plugin class and it will implement our sample plugin. Beside the standard destructor, four other public methods are declared. SetParameter and GetParameter will manage the parameters of the plugin (in our case the only "Value" parameter providing the value to be summed to all pixels), while ProcessFrame will actually do the main job of our sample plugin. For all the other generic and instance specific functions listed in the FreeFrame specification, the FreeFrame SDK provides a default implementation and we usually do not need to care about them. Less simple plugins may have further methods in their class. For example, if a plugin also supports Process Frame Copy working mode it will include a ProcessFrameCopy method; if a custom implementation of getParameterDisplay is needed, the plugin class will have a GetParameterDisplay method.
The fourth public method, CreateIstance, is a factory method used by the FreeFrame SDK to create instances of the plugin class. The FreeFrame Wizard automatically provides its implementation and we will find it in the FFSDKTutorial.cpp file. Notice that this method is the only available way for instantiating a plugin object: in fact, the standard constructor of the plugin class is declared as private. This mechanism allows the FreeFrame SDK to instantiate a plugin object without knowing the name of the plugin class. The FreeFrame SDK will only need a pointer to this method. Then, it will be able to instantiate a plugin object and to invoke its methods through the interface provided by the base class CFreeFramePlugin.

Finally in the private section of the plugin class we will find some variables responsible of storing the current values of the plugin parameters. In our case we just have one of them, m_Value, that will store the standard float value assigned by the host to the “Value” parameter.
Let’s now move to the FFSDKTutorial.cpp file. It should be similar to the following one:

```
//
// FFSDKTutorial.cpp
//
// Plugin implementation file
//

#include "FFSDKTutorial.h"

#define FFPARAM_VALUE   0

////////////////////////////////////////////////////////////////////////////////////////////////
//  Plugin information
////////////////////////////////////////////////////////////////////////////////////////////////

static CFFPluginInfo PluginInfo (
        CFFSDKTutorial::CreateInstance,                 // Create method
        "ZEBs",                                         // Plugin unique ID
        "FFSDKTutorial",                                // Plugin name
        1,                                              // API major version number
        000,                                            // API minor version number
        1,                                              // Plugin major version number
        000,                                            // Plugin minor version number
        FF_EFFECT,                                      // Plugin type
        "A simple plugin showing how to make a FreeFrame plugin using the FreeFrame SDK",
        "InfoMus Lab - DIST - University of Genova"     // About
);

////////////////////////////////////////////////////////////////////////////////////////////////
//  Constructor and destructor
////////////////////////////////////////////////////////////////////////////////////////////////

CFFSDKTutorial::CFFSDKTutorial()
: CFreeFramePlugin()
{
        // Plugin properties
        SetProcessFrameCopySupported(false);
        SetSupportedFormats(FF_RGB_24);
        SetSupportedOptimizations(FF_OPT_NONE);

        // Input properties
        SetMinInputs(1);
```

```
	SetMaxInputs(1);

	// Parameters
	SetParamInfo(FFPARAM_VALUE, "Value", FF_TYPE_STANDARD, 0.500000f);
	m_Value = 0.500000f;
}

CFFSDKTutorial::~CFFSDKTutorial()
{
}

/////////////////////////////////////////////////////////////////////////////////////////////////
//  Methods
/////////////////////////////////////////////////////////////////////////////////////////////////

DWORD CFFSDKTutorial::ProcessFrame(void* pFrame)
{

	// TODO: Add here the code implementing your plugin

	return FF_SUCCESS;
}

DWORD CFFSDKTutorial::GetParameter(DWORD dwIndex)
{
	DWORD dwRet;

	switch (dwIndex) {

	case FFPARAM_VALUE:
		memcpy(&dwRet, &m_Value, 4);
		return dwRet;

	default:
		return FF_FAIL;
	}
}

DWORD CFFSDKTutorial::SetParameter(const SetParameterStruct* pParam)
{
	if (pParam != NULL) {

		switch (pParam->ParameterNumber) {

		case FFPARAM_VALUE:
```

```
                memcpy(&m_Value, &(pParam->NewParameterValue), 4);
                break;

        default:
                return FF_FAIL;
        }

        return FF_SUCCESS;

    }

    return FF_FAIL;
}

//////////////////////////////////////////////////////////////////////////////////////////////////
//  Factory method
//////////////////////////////////////////////////////////////////////////////////////////////////

DWORD CFFSDKTutorial::CreateInstance(void** ppInstance)
{
    *ppInstance = new CFFSDKTutorial();
    if (*ppInstance != NULL) return FF_SUCCESS;
    return FF_FAIL;
}
```

The file begins with some defines. These are the internal indexes of the plugin parameters used by the GetParameter and SetParameter functions for managing the plugin parameters. In our case there is just one of them since our sample plugin only has one parameter.
Then a static CFFPluginInfo object is instantiated. The FreeFrame SDK will use this object to get information about the plugin. Such information is provided through the parameters of the constructor. It includes a pointer to the factory method (it will be used by the FreeFrame SDK for instantiating the plugin object), the plugin unique ID as assigned by the FreeFrame Wizard, the name of the plugin, plugin and API version numbers, the type of the plugin, the description and about string we have provided at Step 2 of the FreeFrame Wizard.
After that the constructor and destructor section comes. The destructor is usually empty (and it will be empty in our sample plugin). The constructor instead provides the FreeFrame SDK with further information about the plugin by calling some methods of the CFFPluginManager class. In our case it will specify that our plugin does not support Process Frame Copy working mode (by calling SetProcessFrameCopySupported(false)), that we only support RGB24 images (by calling SetSupportedFormats(FF_RGB_24)), that we do not provide any particular optimization (by calling SetSupportedOptimizations(FF_OPT_NONE)). The constructor also provides information about the plugin inputs and parameters. As for inputs, our plugin will have just one (since it only supports “in place” processing): therefore the Wizard automatically generates the calls SetMinInputs(1) and SetMaxInputs(1) that inform the FreeFrame SDK about that. SetParamInfo(FFPARAM_VALUE, "Value", FF_TYPE_STANDARD, 0.500000f) is then called to say that our only parameter, named “Value”, is a standard parameter with default value 0.5. Finally, the m_Value variable is initialized with the default value of the parameter.
Let’s now have a look to the GetParameter and SetParameter methods. GetParameter returns to the FreeFrame SDK and to the host the current value of the plugin parameter whose index is specified as parameter of the method. In our case, it returns the value stored in the m_Value variable, when FFPARAM_VALUE (i.e., 0 according to the define at the beginning of the file) is passed as parameter to the method. SetParameter does the opposite job: it updates the value stored in the m_Value variable with the value provided by the host. Usually, we will not have to modify these functions, though a slight modification in SetParameter will be needed later for optimization issues.

The ProcessFrame method is the core of the implementation of our sample plugin. As the comment says it is here that we will have to add the code implementing the plugin. We will do this in the next step of this tutorial.
Finally, at the end of the FFSDKTutorial.cpp file we will find the default implementation of the factory method provided by the FreeFrame Wizard. Usually we will not need to modify it.
Therefore, after observing the files produced by the FreeFrame Wizard it seems that the ProcessFrame method is the only method needing some work from us. This is what we are going to do in the next step.

4.3 Step 3: Implementing the ProcessFrame method

The ProcessFrame method will usually contain the algorithm implemented by the new FreeFrame plugin. In our case, it will just scan the whole input image and add the value stored in the m_Value variable to the RGB values of all the pixels in the image.
However, before going on with the implementation of ProcessFrame we have to discuss an important issue. Since the RGB values of the image pixels are represented as 8 bit values, i.e., as integer numbers in the range [0, 255] we will have to convert the value of the parameter (in the range [0, 1] according to the FreeFrame specification) to the corresponding integer value in the range [0, 255]. Although this operation may be done within the ProcessFrame method, it can be observed that it is not needed to do it so often. In fact, the value of the parameter will not change every time the ProcessFrame method is called, but only when the host modifies it, i.e., when the SetParameter method is called. Thus, we can optimize our plugin by introducing a slight modification to the SetParameter function. We only introduce a new variable, say m_iValue, for the integer representation of the parameter in the range [0, 255], and in the SetParameter function we also update m_iValue (performing the needed conversion) while updating m_Value.
The new variable will be added to the private section of the CFFSDKTutorial subclass that will be modified in this way:

```
class CFFSDKTutorial : public CFreeFramePlugin {
...
private:

    CFFSDKTutorial();

    // Parameters
    float m_Value;
    int m_iValue;
};
```

The SetParameter method will be modified accordingly;

```
DWORD CFFSDKTutorial::SetParameter(const SetParameterStruct* pParam)
{
    if (pParam != NULL) {

        switch (pParam->ParameterNumber) {

        case FFPARAM_VALUE:
            memcpy(&m_Value, &(pParam->NewParameterValue), 4);
            m_iValue = (BYTE)(m_Value * 255.0f);   // conversion to integer in [0, 255]
            break;

        default:
            return FF_FAIL;
        }
```

```
            return FF_SUCCESS;

        }

        return FF_FAIL;
}
```

Let's now move back to the implementation of the ProcessFrame method. A possible way of doing it is as follows:

```
DWORD CFFSDKTutorial::ProcessFrame(void* pFrame)
{
        if (GetFrameDepth() != FF_DEPTH_24) return FF_FAIL;

        for (int k = 0; k < GetFrameWidth() * GetFrameHeight() * 3; ++k) {
            if ((255 - ((BYTE*)pFrame)[k]) < m_iValue) ((BYTE*)pFrame)[k] = 255;
            else ((BYTE*)pFrame)[k] += m_iValue;
        }

        return FF_SUCCESS;
}
```

First, we check for the image depth. If it is not 24 bits (i.e., it is not a RGB24 image) we just return FF_FAIL. Then, for each RGB value of every pixel in the input image we check whether summing the constant parameter value (stored in m_iValue) produces an overflow (i.e., the resulting value is greater that 255). In this case, we just assign 255 as value; otherwise we perform the sum.
Note that in order to know the properties of the input image, namely its width, height, and depth, we only need to call the corresponding functions of the FreeFrame SDK. In particular, we call respectively GetFrameWidth, GetFrameHeight, and GetFrameDepth that are protected methods of the CFFPluginManager class. These methods of the FreeFrameSDK are implemented as inline functions and therefore they should not introduce any further computational load.

4.4 Step 4: Building the plugin DLL

The last step consists in building the DLL of our new plugin. In Microsoft Visual Studio (both 6 and .NET versions) you can do it by clicking on the *Build* button of the build toolbar. If everything works as it should, you should not get any error or warning. Remember that you have to configure Microsoft Visual Studio as described in Chapter 1 for using the FreeFrame SDK with it. The FreeFrame Wizard should have already configured your project for linking with the library files of the FreeFrame SDK. If this is not the case you have to add FreeFrameSDK.lib in the *Object/libraries modules* list in the *Link* property page inside the *Settings* dialog box (you can open it from the *Project* menu). In Microsoft Visual Studio .NET the same operation can be performed opening the *Properties* dialog from the *Project* menu, selecting *Linker* and then *Input*, and typing FreeFrameSDK.lib in the *Additional Dependencies*. Note that FreeFrameSDK.lib is the release version of the FreeFrame SDK library file. If you are in the Debug configuration you should rather use FreeFrameSDKd.lib.
Once the plugin DLL has been created you can test it either with the Plugin Tester provided with the FreeFrame distribution or with whatever FreeFrame host you like.

A final note: what about if you cannot or don't want to use the FreeFrame Wizard? You can still make an empty project for a DLL and provide the needed .h and .cpp files by writing them by hand. You only need four files: the .h and .cpp files implementing the plugin class as discussed above (you can use the ones provided with this tutorial as template), a .def file for exporting the plugMain function (see the FreeFrame specification and the .def file included in the source code of this tutorial), and the FFPluginInfoData.cpp file. You can find the latter in the source code of this tutorial too: you can directly use that one since usually you will not need to modify it.

Appendix A: The FreeFrame Specification

Here follows the FreeFrame Specification 1.0 – 03.
Note that FreeFrame is Copyright (c) 2002 – 2005, www.freeframe.org, all rights reserved, and that the FreeFrame Specification document is subject to the same license reported at the beginning of this manual. The Freeframe Standard was originally conceived and developed by Russell Blakeborough and Marcus Clements at Brightonart Ltd. UK. http://www.brightonart.org. Please visit www.freeframe.org for updated versions of the FreeFrame Specification.

A.1 OS integration

FreeFrame plugins are distributed and used as compiled shared objects (.so) in Linux and Mac OS X, as Dynamically Linked Libraries (.dll) in Windows.

FreeFrame Plugins export a single function: plugMain. This is passed 3 values: a 32-bit function code, a 32-bit input value, and a 32-bit instance identifier. It returns a 32-bit output value.
The input values and output values have different types according to the function code.
This may be implemented differently on different platforms in different languages, but the format of the values must be consistent.

```
plugMain ( unsigned integer functionCode,
           unsigned integer inputValue,
           unsigned integer instanceID )
```

Parameters

functionCode: Tells the plugin which function is being called (see function table).
inputValue: 32-bit value or 32-bit pointer to some structure, depending on function code.
instanceID: 32-bit instance identifier. Only used for instance specific functions (see function table).

Return value

Depends on function code. See function table for details.

Remarks

PLUGIN DEVELOPERS: You shouldn't need to change this function. This is the one and only function exposed by the library containing the actual plugin.

A.2 Basic types and constants

All numbers in this document are ordinary decimal numbers, except if specified otherwise.

The following types are used:

Type	Range of type	Notes
32-bit unsigned integer	0 to 4294967295	-
32-bit pointer	0 to 4294967295	Needs to be a valid memory location
32-bit IEEE float	-lots to +lots	1 sign bit, 8 bit exponent, 24 bit mantissa

The following constants are used as return values:

FF_SUCCESS	0
FF_FAIL	(hex) FFFFFFFF
FF_TRUE	1
FF_FALSE	0
FF_SUPPORTED	1
FF_UNSUPPORTED	0

Remarks
- In functions that return error codes any value other than 0 is assumed to be an error code. These are defined in each function below.
- In functions that return a pointer, (hex) FFFFFFFF also represents a failure condition.

A.3 Enumerated and derived types

InstanceIdentifier (32-bit unsigned integer)

Definition

InstanceIdentifier: Unique identifier for the instance of a plugin. For some plugins, this value might represent a pointer to a plugin-specific data structure containing the instance's current state. Or it could be an index to a table of all active instances of a plugin, or any other value that uniquely identifies an instance.

NumParameters (32-bit unsigned integer)

Definition

NumParameters: Specifies the number of parameters that the plugin implements.

ParameterNumber (32-bit unsigned integer)

Definition

ParameterNumber: Index of a parameter, starting at 0 for the first parameter.

ParameterName (array of 16 1-byte ASCII characters, *not null terminated*)

Definition

ParameterName: The name of the Parameter as it will be displayed by the host on the UI.

ParameterValue (32-bit float value OR 32-bit pointer)

Definition

ParameterValue: Float value from 0-1 or pointer to null terminated string (see remarks).

Remarks

Apart from text parameters, FreeFrame parameter values are always 32-bit floats, and the range of values permitted is STRICTLY 0-1 (0 <= ParameterValue <= 1). This allows faster processing and a good range of values over a standard range, so the host can run up sliders or similar for the plugin. The use of any values outside this range will result in hideous incompatibilities. The ParameterDisplayValue can be used to display whatever actual values the plugin likes e.g. 0-255, 0-767, 1-256 or whatever. The plugin should translate the standard 0-1 float range into the values it needs for its processing. For text parameters, this 32-bit value represents a pointer to a null terminated string.

ParameterDefaultValue (32-bit float value OR 32-bit pointer)

Definition

ParameterDefaultValue: The initial default value for this parameter. 32-bit float value or 32-bit pointer to null terminated string (see remarks).

Remarks

Plugins should always specify default values. Sometimes a host may not implement all parameters on a plugin, so the plugin must use default values until told to do otherwise by the host. Apart from text parameters, FreeFrame parameter values are always 32-bit floats, and the range of values permitted is STRICTLY 0-1.
For text parameters, this 32-bit value represents a pointer to a null terminated string.

ParameterDisplayValue (array of 16 1-byte ASCII characters, *not null terminated*)

Definition

ParameterDisplayValue: String representing the current display value of this parameter. The plugin can display whatever it likes here e.g. just the float, a rounded 0-100 '%' representation for the user, words representing states like 'on' / 'off', different effects that the one plugin can do etc…

PluginCapsIndex (32-bit unsigned integer)

Definition

PluginCapsIndex: 32-bit unsigned integer. Specifies certain capabilities of a plugin that the host may want to enquire about:
0 = 16bit 5-6-5 support
1 = 24bit packed support
2 = 32bit (or 24bit 32-bit aligned) support

3 = Plugin supports processFrameCopy function

Calls to these caps indexes are only meaningful if the plugin has reported that it supports processFrameCopy:
10 = Minimum input frames
11 = Maximum input frames
15 = Plugin optimized for copy or in-place processing

Remarks

See getPluginCaps for more information about the usage of this structure.

ParameterType (32-bit unsigned integer)

Definition

ParameterType: 32-bit unsigned integer. Tells the host what kind of data the parameter is.
Current meaningful values:

Value	Type	Description
0	boolean	0.0 defined as false and anything else defined as true - e.g. checkbox
1	event	Similar to boolean but for a momentary push button style trigger. 1.0 is set

		momentarily to denote a simple event - e.g., pushbutton / keystroke.
2	red	The 3 colors e.g. for a colorpicker
3	green	
4	blue	
5	xpos	For x, y video interaction e.g. cursor - these should denote position within the video frames as specified in the VideoInfoStruct.
6	ypos	
10	standard	A standard parameter representing an unspecified float value
100	text	A null terminated text input type - Note: only this type has a different data type for the moment

InputChannel (32-bit unsigned integer)

Definition

InputChannel: Input channel as relates to the use of multiple inputs in processFrameCopy.
The first channel is 0.

InputStatus (32-bit unsigned integer)

Definition

InputStatus: Status of input channels as relates to the use of multiple inputs in processFrameCopy.
0 = Not in use
1 = In Use

A.4 Structures

PluginInfoStruct (size = 32 bytes)

```
PluginInfoStruct {
  APIMajorVersion: 32-bit unsigned integer
  APIMinorVersion: 32-bit unsigned integer
  PluginUniqueID: 4 1-byte ASCII characters *not null terminated*
  PluginName: 16 1-byte ASCII characters *not null terminated*
  PluginType: 32-bit unsigned integer
}
```

Fields

APIMajorVersion: Represents number before decimal point in version numbers
APIMinorVersion: Represents number after decimal point in version numbers
PluginUniqueID: Unique ID for plugin
PluginName: Name of plugin
PluginType: Current meaningful values (see remarks):
0 = effect
1 = source

Remarks

Plugins of PluginType effect are passed frames of video, which they then modify. Source plugins are simply passed a pointer where they paint frames of video. One example of a source plugin would by a visual synthesizer which uses the parameters to synthesize video.
FreeFrame APIMinorVersion numbers should always have 3 digits - so divide the minor version number by 1000 and add the major version number to get the full version number. Example: for version 0.511, APIMajorVersion=0 and APIMinorVersion=511.

VideoInfoStruct (size = 16 bytes)

```
VideoInfoStruct {
  FrameWidth: 32-bit unsigned integer
  FrameHeight: 32-bit unsigned integer
  BitDepth: 32-bit unsigned integer
  Orientation: 32-bit unsigned integer
}
```

Fields

FrameWidth: Width of video frame in pixels.
FrameHeight: Height of video frame in pixels.
BitDepth: Current meaningful values (see remarks):
0 = 16bit, 5-6-5 packed
1 = 24bit, packed (byte order: BGR)
2 = 32bit, also suitable for 32-bit unsigned integer aligned 24bit (byte order: BGRA)
Orientation: Current meaningful values (see remarks):
1 = origin at top left
2 = origin at bottom left

Remarks

Plugins using 32bit as byte aligned 24 bit video should be careful not to overwrite the alpha (4th) byte of each pixel (e.g. by using it as a processing space) as this may be used soon by hosts with 32bit video processing becoming more accessible.
If Orientation == 1 the first pixel at the pointer to the frame is the top left pixel. If Orientation == 2, it is the bottom left pixel. This is particularly important for text and live input (to avoid mirroring of the image).

SetParameterStruct (size = 8 bytes)

```
SetParameterStruct {
  ParameterNumber: ParameterNumber
  NewParameterValue: ParameterValue
}
```

Fields

ParameterNumber: Index of the parameter to set.
NewParameterValue: New value for this parameter. For float parameters, this is a float from 0-1. For text parameters, this is a pointer to a null terminated string (see ParameterValue).

PluginExtendedInfoStruct (size = 24 bytes)

```
PluginExtendedInfoStruct{
  PluginMajorVersion: 32-bit unsigned integer
  PluginMinorVersion: 32-bit unsigned integer
  Description: 32-bit pointer to null terminated string
  About: 32-bit pointer to null terminated string
  FreeFrameExtendedDataSize: 32-bit unsigned integer
  FreeFrameExtendedDataBlock: 32-bit pointer
}
```

Fields

PluginMajorVersion: Represents number before decimal point in version numbers.
PluginMinorVersion: Represents number after decimal point in version numbers.
Description: A description of the plugin - intended to be made available to the user in hosts supporting this function via the UI. Pointer value can be 0 if not provided by plugin!
About: Author and license information. Pointer value can be 0 if not provided by plugin!

FreeFrameExtendedDataSize: Size in bytes of the FreeFrameExtendedDataBlock - or 0 if not provided by plugin.
FreeFrameExtendedDataBlock: 32-bit pointer to a FreeFrame ExtendedDataBlock - this interface is not yet in use, but provided to allow the FreeFrame project to establish an extended data block format in the future. Please do not use until a data format has been agreed.

Remarks

FreeFrame PluginMinorVersion numbers should always have 3 digits - so divide the minor version number by 1000 and add the major version number to get the full version number. Example: 0.751 PluginMajorVersion=0, PluginMinorVersion=751

ProcessFrameCopyStruct (size = 12 bytes)

```
ProcessFrameCopyStruct{
  numInputFrames: 32-bit unsigned integer
  ppInputFrames: 32-bit pointer to array of pointers
  pOutputFrame: 32-bit pointer
}
```

Fields

numInputFrames: Number of input frames contained in ppInputFrames.
ppInputFrames: A pointer to an array of pointers to input frames.
pOutputFrame: Pointer to output frame.

A.5 Functions

Some functions are specific to an instance of a plugin, and thus require a valid InstanceID in the InstanceID field; others concern global parameters that are the same for all instances, and so do not require a valid InstanceID.

Function Code Table

Global functions

Code	Function	Input value type	Output value type
0	GetInfo	None	Pointer to a PluginInfoStruct
1	Initialize	None	Success/error code
2	DeInitialise	None	Success/error code
4	GetNumParameters	None	NumParameters
5	GetParameterName	ParameterNumber	Pointer to ParameterName
6	GetParameterDefault	ParameterNumber	ParameterDefaultValue
10	GetPluginCaps	PluginCapsIndex	Supported/unsupported/value
13	GetExtendedInfo	None	Pointer to PluginExtendedInfoStruct
15	GetParameterType	ParameterNumber	ParameterType

Instance specific functions

Code	Function	Input value type	Output value type
3	ProcessFrame	Pointer to a frame of video	Success/error code
7	GetParameterDisplay	ParameterNumber	Pointer to ParameterDisplayValue
8	SetParameter	Pointer to SetParameterStruct	Success/error code
9	GetParameter	ParameterNumber	ParameterValue
11	Instantiate	Pointer to VideoInfoStruct	InstanceIdentifier
12	DeInstantiate	None	Success/error code
14	ProcessFrameCopy	Pointer to ProcessFrameCopyStruct	Success/error code
16	GetInputStatus	InputChannel	InputStatus

A.5.1 Global functions

getInfo (function code = 0)

Input value

None

Return value

32-bit pointer to PluginInfoStruct if successful, FF_FAIL otherwise.

Remarks

Gets information about the plugin - version, unique id, short name and type (see PluginInfoStruct).
This function should be identical in all future versions of the FreeFrame API.

HOST: Call this function first to get version information. The version defines the other function codes that are supported.

initialise (function code = 1)

Input value

None

Return value

FF_SUCCESS if successful, FF_FAIL or other error-codes if failed (32-bit unsigned integer).

Remarks

PLUGIN: Prepare the plug-in for processing. Set default values, allocate memory. When the plug-in returns from this function it must be ready to accept calls to instantiate.
HOST: This function *must* return before a call to instantiate.

deinitialise (function code = 2)

Input value

None

Return value

FF_SUCCESS if successful, FF_FAIL or other error-codes if failed (32-bit unsigned integer).

Remarks

PLUGIN: Tidy up, deallocate memory.
HOST: This *must* be the last function called on the plugin.

getNumParameters (function code = 4)

Input value

None

Return value

NumParameters indicating the number of parameters in plugin or FF_FAIL on error.

Remarks

PLUGIN: Plugin developers should normally expect hosts to expose up to 8 parameters. Some hosts may only expose 1 or 4 parameters, some may expose larger numbers. All parameters should have sensible defaults in case the user is unable to control them from the host.
HOST: Host developers should try to implement at least the first 4 parameters. 8 is the recommended number to expose.

getParameterName (function code = 5)

Input value

index: ParameterNumber

Return value

32-bit pointer to ParameterName (16 byte char array) containing the name of parameter specified by index. Returns FF_FAIL on error.

getParameterDefault (function code = 6)

Input value

index: ParameterNumber

Return value

ParameterDefaultValue indicating the default value of parameter specified by index. For float parameters, the return value is a 32-bit float (0<=value<=1). Return value for text parameters is a 32-bit pointer to a null terminated string. Returns FF_FAIL on error.

getPluginCaps (function code = 10)

Input value

capsIndex: PluginCapsIndex

Return value

Indicates capability of feature specified by capsIndex. See PluginCapsIndex for a more detailed description of the individual capabilities. In every case, the return value can be represented as a 32-bit unsigned integer.

capsIndex value	**capsIndex meaning**	**Return values**
0	16 bit video	FF_SUPPORTED: plugin supports 16 bit video FF_UNSUPPORTED: plugin doesn't support 16 bit video
1	24 bit video	FF_SUPPORTED: plugin supports 24 bit video FF_UNSUPPORTED: plugin doesn't support 24 bit video
2	32 bit video	FF_SUPPORTED: plugin supports 32 bit video FF_UNSUPPORTED: plugin doesn't support 32 bit video

3	processFrameCopy support	FF_SUPPORTED: plugin supports processFrameCopy FF_UNSUPPORTED: plugin doesn't support processFrameCopy
10	Minimum input frames	Returns minimum number of input frames plugin requires or FF_FALSE if capability not supported.
11	Maximum input frames	Returns maximum number of input frames plugin can process or FF_FALSE if capability not supported.
15	Plugin optimization	Return value indicates whether plugin is optimized for Copy or InPlace processing. 0 = no preference 1 = InPlace processing is faster 2 = Copy processing is faster 3 = Both are optimized

Remarks

Bitdepth format of the video: the host asks the plugin if it is capable of its favorite bit depth, and uses that if it is available. If not the host may decide not to use the plugin and deinitialise it, or it may enquire again to see if a second choice format is supported.

getExtendedInfo (function code = 13)

Input value

None

Return value

32-bit pointer to PluginExtendedInfoStruct or FF_FAIL on error.

getParameterType (function code = 15)

Input value

index: ParameterNumber

Return value

ParameterType value which tells the host what kind of data the parameter index is. Returns FF_FAIL on error.

Remarks

Hosts may decide to present an appropriate visual interface depending on the parameter type. All FreeFrame data for the moment is passed as 32bit floats from 0-1, except text data.

A.5.2 Instance specific functions

processFrame (function code = 3, needs valid instanceID!)

Input value

pFrame: 32-bit pointer to byte array containing frame of video.

Return value

FF_SUCCESS if successful, FF_FAIL if failed. (32-bit unsigned integer)

Remarks

PLUGIN: Process a frame of video 'in place'. This is also the basic frameport for source plugins.
HOST: pFrame needs to be a valid pointer throughout this call as the plugin processes the frame 'in place'.

getParameterDisplay (function code = 7, needs valid instanceID!)

Input value

index: ParameterNumber

Return value

Pointer to ParameterDisplayValue containing a string to display as the value of parameter index. Returns FF_FAIL if failed.

setParameter (function code = 8, needs valid instanceID!)

Input value

setParam: Pointer to SetParameterStruct.

Return value

Returns FF_SUCCESS if successful or FF_FAIL if failed.

Remarks

Sets the value of a parameter.

getParameter (function code = 9, needs valid instanceID!)

Input value

index: ParameterNumber

Return value

A ParameterValue or FF_FAIL on error.

Remarks

Returns value of a parameter. Note that the return value has different interpretations, depending on the type of the parameter (see ParameterValue).

instantiate (function code = 11, needs valid instanceID!)

Input value

videoInfo: 32-bit pointer to VideoInfoStruct.

Return value

An InstanceIdentifier for the newly instantiated effect. Will be used as instanceID for future function calls. Returns FF_FAIL on error.

Remarks

PLUGIN: Prepare an instance of the Plug-in for processing. Set default values, allocate memory When the plug-in returns from this function it must be ready to process a frame. Make a copy of the VideoInfoStruct locally as pointer may not be valid after function returns. The instance identifier you provide to the host must be the unique identifier of this instance until a call to deinitialise.
HOST: This function *must* return before a call to processFrame. Pointer to VideoInfoStruct *must* be valid until function returns. The InstanceIdentifier provided by the plugin is your handle to this instance of the plugin. The host must keep this instance identifier, and provide it in all calls to the plugin regarding that instance.

deInstantiate (function code = 12, needs valid instanceID!)

Input value

None

Return value

Returns FF_SUCCESS if successful or FF_FAIL if failed.

Remarks

PLUGIN: Tidy up, deallocate memory All memory associated with the instance being deinited must be freed here.
HOST: This function must be called to close an instance of the plugin. All instances should be closed before deInitialising the plugin!

processFrameCopy (function code = 14, needs valid instanceID!)

Input value

copyStruct: 32-bit pointer to ProcessFrameCopyStruct

Return value

Returns FF_SUCCESS if successful or FF_FAIL if failed.

Remarks

The ProcessFrameCopy function performs a source->dest buffer frame process in effects plugins. It is capable of processing multiple input frames - getPluginCaps should be used to see how many input frames the plugin would like.

PLUGIN: Effect Plugins must support the processFrame (in-place) method of processing. This method is optional. Plugins should specify if they are capable of this kind of processing - and if they have optimized a particular method - in the Plugin Caps system. Source Plugins should not use this function as they do not require input frames. Source plugins should just use processFrame

getInputStatus (function code = 16, needs valid instanceID!)

Input value

channel: InputChannel value.

Return value

Returns an InputStatus value describing the status of channel.

Remarks

This function is provided to allow hosts to optimize the rendering of input frames. Due to user input a plugin may only plan to render certain input channels - this function allows hosts to ask the plugin which input channels it plans to render.

HOST: A host may only call this function for channels in this range: ‘Minimum input frames’ <= channel < ‘Maximum input framesì (see getPluginCaps). One should assume that all channels < ‘Minimum input frames’ are always in use.