Optimizing ADOBE® FLASH® LITE®
for the Digital Home
# Contents

## Chapter 1: Introducing Adobe Flash Lite for the digital home
- Getting started ......................................................... 1
- Developing platform-specific drivers ............................... 1
- Architecture of Flash Lite for the digital home ...................... 2
- Modules in Flash Lite for the digital home ........................ 2
- Running Flash Lite for the digital home ............................ 3
- Header files ...................................................................... 3
- API reference documentation .......................................... 4

## Chapter 2: The graphics driver
- Class overview ............................................................. 5
- User input handling ...................................................... 8
- Implementations included with source distribution ................. 8
- Implementation tasks overview ....................................... 9
- Plane class details ....................................................... 10
- Plane class methods ..................................................... 13
- PlaneFactory class details ........................................... 16
- PlaneFactory class methods ........................................... 17
- I2D class details .......................................................... 18
- I2D class methods ....................................................... 19
- IGraphicsDriver class details ........................................ 21
- Sample implementation walkthrough ................................ 23
- Implementation considerations ..................................... 27
- Creating files for your platform-specific graphics driver ........... 29
- Building your platform-specific graphics driver .................... 29
- Detailed tasks checklist ............................................... 29

## Chapter 3: The overlay video driver
- Class overview ............................................................. 32
- Audio and video codecs .................................................. 34
- Falling back to software decoding .................................... 34
- Implementations included with source distribution ................. 34
- StreamPlayer class details ............................................. 35
- StreamPlayer methods .................................................. 42
- IStreamPlayer class details ............................................ 46
- IStreamPlayer class methods .......................................... 47
- Creating files for your platform-specific graphics driver ........... 47
- Building your platform-specific overlay video driver ................ 47
- Buffer level tracking tools ............................................. 48

## Chapter 4: The sound driver
- Class and structure overview .......................................... 49
- Class interaction .......................................................... 49
Implementation included with source distribution .................................................. 52
Implementation tasks overview ............................................................................. 52
SoundOutput and SoundOutputBase class hierarchy ............................................. 52
SoundOutput methods .......................................................................................... 53
IFL31NativeSoundOutput class details ................................................................. 55
FL31NativeSoundOutput class methods ............................................................... 56
ALSA implementation ........................................................................................... 58
Creating files for your platform-specific sound driver ......................................... 59
Building your platform-specific sound driver ....................................................... 59

Chapter 5: The image decoder
Class overview ....................................................................................................... 60
Class interaction and logic flow ............................................................................ 60
Synchronous or asynchronous implementation .................................................... 63
Implementations included with source distribution .............................................. 63
Creating files for your platform-specific image decoder .................................... 63
Building your platform-specific image decoder .................................................... 64

Chapter 6: The audio decoder
Class overview ....................................................................................................... 65
Implementations included with source distribution .............................................. 66
AudioDecoder methods ........................................................................................ 66
IAudioDecoder class methods .............................................................................. 67
Creating files for your platform-specific audio decoder ..................................... 68
Building your platform-specific audio decoder ..................................................... 68

Chapter 7: The video decoder
Class overview ....................................................................................................... 69
Implementations included with source distribution .............................................. 70
VideoDecoder structures ...................................................................................... 70
VideoDecoder methods ....................................................................................... 72
IVideoDecoder class methods .............................................................................. 73
Creating files for your platform-specific video decoder .................................... 73
Building your platform-specific video decoder ..................................................... 74

Chapter 8: Coding, building, and testing
Common types and macros ................................................................................... 75
Kernel functionality ............................................................................................... 75
Operating system functionality ............................................................................ 77
The memory watchdog ......................................................................................... 78
Placing code in the directory structure ............................................................... 78
Building platform-specific drivers and decoders ................................................ 79
Measuring performance ....................................................................................... 84
Chapter 1: Introducing Adobe Flash Lite for the digital home

Adobe® Flash® Lite® for the digital home is Adobe® Flash® Player optimized for the hardware and software architecture of digital home electronics, such as television sets and Blu-ray players. Adobe® Flash® developers can create applications for Flash Lite for the digital home that stream and play high-definition video from the Internet. These developers can also create rich Internet applications and graphical user interfaces for Flash Lite for the digital home.

You, the developer for a digital home electronics platform, can optimize Flash Lite for the digital home to take advantage of your platform’s hardware capabilities.

Getting started

To get started, read Getting Started with Adobe Flash Lite for the Digital Home. The Getting Started document includes the following:

- An introduction to the Flash Lite for the digital home.
- How to install and build Flash Lite for the digital home on your Linux® platform.
- How to run Flash Lite for the digital home.

Developing platform-specific drivers

You can develop platform-specific drivers and decoders to optimize Flash Lite for the digital home. These drivers and decoders interact with components of Flash Lite for the digital home. Each of these drivers or decoders is an implementation of some C++ abstract classes included in the source distribution for Flash Lite for the digital home. The drivers and decoders you can develop are:

- **The graphics driver** Provides the interfaces to display Adobe Flash® Lite® animation on your platform’s display device. The graphics driver also provides hardware-acceleration of 2-D bitmap primitives. Details are in “The graphics driver” on page 5.

- **The overlay video driver** Directs dedicated hardware to decode and present an audio/video stream. Details are in “The overlay video driver” on page 31.

- **The sound driver** Directs your platform’s audio output hardware to play PCM samples that the Flash Lite applications generate. Details are in “The sound driver” on page 49.

- **The image decoder** Provides interfaces to dedicated hardware decoders to decode PNG and JPEG images to accelerate SWF movie playback. Details are in “The image decoder” on page 60.

- **The audio decoder** Directs your platform’s hardware to decode audio data into PCM samples. Details are in “The audio decoder” on page 65.

- **The video decoder** Directs your platform’s hardware to decode and render video data. The video decoder returns the decompressed frames to Flash Lite for the digital home to further process and display. Details are in “The video decoder” on page 69.
Architecture of Flash Lite for the digital home

A Flash Lite for the digital home application is in SWF file format. Flash Lite for the digital home can run multiple SWF movies at one time. The SWF content of each application appears in its own Stage. The Stage is a rectangular area on the display device. Flash Lite for the digital home uses a StageWindow instance to control each application’s Stage. Each StageWindow instance creates a Flash Lite instance. The Flash Lite instance contains a Flash Player that runs the SWF content. Flash Player supports Adobe® Flash® Lite® 3.1.

As the Flash Lite instance runs the SWF content, Flash Lite for the digital home interacts with your platform-specific drivers and decoders. The following diagram shows the relevant architecture of Flash Lite for the digital home:

![Flash Lite for the digital home architecture as it relates to platform-specific drivers and decoders](image)

*Note: Although this diagram shows three StageWindow instances, the actual number of StageWindow instances depends on how many SWF movies are running concurrently.*

Modules in Flash Lite for the digital home

Flash Lite for the digital home loads modules to perform tasks. For example, a StageWindow instance loads the IFlashLib module, and uses that module to create the Flash Lite instance. Similarly, when a StageWindow instance loads a SWF file, it prepares to display the SWF content by loading the GraphicsDriver module. All modules are subclasses of the IAEModule class. Many of the interfaces you implement to create platform-specific drivers and decoders also derive from the IAEModule class. Therefore, Flash Lite for the digital home loads your platform-specific modules as needed. For example, when the SWF content starts to play a video, the Flash Lite instance creates an instance of your platform-specific IStreamPlayer, which derives from IAEModule.
Running Flash Lite for the digital home

Flash Lite for the digital home provides an interface to load and run SWF content on the target platform. This interface is the IStagecraft interface. A C++ application running on your platform that uses the IStagecraft interface is called the host application. The host application is the client of the IStagecraft interface (just as any program that uses an interface is a client of the interface). A system developer for your platform develops the host applications.

The Flash Lite for the digital home source distribution provides a host application you can use for testing. This host application is in stagecraft_main.cpp in the directory source/executables/stagecraft.

*Note:* Your system developer can use stagecraft_main.cpp as your platform’s host application, or as a basis for your platform’s host application.

Header files

The source distribution contains all the files that make up Flash Lite for the digital home. The following table lists directories containing the header files you use to develop platform-specific drivers and decoders. In this example, the top directory stagecraft is the installation directory of Flash Lite for the digital home. More information about each of these header files is in the appropriate driver or decoder chapter.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stagecraft/include/ae/ddk/graphicsdriver</td>
<td>Header files for abstract interfaces of the graphics driver, including the PlaneFactory class.</td>
</tr>
<tr>
<td>stagecraft/include/ae/ddk/streamplayer</td>
<td>Header files for abstract interfaces of the overlay video driver.</td>
</tr>
<tr>
<td>stagecraft/include/ae/ddk/fl31nativesoundoutput</td>
<td>Header files for abstract interfaces of the sound driver.</td>
</tr>
<tr>
<td>stagecraft/include/ae/ddk/imagedecoder</td>
<td>Header files for abstract interfaces of the image decoder.</td>
</tr>
<tr>
<td>stagecraft/include/ae/ddk/audiodecoder</td>
<td>Header files for abstract interfaces of the audio decoder.</td>
</tr>
<tr>
<td>stagecraft/include/ae/ddk/videodecoder</td>
<td>Header files for abstract interfaces of the video decoder.</td>
</tr>
<tr>
<td>stagecraft/include/ae/os</td>
<td>Header files for abstract interfaces for accessing the platform’s operating system services. The source distribution provides Linux (and Win32) implementations. If your platform does not use Linux, a system developer for your platform implements these interfaces.</td>
</tr>
</tbody>
</table>
OPTIMIZING FLASH LITE FOR THE DIGITAL HOME
Introducing Adobe Flash Lite for the digital home

API reference documentation

To learn how to implement a platform-specific driver and decoder, see the appropriate chapter. However, for specific coding details about classes, methods, parameters, and return values, refer to the C++ API Doxygen reference. The C++ API Doxygen reference is provided with the source distribution.
Chapter 2: The graphics driver

Adobe® Flash® Lite® for the digital home renders and displays Adobe® Flash® Lite® animation on the display hardware of your target platform. To direct Flash Lite for the digital home to use your display hardware and APIs, you implement the graphics driver interfaces. The interfaces are abstract C++ classes. One of these classes, the Plane class, serves as the basis for implementing a bitmap image on the target platform. The image decoder and the overlay video driver also use the Plane class.

Class overview

The graphics driver interface includes four main classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plane</td>
<td>Abstract class you implement to provide platform-specific implementations of a bitmap.</td>
</tr>
<tr>
<td>PlaneFactory</td>
<td>Abstract class you implement to create and destroy Plane objects.</td>
</tr>
<tr>
<td>I2D</td>
<td>Abstract class you implement to provide hardware acceleration methods.</td>
</tr>
<tr>
<td>IGraphicsDriver</td>
<td>Abstract class implemented in Flash Lite for the digital home. You add methods to specify the names of your platform-specific Plane subclasses.</td>
</tr>
</tbody>
</table>

Plane class

Flash Lite for the digital home uses your implementations of the Plane class to render the frames of a SWF movie in memory and output them on a display device. Flash Lite for the digital home renders vector graphics as well as bitmap images.

The modules in this table use your Plane subclass objects:

<table>
<thead>
<tr>
<th>Module</th>
<th>Plane subclass usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphics driver</td>
<td>Creates the Plane objects.</td>
</tr>
<tr>
<td>IStagecraft module and the IFlashLib module</td>
<td>Renders frames of Flash animation into a Plane object, and uses a Plane object for bitmap caching. Also, associates a Plane object with a display device for output to the user.</td>
</tr>
<tr>
<td>Stream player</td>
<td>Uses a Plane subclass to define the position and size of the video display.</td>
</tr>
<tr>
<td>Image decoder</td>
<td>Decodes an image such as JPEG or PNG into a Plane subclass.</td>
</tr>
</tbody>
</table>

Plane objects are used as output planes and render planes.

output plane  The bitmap used to display each completed frame of animation on the display device. The output plane is typically memory-mapped to the display device, although this implementation is platform-dependent.

render plane  The bitmap on which Flash Lite for the digital home renders each frame of Flash animation. Also, while executing a SWF movie, Flash Lite for the digital home uses temporary render planes to perform bitmap caching as needed.

See also

“Planes and bitmap caching” on page 11

Updated 13 May 2009
**PlaneFactory class**

A PlaneFactory subclass creates and destroys Plane objects. You implement one PlaneFactory subclass for each Plane subclass. You also register each PlaneFactory subclass with the GraphicsDriver module. Registration allows Flash Lite for the digital home to automatically create an instance of your PlaneFactory subclass, and then automatically create instances of the associated Plane subclass as needed when executing a SWF movie.

**I2D class**

The I2D class is the key component for interfacing to your platform’s hardware acceleration for raster operations. Your implementation of this interface supports two-dimensional graphical operations that utilize your hardware’s capabilities and APIs. Each Plane subclass provides an accessor function to return the corresponding I2D subclass. Therefore, when Flash Lite for the digital home uses your Plane subclass, it uses your I2D subclass for blit and fill operations. The blit and fill operations utilize the hardware acceleration provided by your platform to transfer images to the plane.

**IGraphicsDriver class**

Flash Lite for the digital home includes the GraphicsDriver class. The GraphicsDriver class is an implementation of the IGraphicsDriver abstract class. The GraphicsDriver class is a module, loaded by a StageWindow instance when the StageWindow instance loads a SWF file.

The GraphicsDriver module manages the PlaneFactory subclasses. This management includes the following:

- Creating instances of registered PlaneFactory subclasses
- Maintaining a list of PlaneFactory objects
- Using PlaneFactory objects to create and destroy Plane objects

You do not modify the GraphicsDriver class except for providing two private methods. Each of these methods returns a name you specify for a Plane subclass. The GraphicsDriver module uses these names to know which PlaneFactory object to use to create a Plane object.

**Class interaction**

The host application interacts with Flash Lite for the digital home to run Flash Lite applications. Specifically, the host application interacts with the IStagecraft module. Using this interface, the host application creates a StageWindow instance. The StageWindow instance contains an instance of Adobe® Flash® Lite® 3.1. Flash Lite loads the SWF file specified by the host application. The StageWindow instance asks the Graphics Driver module to provide a PlaneFactory object. The PlaneFactory object creates a Plane object.

The following illustration provides a high-level depiction of the call flow involved with creating a Plane object.
When the host application no longer needs a StageWindow instance, the host application asks the IStagecraft module to destroy the StageWindow instance. Upon its destruction, the StageWindow instance asks the Graphics Driver module to destroy the planes. The Graphics Driver module in turn asks the PlaneFactory objects to destroy their plane objects.

See also
“Architecture of Flash Lite for the digital home” on page 2
“Running Flash Lite for the digital home” on page 3

File locations

<table>
<thead>
<tr>
<th>Class</th>
<th>Header file</th>
<th>Implementation file</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGraphicsDriver</td>
<td>include/ae/ddk/graphicsdriver/IGraphicsDriver.h</td>
<td>Not applicable for abstract classes</td>
</tr>
<tr>
<td>GraphicsDriver</td>
<td>source/ae/ddk/graphicsdriver/GraphicsDriver.h</td>
<td>source/ae/ddk/graphicsdriver/GraphicsDriver.cpp</td>
</tr>
<tr>
<td>PlaneFactory</td>
<td>include/ae/ddk/graphicsdriver/IGraphicsDriver.h</td>
<td>Not applicable for abstract classes</td>
</tr>
<tr>
<td>Plane</td>
<td>include/ae/stagecraft/StagecraftTypes.h</td>
<td>Not applicable for abstract classes</td>
</tr>
<tr>
<td>I2D</td>
<td>include/ae/stagecraft/StagecraftTypes.h</td>
<td>Not applicable for abstract classes</td>
</tr>
</tbody>
</table>
User input handling

User input events occur, for example, when a user presses a key on a remote control device or other user input device. A SWF movie running in Flash Lite for the digital home makes program execution choices based on the user input events it receives. A StageWindow instance contains the Flash Lite instance that is executing the SWF movie. The StageWindow instance must be notified about user input events, so that it can pass the events on to its Flash Lite instance. The Flash Lite instance in turn passes the events on to the SWF movie.

If your platform uses a window-based graphical environment, the operating system delivers user input events directly to the active window. In such environments, you typically have a Plane subclass implementation for the output plane that you associate with a window. Therefore, in window-based graphical environments, your Plane subclass is a logical place to handle user input events as follows:

1. The active window receives a user input event.
2. The Plane object associated with the active window receives the event.
3. The Plane object passes the event to its StageWindow instance.
4. The StageWindow instance passes the event to its Flash Lite instance.
5. The Flash Lite instance passes the event to the SWF movie.

For further detail on how your Plane subclass implementation can handle user input events, see “User input handling” on page 12.

If your platform does not use a window-based graphical environment, your Plane subclass implementation is not involved in user input event handling. Instead, the system developer typically writes a user input driver to forward events to the IStagecraft interface.

Implementations included with source distribution

The source distribution for Flash Lite for the digital home includes implementations of the Plane and PlaneFactory classes for some platforms.

<table>
<thead>
<tr>
<th>Plane and PlaneFactory implementation classes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DirectFBPlane DirectFBPlaneFactoryImpl</td>
<td>Use to create either render planes or output planes for a device that supports the DirectFB (Direct Frame Buffer) library. If your platform supports the DirectFB library, use this implementation as provided, modify it, or copy it to use as a starting point.</td>
</tr>
<tr>
<td>MemPlane MemPlaneFactoryImpl</td>
<td>Use to create planes using system memory. Use only as an example or a starting point.</td>
</tr>
</tbody>
</table>

Updated 13 May 2009
OPTIMIZING FLASH LITE FOR THE DIGITAL HOME

The graphics driver

<table>
<thead>
<tr>
<th>Plane and PlaneFactory implementation classes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win32WindowPlane</td>
<td>Use to create Win32 output planes. The Plane object passes user input events received from the Windows API to the associated StageWindow instance. The Plane object also informs its StageWindow instance when the Window has received the focus. Use only as an example or a starting point.</td>
</tr>
<tr>
<td>Win32PlaneFactoryImpl</td>
<td></td>
</tr>
<tr>
<td>X11Plane</td>
<td>Use to create X11 output planes. The Plane object passes user input events received from the X11 API to the associated StageWindow instance. The Plane object also informs its StageWindow instance when the X11 window has received the focus. Use only as an example or a starting point.</td>
</tr>
<tr>
<td>X11PlaneFactoryImpl</td>
<td></td>
</tr>
<tr>
<td>LinuxFBPlane</td>
<td>Use to create output planes on a Linux frame buffer device. Use only as an example or a starting point.</td>
</tr>
<tr>
<td>LinuxFBPlaneFactoryImpl</td>
<td></td>
</tr>
</tbody>
</table>

Only the DirectFB implementations are suitable for a production environment. The Win32, X11, LinuxFB, and MemPlane implementations are not. They are developer tools only. They are not necessarily complete, efficient, or bug free. However, they are useful as examples or starting points for your own implementations.

Each platform-specific concrete Plane subclass derives from the Plane class. However, in the DirectFB implementation, an intermediary abstract Plane subclass called DirectFBPlane derives from the Plane class. Then, the concrete DirectFBPlaneImpl class derives from DirectFBPlane. The intermediary abstract Plane subclass provides an additional publicly accessible method.

Similarly, each platform-specific concrete PlaneFactory class derives from an intermediary abstract class. For example, X11PlaneFactoryImpl derives from X11PlaneFactory which derives from PlaneFactory. In most cases, the header file for the intermediary abstract class adds only a #define for the Plane class name. However, for DirectFBPlaneFactory and MemPlaneFactory, the intermediary subclass provides publicly accessible methods in addition to those defined by the Plane base class. You can use the same coding technique if appropriate to your implementation. For more information, see “Providing intermediary classes for public method accessibility” on page 27.

The platform-specific Plane and PlaneFactory files provided with the source distribution are in these directories:

• include/ae/ddk/graphicsdriver/host
• source/ae/ddk/graphicsdriver/host

Note: Do not put your platform-specific graphics driver files in these directories. For information about where to put your files, see “Building platform-specific drivers and decoders” on page 79.

Implementation tasks overview

To implement a platform-specific graphics driver, do the following high-level tasks:

1 Implement a class that derives from the Plane class, if the source distribution does not provide one to meet your needs.
2 Implement a class that derives from the PlaneFactory class, if the source distribution does not provide one to meet your needs.
3 Implement the GraphicsDriver methods that return the names of your default render plane and default output plane.
4 Implement a class that derives from the I2D class.
5 If yours is a window-based platform, implement user input event handling in your output plane class.
For details about implementing these interfaces, see the class details.

**Plane class details**

**Plane Class definition**
The Plane class hierarchy and methods are given in the following illustration:

![Plane class hierarchy diagram](image)

**Planes and double buffering**
Flash Lite for the digital home uses double buffering to display the SWF movie. Double buffering means that Flash Lite for digital home does the following:

1. Renders each frame of the SWF movie on a render plane.
2. Copies the render plane results to the output plane.

Because the output plane is typically memory-mapped to the display device, double buffering keeps the user from seeing partially rendered animations.

Keep in mind both plane types when designing a new Plane class implementation. Because both the render plane and output plane typically use the same graphics library APIs, they share much of the same implementation. Therefore, you usually create one new PlaneFactory subclass that is the factory for both the render plane and the output plane. Then, during run time, a Plane object specializes itself into either a render plane or output plane. For example, when the PlaneFactory subclass creates a render plane, the PlaneFactory object calls methods of the new render plane object to perform initializations specific to render planes.

The StageWindow instance running the SWF movie uses the render plane and output plane you specify as the defaults. See “IGraphicsDriver class details” on page 21.
Planes and bitmap caching

Flash Lite for the digital home uses bitmap caching when a bitmap image does not change between the frames of a SWF movie. By keeping the bitmap image in a memory cache, Flash Lite for the digital home does not redraw the image in every frame. Rather, it can copy the image. Flash Lite for the digital home uses bitmap caching when:

- A SWF movie uses a bitmap image created from bitmap library items.
- A SWF movie loads a bitmap image with ActionScript.
- A SWF content developer explicitly requested bitmap caching for Movie Clip or Button instances that use complex vector graphics. Typically, developers make these requests for complex vector graphics that don’t update frequently. Because of the bitmap caching request, Flash Lite for the digital home can optimize the performance of displaying, moving, and blending these vector graphics.

Adobe recommends using bitmaps instead of vector graphics for complex image components to speed up SWF movies on many embedded systems. This recommendation applies when hardware-assisted bitmap compositing functions provide better performance than vector animation processed on the main CPU.

When Flash Lite for the digital home performs bitmap caching while executing a SWF movie, it uses a render plane. Flash Lite for the digital home creates and destroys instances of a render plane as needed for bitmap caching. This behavior differs from the use of a render plane in double buffering. For double buffering, one render plane is instantiated for the life of the StageWindow instance and its SWF movie.

Typically, the bitmap caching render plane is instantiated from the same Plane subclass you use for the render plane in double buffering. However, you can define a different Plane subclass for bitmap caching use. Define a separate Plane subclass if you want the bitmap caching render plane to have different characteristics or a different 2D subclass than the render plane used in double buffering. This flexibility is typically useful only during development for doing performance analysis. To specify a separate Plane subclass for the bitmap caching render plane, see the StageWindowsParameter class in include/ae/stagecraft/StageWindow.h.

Plane dimensions and scaling

The system developer can configure a StageWindow instance to use explicit dimensions for the render plane and the output plane. However, by default, Flash Lite for the digital home handles your planes’ dimensions as follows:

- The render plane is set to the dimensions of the Stage of the SWF movie.
- The output plane is set to the dimensions of the render plane.
- If the resulting render and output plane dimensions are bigger than the maximum output plane dimensions (for example, the device screen size), then Flash Lite for the digital home scales the render plane to fit. The aspect ratio is preserved.

Note: The system developer can design a StageWindow instance to not scale down to fit the maximum output plane dimensions. If so, and the output plane dimensions are bigger than the maximum output plane dimensions, all the content is not visible to the user. When the StageWindow instance requests your PlaneFactory object to create the Plane object, code the PlaneFactory object to decide whether to create the Plane object when all the content will not be visible to the user.
In most cases, the system developer lets the default behavior set the dimensions of the render plane and output plane to be the same. The Flash Lite instance renders high-quality output at any size. This rendering includes vector graphics operations and compositing bitmaps. The compositing operations utilize your hardware accelerator’s blit and fill operations. The vector graphics, however, use the system processor. For large plane sizes, especially for SWF movies that do lots of vector graphics, the system processor usage can be high. To reduce system processor usage, the system developer can specify a render plane size that is smaller than the output plane size. However, the resulting frames are not as high in rendering quality. Moreover, the blit operation you provide in your I2D implementation must be able to enlarge (stretch blit) the bitmap to the output plane size. For more information, see “Blit() method” on page 19.

**User input handling**

In window-based graphical environments, your Plane subclass for the output plane is a logical place to handle user input events. In your Plane subclass implementation, do the following:

1. Store as a data member a pointer to the Plane object’s StageWindow instance. This pointer is passed to the PlaneFactory object in its CreateOutputPlane() method. In your Plane subclass, implement a public method that the PlaneFactory object calls to pass in the pointer to the StageWindow instance.

2. Receive user input events by using your platform’s APIs for accessing the active window.

3. Call the corresponding StageWindow method to handle each event received.

The StageWindow class provides these methods to handle user input events (defined in include/ae/stagecraft/StageWindow.h):

- `DispatchKeyDown()`
- `DispatchKeyUp()`
- `DispatchMouseButtonDown()`
- `DispatchMouseButtonUp()`
- `DispatchMouseMove()`
- `DispatchScrollWheelScroll()`

Your system developer provides a mapping of keys on remote controls and other input devices. However, the key mapping does not affect the Plane object’s role in passing events to the StageWindow instance. The information contained in the event, such as the key pressed or the mouse coordinates, is forwarded as parameters to the StageWindow instance event handling methods.

The source distribution provides two examples of handling user input events in a Plane object:

- `Win32WindowPlane`: See source/ae/ddk/graphicsdriver/host/Win32PlaneFactoryImpl.cpp
- `X11Plane`: See source/ae/ddk/graphicsdriver/host/X11PlaneFactoryImpl.cpp

Handling user input events in non-windowing platform is outside the scope of this document. However, an example is in the DirectFB implementation. See these files:

- `source/ae/ddk/graphicsdriver/host/DirectFBPlaneFactoryImpl.cpp`
- `source/ae/ddk/graphicsdriver/host/TTYUserInputDriver.h` and `TTYUserInputDriver.cpp`

Updated 13 May 2009
Plane class methods

When you implement your Plane subclass, you provide implementation for the following public methods. For detailed definitions of return values and parameters of the Plane class methods, see include/ae/stagecraft/StagecraftTypes.h.

GetClassName() method
This method returns a string that is the name you define for your Plane subclass. This string is the same as the string you return in your GraphicsDriver class implementation of GetDefaultOffScreenPlaneClass() and GetDefaultOutputPlaneClass().

A common use of GetClassName() is in the Blit() method of your I2D subclass implementation. The Blit() method is passed a Plane object pointer as its source plane parameter. Because the behavior of the Blit() method usually depends on the type of the source plane, Blit() calls the Plane object’s GetClassName().

Other objects and modules also use GetClassName(). For example, a StreamPlayer object uses GetClassName() to determine the class of its output plane.

GetColorFormat() method
This method returns the ColorFormat of the Plane object. The ColorFormat is an enumeration which defines possible bitmap pixel formats for Plane objects. For example, a common color format is ARGB8888.

For render planes, the PlaneFactory receives a ColorFormat parameter in its CreatePlane() method. The ColorFormat parameter specifies the pixel format of the plane to be created. Provide a method or constructor in your Plane subclass to receive the value from the PlaneFactory object. The GetColorFormat() method returns the same ColorFormat value. The Plane object either stores the ColorFormat value, or GetColorFormat() dynamically determines the value.

For render planes, the ColorFormat value that Flash Lite for the digital home passes to the PlaneFactory object’s CreatePlane() method is always ARGB8888. For temporary render planes used for bitmap caching, Flash Lite for the digital home passes either the value ARGB8888 or kCLUT8, an indexed-color bitmap.

For output planes, the system developer determines the ColorFormat. However, in many platforms the ColorFormat for the output plane is also ARGB8888.

A common use of GetColorFormat() is in the Blit() method of your I2D subclass implementation. The Blit() method uses the color format of its source plane and destination plane.

Note: When the bitmap caching render plane uses kCLUT8, the blit from the bitmap caching plane to the render plane, which uses ARGB8888, requires a format conversion. Implement the Blit() method for the I2D object of the render plane to perform the conversion using platform-specific hardware accelerators.

For the complete ColorFormat enumeration, see include/ae/stagecraft/StagecraftTypes.h.

GetDims() method
This method returns the dimensions of the Plane object. The dimensions are the width and height of the plane bitmap in pixels.

Your Plane subclass implementation is responsible for determining and storing the plane’s dimensions. For example, you can hardcode the dimensions, or you can provide a method or constructor to receive the value from another object, such as the PlaneFactory object.

Updated 13 May 2009
A common use of `GetDims()` is in the `Blit()` method of your I2D subclass implementation. The `Blit()` method uses the dimensions of its source plane.

Other objects and modules also use `GetDims()`. For example, the Flash Lite module and the StageWindow instance use the dimensions of the render plane.

**GetPalette() method**

This method provides a pointer to an array of Color objects which define the available colors of the plane. The Color class is defined in `include/ae/stagecraft/StagecraftTypes.h`. This method provides the array pointer only if the plane uses indexed colors. If the plane does not use indexed colors, `GetPalette()` sets the pointer to `null` and returns `false`.

Your Plane subclass implementation also provides a `SetPalette()` method. A common use of `GetPalette()` is in the `Blit()` method of your I2D subclass implementation. The `Blit()` method often gets the color palette of its source plane.

**GetPixelAt() method**

This method returns a Color object representing the color of the pixel at the location specified by the parameters. However, `GetPixelAt()` is used only for testing Flash Lite for the digital home. Therefore, in your implementation, return a Color object constructed with the default constructor:

```
return Color();
```

**GetRowBytes() method**

This method returns the number of bytes used to store one scan line of the plane in memory. This value is the number of bytes you add to the address of a pixel to get to the same pixel one line below. Your Plane subclass implementation is responsible for determining and storing this value. For example, you can hardcode its value, or you can provide a method or constructor to receive the value from another object, such as the PlaneFactory object, or retrieve its value from your platform libraries.

A common use of `GetRowBytes()` is in the `Blit()` method of your I2D subclass implementation. The `Blit()` method often uses the number of bytes in a row of its source plane, and `Fill()` method often uses the number of bytes in a row of its destination plane. Other objects and modules, such as the Image Decoder Module and the Flash Lite Module also use `GetRowBytes()`.

**Get2DInterface() method**

This method returns a pointer to an I2D subclass object. Your Plane subclass implementation is responsible for the following:

- Creating the I2D subclass object and storing a pointer to it.
- Destroying the I2D subclass object when the Plane object is destructed.

For more information about a plane's I2D subclass object, see "I2D class details" on page 18.
**LockBits() method**

This method returns a pointer to the memory representing the bitmap pixels of the plane. Depending on your implementation, the plane’s bitmap memory is allocated by the PlaneFactory subclass method CreatePlane() or by the Plane subclass object when it is instantiated. The Flash Lite module calls LockBits() to get the memory pointer. Then, the Flash Lite module directly manipulates the memory when rendering the Flash animation. The Flash Lite module calls UnlockBits() when it is done accessing the memory.

The word “lock” in LockBits() refers to ensuring that the plane’s bitmap pixels are “locked” in memory and so can be safely accessed directly. Some graphics libraries sometimes move bitmap data around in memory for memory-management purposes. The Flash Lite module, however, requires that the memory is locked in place while it renders the Flash animation. Your Plane subclass implementation must allow multiple calls to LockBits() before the corresponding calls to UnlockBits(). Make sure that the number of calls to LockBits() and UnlockBits() match up.

The LockBits() method returns NULL only if a catastrophic failure occurs. Flash Lite for the digital home does not correctly render the Flash animation in this case.

**Note:** The Flash Lite module calls LockBits() for render planes only. The LockBits() implementation for an output plane is never executed. However, the software implementation of I2D requires a LockBits() implementation for both the render plane and the output plane. Therefore, if you test with the I2D software implementation, implement LockBits() for the output plane, too. The I2D software implementation is in source/ae/ddk/graphicsdriver/host/I2DImpl.cpp.

Flash Lite for the digital home does not require thread-safety in LockBits(). However, make your implementation of LockBits() thread-safe if your platform requires it. For example, some graphics driver implementations perform memory management to achieve better memory utilization. This memory management sometimes involves a separate thread which moves around the blocks of bitmap memory. The separate thread means that LockBits() must be thread-safe. For example, make sure you return a valid pointer regardless of the memory management activities in a separate thread.

**OnRectUpdated() method**

This method is a notifier method. Flash Lite for the digital home calls OnRectUpdated() to notify the plane that it has updated the plane.

- For a render plane, Flash Lite for the digital home calls OnRectUpdated() when it has finished rendering a frame of Flash animation. This call indicates that the frame is ready for presentation. Typically, a render plane uses OnRectUpdated() only if no output plane exists. That is, the platform does not use double buffering. For example, some platforms use a page flipping implementation. In a page flipping implementation, two bitmaps alternate being the display bitmap and the backing memory bitmap. The OnRectUpdated() method performs the page flip. That is, it points the display to the newly updated backing memory.

- For an output plane, Flash Lite for the digital home calls OnRectUpdated() when it has blitted the render plane to the output plane. Typically, an implementation uses OnRectUpdated() to update the display if the platform does not have a memory-mapped display device. For example, in the Win32 implementation, OnRectUpdated() calls the Windows API InvalidateRect(). This Windows API causes Windows to copy the Win32WindowPlane object’s bitmap to the display.

A parameter of OnRectUpdated() indicates the subrectangle that was updated. The subrectangle can specify the whole plane.
An alternative to using `OnRectUpdated()` is the `StageWindowNotifier` class. Your host application can implement the `StageWindowNotifier` interface to receive plane update notifications. For example, use the `StageWindowNotifier` if you are using a `Plane/PlaneFactory` implementation from the source distribution which you do not want to modify. However, using `OnRectUpdated()` keeps all `Plane`-related code in the `Plane` object. Work with your system developer to determine the right solution for your platform.

**SetPalette() method**

This method is used to set an array of `Color` objects that define the available colors of the plane. The `Color` class is defined in `include/ae/stagecraft/StagecraftTypes.h`. This method sets up the array only if the plane uses indexed colors. This method returns `false` under these circumstances:

- The plane does not use indexed colors.
- The number of colors requested for the palette is greater than the maximum number of colors the plane allows for the palette.

Your implementation also provides a `GetPalette()` method.

An image decoder calls `SetPalette()`, for example, while decoding a PNG image that is stored as 8-bit indexed color. `SetPalette()` builds the palette for the plane during image decoding.

**UnlockBits() method**

The Flash Lite module calls this method when it is done directly accessing the plane’s bitmap memory. The caller balances calls to `LockBits()` and `UnlockBits()`. For more information, see “`LockBits()` method” on page 15.

**PlaneFactory class details**

**PlaneFactory class definition**

The `PlaneFactory` class hierarchy and methods are given in the following illustration:
**PlaneFactory registration**

Register your PlaneFactory subclass with the GraphicsDriver module. Registration allows the Graphics Driver module to use your PlaneFactory subclass when a StageWindow instance needs to create a render plane and output plane. To register your PlaneFactory subclass, include the following line of code in the implementation file of your PlaneFactory subclass:

```c
static RegisterFactory<PlatformPlaneFactory> registerFactory;
```

In this line of code, PlatformPlaneFactory is the class name of the PlaneFactory subclass. The subclass PlatformPlaneFactory derives from the PlaneFactory class.

When a SWF movie executes, the StageWindow instance requests the GraphicsDriver module to create the planes. The GraphicsDriver module calls its methods `GetDefaultOffscreenPlaneClass()` and `GetDefaultOutputPlaneClass()` to determine which PlaneFactory subclass objects to use to create the planes. For more information on these methods, see “IGraphicsDriver class details” on page 21.

**PlaneFactory class methods**

**CreatePlane() method**

This method creates a plane to be used as a render plane. The plane the method creates is of the Plane subclass type that corresponds to the PlaneFactory subclass. CreatePlane() parameters include the following:

- The dimensions of the plane.
- The color format of the plane, expressed as a `ColorFormat` enumeration value such as `kARGB8888`.
- A pointer to the StageWindow object calling `CreatePlane()`. The pointer to the StageWindow object is valid for the life of the plane object. This pointer must not be `NULL`, except possibly during unit testing.

**CreateOutputPlane() method**

This method creates a plane to be used as an output plane. The plane the method creates is of the Plane subclass type that corresponds to the PlaneFactory subclass. CreatePlane() parameters include the following:

- A Rect object that defines the plane’s x and y coordinates and width and height. The x and y coordinates are the upper leftmost corner of the Rect object relative to the upper leftmost corner of the output screen. The Rect class definition is in `include/ae/stagecraft/StagecraftTypes.h`.
- A flag indicating whether to position the plane as directed by the Rect object parameter, or whether to use a default position. When the flag indicates to use a default position, the `CreatePlane()` method decides how to position the plane. Depending on your PlaneFactory subclass implementation, the default position might be to center the plane, or to tile the plane.
- A pointer to the StageWindow object calling `CreatePlane()`. The pointer to the StageWindow object is valid for the life of the plane object. In a window-based graphical environment in which the output plane receives user input events, use the StageWindow pointer to pass the events to the StageWindow object. This pointer must not be `NULL`, except possibly during unit testing.
- The title of the window containing the output plane. Some platforms allow a window to have a title. In those cases, use this parameter to pass the title to the output plane.
• A pointer to a compatible plane. Use this pointer to provide access to an existing plane which the PlaneFactory object can use to define characteristics of the new plane. For example, pass a pointer to the render plane. Then, CreateOutputPlane() creates an optimal output plane for the blits from the render plane to the output plane. The output plane gets the same characteristics, such as color format, as the render plane. The compatible color formats results in faster blits.

  Note: Currently, the render plane always uses color format ARGB8888, so creating the output plane from a compatible plane is not necessary to create an output plane with the same color format.

DestroyPlane() method
This method destroys a plane. The StageWindow instance calls DestroyPlane() when the host application is done with the StageWindow instance. Also, the Flash Lite instance calls DestroyPlane() when it has finished with a temporary render plane. For example, if a MovieClip is configured for bitmap caching, when ActionScript destroys the MovieClip, the Flash Lite instance destroys the temporary render plane it used for bitmap caching.

GetClassName() method
This method returns a string that is the name you define for the Plane subclass that this PlaneFactory subclass creates. This string is the same as the string you return in your GraphicsDriver class implementation of GetDefaultOffScreenPlaneClass() and GetDefaultOutputPlaneClass().

The Graphics Driver module uses this name to identify a PlaneFactory subclass object in its list of PlaneFactory objects.

GetMaxOutputPlaneDims() method
This method returns the maximum dimensions of an output plane. The return value is a Dims object. The Dims class is defined in include/ae/stagecraft/StagecraftTypes.h. The returned dimensions typically correspond to the device screen size.

Flash Lite for the digital home uses the return value to automatically scale down the render plane to fit into the output device. If your platform does not use the automatic downscale feature, GetMaxOutputPlaneDims() can return a Dims object in which both the width and height have the value 0. Similarly, if your graphics library does not expose a device screen size value, return Dims(0,0).

For more information, see “Plane dimensions and scaling” on page 11.

I2D class details
The I2D class hierarchy and methods are given in the following illustration:
Your Plane subclass provides an accessor function to return an instance of the corresponding I2D subclass. Flash Lite for the digital home uses the I2D subclass object to utilize the platform’s hardware acceleration capabilities to do the following:

- Transfer bitmap images (blit images) from a source plane to a destination plane.
- Fill a rectangle or subrectangle of a plane with a specified color.

## I2D class methods

### Blit() method

This method blits a bitmap image from a source plane to a destination plane. One of the parameters to `Blit()` is a pointer to the source plane. The destination plane is the Plane subclass object from which Flash Lite for the digital home gets the I2D subclass object.

When the destination plane is a render plane, the source plane is a Plane subclass object used internally by the Flash Lite instance for bitmap caching. The source plane can also contain a decoded JPEG or PNG image that was decoded with the image decoder module. When the destination plane is an output plane, the source plane is a render plane that contains a frame that is ready for display to the user. The source Plane can also contain a decoded JPEG or PNG image that was decoded with the ImageDecoder module.

In this method, you interface with your hardware acceleration hardware or APIs. The parameters to `Blit()` specify how to scale, rotate, blend, and transform the source plane into the destination plane. The parameters passed to `Blit()` include the following:

- A pointer to the source plane object.
- A `SrcBlitInfo` structure. This structure is defined in `include/ae/stagecraft/StagecraftTypes.h`. The structure includes `x` and `y` that are 16.16 fixed-point numbers. These values specify the upper-left corner of the subrectangle to blit, relative to the upper-left corner of the source plane.

The structure also includes `dx` and `dy` values, which are also 16.16 fixed-point numbers. The values of `dx` and `dy` specify the inverse scaling factors to use in the blit operation. Calculate the source subrectangle dimensions by multiplying the respective inverse scaling factor by the dimensions of the destination subrectangle. If each of `dx` and `dy` equals 1.0, then the blit operation performs no scaling.
Note: Blit() uses inverse scaling factors because the destination subrectangle dimensions are always whole numbers. However when the dimensions are multiplied by the inverse scaling factor, which is a fraction, the resulting source subrectangle dimensions can have fractional components. Blit() must handle the fractional parts correctly. If not handled correctly, a small wave effect sometimes appears in the animations.

- A Rect object that defines a subrectangle of the destination plane. The Rect class is defined in include/ae/stagecraft/StagecraftTypes.h. The specified subrectangle is the target of the blit operation.

- A pointer to a Color object, which specifies an overlay color, if any. The Color class is defined in include/ae/stagecraft/StagecraftTypes.h. You use this parameter to tint the entire blitted region with the overlay color. You use the overlay color during development only to visually determine which objects the I2D implementation is rendering, and which objects the Flash Lite instance is rendering. Enable the overlay color using the showblit Shell command. For more information, see the --useShell command-line option in Getting Started with Adobe Flash Lite for the digital home.

- A pointer to a Transformation structure. The Transformation structure is defined in include/ae/stagecraft/StagecraftTypes.h. Its data members include the following:
  
  m_matrix This Matrix structure contains four fixed point values: a, b, c, and d. This matrix represents a standard computer graphics 2x2 transformation matrix. For example, if a is negative, reflect the source plane using a horizontal mirror operation. If d is negative, use a vertical mirror operation. If b or c are non-zero, use a shearing operation. However, this matrix is an inverse transformation matrix. An inverse transformation matrix means that the values are applied to the destination coordinates to compute the source coordinates.

  m_bHasTransparency If m_bHasTransparency is true, perform an alpha blend operation while blitting the source onto the destination. In this case, the source bitmap pixel colors are pre-multiplied with the alpha value of the respective pixel. Pre-multiplying the alpha value facilitates higher performance of the blend operation.

  If m_bHasTransparency is false, a copy operation is sufficient. Some platforms perform faster with a copy as compared to an alpha blend.

  m_colorOffset and m_colorMultiplier Use these values to compute the source pixel color as follows:

  New red value = (old red value * redMultiplier) + redOffset  
  New green value = (old green value * greenMultiplier) + greenOffset  
  New blue value = (old blue value * blueMultiplier) + blueOffset  
  New alpha value = (old alpha value * alphaMultiplier) + alphaOffset  

  The range of the m_colorOffset values is -255 to +255. The m_colorMultiplier is a fixed point 8.8 value, which means the range is 0.0 to 255.99. Clamp the new computed value to the range of 0 to 255.

  m_bHasColorTransform If m_bHasColorTransform is true, use the m_colorOffset and m_colorMultiplier in the blit operation. If m_bHasColorTransform is false, ignore m_colorOffset and m_colorMultiplier.

  m_bHasAlphaOnly If m_bHasAlphaOnly is true, ignore m_colorOffset and use only the alpha value, not the red, green, or blue values, of m_colorMultiplier.

Blit() returns true for success, and false for failure. Returning true indicates to the caller only that the blit operation is acceptable, not that it has completed. Your hardware acceleration implementation determines whether the actual blit operation is synchronous or asynchronous.

Return false if the parameters request a blit that is too complex for your implementation. When Blit() returns false, the Flash Lite instance performs the blit operation in software. This software implementation results in a correct image but is less efficient than blitting with hardware.
Fill() method
This method fills a rectangle or subrectangle of a destination plane with a specified color. In this method, you interface with your hardware acceleration hardware or APIs.

The parameters passed to Fill() include the following:

- A Rect object that defines a subrectangle of the destination plane. The Rect class is defined in include/ae/stagecraft/StagecraftTypes.h. The Rect object specifies the subrectangle to fill with the specified color.
- A Color object that defines the fill color. Color is defined in include/ae/stagecraft/StagecraftTypes.h.
- A flag indicating whether to perform a standard alpha blend using the fill color on the destination subrectangle. The fill color values have pre-multiplied alpha values. If this flag is false, fill the destination subrectangle with the fill color without any consideration of the existing color value of the destination.

Fill() returns true for success, and false for failure. Returning true indicates to the caller only that the fill operation is acceptable, not that it has completed. Your hardware acceleration implementation determines whether the actual fill operation is synchronous or asynchronous.

Return false if the parameters request a fill operation that is too complex for your implementation. When Fill() returns false, the Flash Lite instance performs the fill operation in software. This software implementation results in a correct image but is less efficient than performing the fill operation with hardware.

Flush() method
This method returns only after all previously called blit and fill operations have completed. When Flash Lite for the digital home wants to directly access the bitmap of a plane involved in a blit or fill operation, it first calls Flush(). Because Flush() does not return until all the previous blit and fills operations are completed, Flash Lite for the digital home can safely follow a call to Flush() with operations that directly access the bitmap of one of the involved planes.

For example, Flash Lite for the digital home calls an output plane’s Blit() method to transfer the pixels from the render plane. Flash Lite for the digital home then begins to render the next frame of the render plane. By first calling the output plane’s Flush(), Flash Lite for the digital home knows that the transfer is complete, and that it can safely overwrite the render plane.

Similarly, Flash Lite for the digital home calls a render plane’s Blit() and Fill() methods to transfer pixels from a temporary render plane used in bitmap caching to the render plane for compositing the frame. Calling Flush() on the destination plane ensures that the transfer is complete.

IGraphicsDriver class details
The IGraphicsDriver class hierarchy and methods are given in the following illustration:
You use the GraphicsDriver class derived from the IGraphicsDriver abstract class. The GraphicsDriver class implements the virtual methods of the IGraphicsDriver class, including the `RegisterPlaneFactory()` method. For more information, see “PlaneFactory registration” on page 17.

You provide the implementation of only two static private methods: `GetDefaultOffscreenPlaneClass()` and `GetDefaultOutputPlaneClass()`. The Graphics Driver module uses these methods to determine which PlaneFactory object to use to create the planes for a StageWindow instance. These methods return a string that is the name of a Plane subclass. For example:

```cpp
// In a header file:
#define MEM_PLANE_CLASS_NAME "MemPlane"
#define LINUXFB_PLANE_CLASS_NAME "LinuxFBPlane"

// In a .cpp file:
const char * GraphicsDriver::GetDefaultOffscreenPlaneClass()
{
    return MEM_PLANE_CLASS_NAME;
}

const char * GraphicsDriver::GetDefaultOutputPlaneClass()
{
    return LINUXFB_PLANE_CLASS_NAME;
}
```

Do not modify the GraphicsDriver.cpp file. Add your implementations of `GetDefaultOffscreenPlaneClass()` and `GetDefaultOutputPlaneClass()` to a new separate.cpp file. For example, name the new file `GraphicsDriverMyPlatform.cpp`. 

Updated 13 May 2009
Sample implementation walkthrough

As an example, consider a graphics library called FakeGraphicsLib. This example shows how to implement Plane, PlaneFactory, and I2D subclasses to use FakeGraphicsLib.

Fake graphics library class

The FakeGraphicsLib header file contains the following:

```cpp
// FakeGraphicsLib.h
class FakeGraphicsLib {
    public:
        struct Dims { int width; int height; };
        struct Rect { int x; int y; int width; int height; };
    public:
        static void Initialize();
        static void Shutdown();
        static char * AllocScreenBits(const Rect & rect);
        static char * AllocBits(const Dims & dims);
        static void FastBlit(char * pSourceBits, char pDestBits,
                                const Rect & sourceRect, const Rect & destRect);
        static void FreeBits(char * pBits);
    };

    This graphics library provides the following:
    • the AllocScreenBits() method to allocate an onscreen graphics buffer and return a pointer to the buffer.
    • the AllocBits() method to allocate an offscreen graphics buffer and return a pointer to the buffer.
    • the FreeBits() method to deallocate a graphics buffer.
    • the FastBlit() method to provide the blit operations.
```

Sample PlaneFactory subclass declaration

Implement a PlaneFactory subclass to create plane objects that interface with the FakeGraphicsLib. Name the PlaneFactory subclass FakeGraphicsLibPlaneFactoryImpl. The subclass derives from the PlaneFactory class. The Impl suffix in the derived class name is a convention that allows for adding an abstract class in the class hierarchy between PlaneFactory and FakeGraphicsLibPlaneFactoryImpl. An intermediary abstract class is convenient for adding public methods. For more information, see “Providing intermediary classes for public method accessibility” on page 27.


Begin the source file with the following:
// Include the interface for the Fake Graphics Library.
// Also include the interface for the I2D subclass,
// and the Graphics Driver module interface.
// The Graphics Driver module interface is needed for PlaneFactory registration.
#include <fakegraphics/FakeGraphicsLib.h>
#include "I2DImpl.h"
#include "../GraphicsDriver.h"

// Use these namespaces for convenience.
using namespace ae::stagecraft;
using namespace ae::ddk::graphicsdriver;

// Provide a name for the PlaneFactory class.
// Use the name in PlaneFactory registration.
#define FAKEGRAPHICS_PLANE_CLASS_NAME "fakeGraphicsPlane"

Note: Typically, the #define for the class name is in a header file, not a source file. Other source files can include the header file, and therefore can use the class name definition. This example is a simplification.

Next, the FakeGraphicsLibPlaneFactoryImpl.cpp contains the class declaration for the FakeGraphicsPlaneFactoryImpl class. Because this subclass is concrete, it declares an implementation of each pure virtual function in the abstract PlaneFactory class.

class FakeGraphicsPlaneFactoryImpl : public PlaneFactory
{
public:
    FakeGraphicsPlaneFactoryImpl();
    virtual ~FakeGraphicsPlaneFactoryImpl();

public:
    virtual const char * GetClassName() const;
    virtual Plane * CreatePlane(const Dims & dims, ColorFormat colorFormat, StageWindow * pStageWindow);
    virtual Plane * CreateOutputPlane(const Rect & rect, bool bDefaultPosition, Plane * pCompatiblePlane, StageWindow * pStageWindow, const char * pTitle);

    virtual Dims GetMaxOutputPlaneDims();
    virtual void DestroyPlane(Plane * pPlane);
};

The next line in the FakeGraphicsLibPlaneFactoryImpl.cpp file registers the PlaneFactory subclass with the Graphics Driver module. The Graphics Driver module uses a C++ template to implement PlaneFactory registration. The implementation requires only one line of code for registration. This line declares the static variable registerFactory. This static variable causes a new entry to be created in a linked list of available PlaneFactory objects when the Graphics Driver module is constructed.

static RegisterFactory<FakeGraphicsPlaneFactoryImpl> registerFactory;

**Sample Plane subclass declaration**


Updated 13 May 2009
class FakeGraphicsPlane : public Plane
{
  public:
    FakeGraphicsPlane()
    {
      m_pI2D = AE_NEW I2DImpl(this);
      m_pBits = NULL;
    }

    virtual ~FakeGraphicsPlane()
    {
      AE_DELETE(m_pI2D);
      if (m_pBits) FakeGraphicsLib::FreeBits((char *) m_pBits);
    }

    public:
    virtual const char * GetClassName() const
    { return FAKEGRAPHICS_PLANE_CLASS_NAME; }
    virtual Dims GetDims() const
    { return m_dims; }
    virtual ColorFormat GetColorFormat() const
    { return m_colorFormat; }
    virtual u32 GetRowBytes() const
    { return m_nRowBytes; }
    virtual Color GetPixelAt(u32 x, u32 y) const
    { return Color(); }
    virtual bool GetPalette(const Color * & pPaletteToSet, u32 & nNumEntriesToSet) const
    { return false; }
    virtual bool SetPalette(Color * pPalette, u32 nNumEntries)
    { return false; }
    virtual u8 * LockBits()
    { return m_pBits; }
    virtual voidUnlockBits()
    { }
    virtual I2D * Get2DInterface()
    { return m_pI2D; }
    virtual void OnRectUpdated(const Rect & updateRect)
    { }

  public:
    ColorFormat m_colorFormat;
    u32 m_nRowBytes;
    Dims m_dims;
    u8 * m_pBits;
    I2DImpl * m_pI2D;
};

Note: This simple Plane subclass uses inline virtual function declarations only as a convenience for this example. These declarations can cause compilers to generate extra code in some circumstances.

Although simple, the FakeGraphicsPlane class illustrates these points:

- The GetClassName() method returns the name of the class defined at the top of the file.
- The GetPixelAt() method returns an empty Color object. Flash Lite for the digital home uses this method only for its own unit testing.
• The constructor creates a I2D subclass object. The Get2DInterface() method returns a pointer to the object. The destructor destroys the object. The I2DImpl class is provided with the source distribution. It provides a software implementation of the Blit() and Fill() methods. In an actual implementation, after initial testing, you replace the I2DImpl class with your own platform-specific I2D subclass. For more information, see “I2D class details” on page 18.

• Memory allocation and deallocation use the AE_NEW() and AE_DELETE() macros. For details on these macros, see “Common types and macros” on page 75.

Sample PlaneFactory subclass method definitions


```cpp
FakeGraphicsPlaneFactoryImpl::FakeGraphicsPlaneFactoryImpl()
{
    FakeGraphicsLib::Initialize();
}

FakeGraphicsPlaneFactoryImpl::~FakeGraphicsPlaneFactoryImpl()
{
    FakeGraphicsLib::Shutdown();
}

cnst char * FakeGraphicsPlaneFactoryImpl::GetClassName() const
{
    return FAKEGRAPHICS_PLANE_CLASS_NAME;
}

Plane * FakeGraphicsPlaneFactoryImpl::CreatePlane(const Dims & dims,
    ColorFormat colorFormat,
    StageWindow * pStageWindow)
{
    if (colorFormat != kARGB8888) return NULL;
    FakeGraphicsLib::Dims fakeDims = { dims.w, dims.h }; pPlane->m_pBits = (u8 *) FakeGraphicsLib::AllocBits(fakeDims);
    return pPlane;
}

Plane * FakeGraphicsPlaneFactoryImpl::CreateOutputPlane(const Rect & rect,
    bool bDefaultPosition,
    Plane * pCompatiblePlane,
    StageWindow * pStageWindow,
...
const char * pTitle)
{
    FakeGraphicsLib::Rect fakeRect = { rect.x, rect.y, rect.w, rect.h }; 
    pPlane->m_pBits = (u8 *) FakeGraphicsLib::AllocScreenBits(fakeRect);
    return pPlane;
}

Dims FakeGraphicsPlaneFactoryImpl::GetMaxOutputPlaneDims()
{
    // Returns 0,0 dimensions in this simple example.
    // For a real implementation, return the maximum size of the output plane.
    return Dims(0, 0);
}

void FakeGraphicsPlaneFactoryImpl::DestroyPlane(Plane * pPlane)
{
    AE_DELETE(pFakeGraphicsPlane);
}

Although simple, the FakeGraphicsPlaneFactory class illustrates these points:

- The constructor initializes the graphics library. The destructor shuts down the graphics library. For more information about how and when to initialize your graphics library, see “Initializing the Graphics Driver module” on page 28.
- Both CreatePlane() and CreateOutputPlane() allocate a FakeGraphicsPlane object. In an actual implementation, however, these methods typically allocate different types of Plane subclasses. The methods then illustrate using the graphics library to allocate the bitmap for the planes. In this case, the dimensions of the plane default to the dimensions of the Stage of the SWF movie.
- Memory allocation and deallocation use the AE_NEW() and AE_DELETE() macros. For details on these macros, see “Common types and macros” on page 75.

**Implementation considerations**

When you implement your platform-specific classes for the Graphics Driver module, consider the following guidelines.

**Providing intermediary classes for public method accessibility**

You derive your Plane and PlaneFactory subclasses from the abstract Plane and PlaneFactory classes. Depending on your platform needs, you sometimes want to expose other public interfaces besides the ones in the base abstract classes. To do so, derive an intermediary abstract subclass from the Plane or PlaneFactory class. The abstract subclass declares other public interfaces you want to expose. Then, derive your implementation subclass from the intermediary abstract subclass.

For example, the source distribution provides a MemPlaneFactory abstract subclass derived from PlaneFactory. The MemPlaneFactory subclass declares an additional CreatePlane() method:
virtual ae::stagecraft::Plane * CreatePlane(u8 * pBits, 
    const ae::stagecraft::Dims & dims, 
    u32 nRowBytes, 
    ae::stagecraft::ColorFormat colorFormat) = 0;

This overloaded `CreatePlane()` method allows the client of the MemPlaneFactory subclass to create a plane from an existing block of memory that defines a bitmap. The client passes a pointer to the memory in the first parameter. The MemPlaneFactoryImpl concrete subclass derives from the intermediary MemPlaneFactory abstract subclass, and implements the overloaded `CreatePlane()` along with all the other abstract methods.

A PlaneFactory intermediary abstract subclass for your platform can similarly define an overloaded `CreatePlane()` method. For example, the platform-specific method can allow the client to create a plane from an existing handle to a graphics library data structure that has already been created.

Besides overloading `CreatePlane()`, the intermediary subclass can declare other interfaces that allow the client to access structures and handles specific to the platform’s graphics library. For example, if your graphics library supports a mechanism for decoding compressed image formats directly into a graphics plane, add a method to get the handle to the graphics plane. Then, a platform-specific image decoder module can use the method to get the handle and access the graphics plane directly.

**Starting with the I2D software implementation**

The source distribution provides a software implementation of the I2D class. The subclass is named I2DImpl. The .cpp and .h files for I2DImpl are in the directory source/ae/ddk/graphicsdriver/host/.

Use I2DImpl when you first set up your platform-specific Plane subclass. Using I2DImpl allows you to test your Plane subclass interfaces without your hardware acceleration libraries. The I2DImpl `Blit()` method implementation uses the `LockBits()` methods of the source and destination planes to get a pointer to the planes’ bitmap memory. Then the method manipulates the bitmaps directly. Although good for initial testing, this implementation is not optimized for speed. After you verify your Plane subclass interfaces, substitute an I2D subclass for your platform’s hardware accelerators.

**Initializing the Graphics Driver module**

To run a SWF movie, a host application creates a StageWindow instance. The StageWindow instance loads the Graphics Driver module. When the Graphics Driver module loads, its initialization includes instantiating all its registered PlaneFactory subclasses. Therefore, your PlaneFactory subclass constructor is executed.

In the FakeGraphicsLib example, the PlaneFactory subclass constructor creates the FakeGraphicsLib object. However, a platform sometimes needs to initialize the graphics driver library *before* creating any StageWindow instances. Later, when Flash Lite for the digital home creates the Plane object, the PlaneFactory object requires an interface pointer to the graphics driver library to create the Plane object. Use the following implementation steps for this scenario:

1. In your host application initialization code, acquire a pointer to the Graphics Driver module.

   ```
   ae::ddk::graphicsdriver::IGraphicsDriver * pGraphicsDriver;
   pGraphicsDriver = (ae::ddk::graphicsdriver::IGraphicsDriver *)
       IAEKernel::GetKernel()->AcquireModule("IGraphicsDriver");
   ```

2. Use the pointer to the Graphics Driver module to acquire a pointer to your PlaneFactory subclass object. In this example the PlaneFactory subclass is PlatformPlaneFactory, and the Plane subclass name is a string defined by `PLATFORM_PLANE_CLASS_NAME`.

   ```
   PlatformPlaneFactory *pPlatformFactory = (PlatformPlaneFactory *)
       pGraphicsDriver->AcquirePlaneFactory(PLATFORM_PLANE_CLASS_NAME);
   ```

Updated 13 May 2009
3 Use the PlaneFactory subclass pointer to access a public method you defined for passing a graphics library interface pointer to your PlaneFactory instance.

   pPlatformFactory->SetGraphicsLibraryInterfacePointer(pMyGraphicsLibraryInterface);

Your PlaneFactory subclass can also provide other public methods for accessing methods or data from the graphics library.

The DirectFBPlaneFactory in the source distribution contains a detailed example of a PlaneFactory subclass that provides public methods for initializing a graphics library.

Creating files for your platform-specific graphics driver

Put the header and source files for your platform-specific graphics driver in a subdirectory of the thirdparty-private/stagecraft-platforms directory. For information, see “Building platform-specific drivers and decoders” on page 79.

You can use the implementations provided by the source distribution without modification if they meet your needs. Otherwise, copy them to use as a starting point for your own implementation. For more information on the source distribution implementations, see “Implementations included with source distribution” on page 8.

Building your platform-specific graphics driver

For information about building your graphics driver, see “Building platform-specific drivers and decoders” on page 79.

Detailed tasks checklist

This checklist summarizes the steps for implementing a platform-specific Graphics Driver module.

1 Determine if you can use a Plane and PlaneFactory implementation provided in the source distribution. If you can, go to step 11.

2 Implement a class for your render plane that derives from the Plane class. For initial testing, use the I2DImpl subclass included with the source distribution for the I2D interface of the plane. Determine if you need a separate .h file for your Plane subclass, or even an abstract Plane subclass from which you derive your implementation.

3 Repeat step 2 for your output plane, if it has different requirements than your render plane. Typically, however, create one Plane subclass for your render plane, output plane, and bitmap caching render plane. Then, set characteristics of the Plane object at runtime depending on whether CreatePlane() or CreateOutputPlane() creates the Plane object. CreatePlane() creates the render plane and the bitmap caching render plane. CreateOutputPlane() creates the output plane.

4 Repeat step 2 for your bitmap caching render plane, if it has different requirements than your render plane.

5 For each Plane subclass you implemented, implement a class that derives from the PlaneFactory class. Determine if you need a separate .h for your subclass, or even an abstract subclass from which you derive your implementation.

6 Add a #define for the name of each Plane subclass you implemented. Put the #define into a header file to be included, at least, by your GraphicsDriverPlatform.cpp and by your Plane and PlaneFactory subclass implementation files.
7 In each platform-specific PlaneFactory.cpp file, register the PlaneFactory subclass with the Graphics Driver module.

8 Implement the GraphicsDriver methods GetDefaultOffscreenPlaneClass() and GetDefaultOutputPlaneClass() that return your default render plane and output plane subclass names.

9 If yours is a window-based platform, add user input handling to your output plane subclass.

10 Build your platform-specific graphics driver module. Follow the instructions in “Building platform-specific drivers and decoders” on page 79.

11 Test your Plane and PlaneFactory subclasses using the I2DImpl implementation.

12 Implement a class that derives from the I2D class. This I2D subclass interfaces with your hardware acceleration APIs and hardware.

13 Update the IGraphicsDriver.mk file to account for the new I2D subclass files you created. See “Building platform-specific drivers and decoders” on page 79.

14 Test using your platform-specific I2D subclass.
Chapter 3: The overlay video driver

What is overlay video?
Adobe® Flash® Lite® for the digital home supports overlay video. Overlay video means that dedicated hardware is responsible for the decoding and presentation of an audio/video stream. Hardware planes, sometimes called overlays, perform the output composition into a rectangular region specified by Flash Lite for the digital home. The overlay video decoding and presentation can be done by software, too, if the CPU can provide a high enough level of performance. However, hardware typically provides overlay video processing.

The role of Flash Lite for the digital home with overlay video
Flash Lite for the digital home provides the elementary audio and video streams to the hardware. Specifically, Flash Lite for the digital home does the following:

• Acquires the video. The video can also be progressively downloaded from file:// or http:// URLs. Alternatively, the video can be streamed from an Adobe® Flash® Media Server.
• Demultiplexes the audio/video stream into separate timestamped audio and video elementary streams.
• Sets the size and position of the rectangular region for the video on the display device.
• Feeds the elementary streams to a platform-specific decoder. The decoder is called a StreamPlayer in Flash Lite for the digital home.
• Passes control sequences to the StreamPlayer. Control sequences include play, pause, stop, and play from a new position.

Note: When video files are embedded in SWF content, the video is not decoded by a StreamPlayer. Rather, internal software video decoders decode embedded video.

The role of the StreamPlayer
A StreamPlayer is an audio/video decoder that provides the interface between Flash Lite for the digital home and your platform hardware. Flash Lite for the digital home provides abstract C++ classes that define the StreamPlayer interfaces. When you implement these interfaces, Flash Lite for the digital home is able to feed the elementary streams, the video size and position, and the control sequences to your StreamPlayer. Your StreamPlayer then interacts with the platform hardware.

Overlay video characteristics
The Flash Lite instance of Flash Lite for the digital home handles overlay video differently than video content that it decodes with its own internal software:

• The Flash Lite instance does not do the decoding and presentation of overlay video. The Flash Lite instance does not receive the decoded video pixels. Therefore, the Flash Lite instance performs no rendering operations on the decoded pixels.
• Because the Flash Lite instance does no rendering operations on overlay video, overlay video is displayed in a rectangular region only. Furthermore, flipping, skewing, rotation, and other transformation operations are not possible. Masking support for overlay video is limited to a single rectangular mask.
• Similarly, the Flash Lite instance does not receive the decoded audio PCM (pulse code modulation) stream. Therefore, the Flash Lite instance does not mix the decoded audio stream with other sounds that the Flash Lite instance generates.
• Overlay video is always presented underneath the Flash Lite frame buffer. Therefore, blending overlay video with graphics objects underneath the overlay video is not possible. However, Flash Lite can blend graphics objects on top of the overlay video.

**Class overview**

The overlay video driver includes these classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreamPlayer</td>
<td>Abstract class that defines the interfaces Flash Lite for the digital home uses to interact with hardware that decodes and presents audio/video streams.</td>
</tr>
<tr>
<td>StreamPlayerBase</td>
<td>Abstract class derived from StreamPlayer. The StreamPlayerBase class provides implementations of some of the pure virtual methods defined in StreamPlayer. You derive a platform-specific StreamPlayer from StreamPlayerBase, and implement the remaining StreamPlayer pure virtual methods.</td>
</tr>
<tr>
<td>IStreamPlayer</td>
<td>Abstract class which defines the interfaces for a StreamPlayer factory for creating and destroying StreamPlayer objects.</td>
</tr>
<tr>
<td>IStreamPlayerBase</td>
<td>Abstract class derived from IStreamPlayer. The IStreamPlayerBase class provides the implementations of some of the pure virtual methods defined in IStreamPlayer. You derive a platform-specific IStreamPlayer from IStreamPlayerBase, and implement the remaining IStreamPlayer pure virtual methods.</td>
</tr>
</tbody>
</table>

To implement a platform-dependent overlay video driver, you have two fundamental tasks:

• Implement a class that derives from the StreamPlayerBase class.
• Implement a class that derives from the factory class, IStreamPlayerBase.

**StreamPlayer and StreamPlayerBase classes**

The StreamPlayerBase abstract class derives from the StreamPlayer abstract class. Your implementation of the StreamPlayerBase class has these primary responsibilities:

• Provide the memory to which Flash Lite for the digital home copies the elementary audio and video stream data.
• Receive the elementary audio and video streams from Flash Lite for the digital home.
• Utilizing the platform hardware, decode and present the decoded audio and video streams to the user.
• Send Flash Lite for the digital home events that provide decoding status, such as buffer level status.
• Interact with the platform hardware to execute control sequences received from Flash Lite for the digital home.

StreamPlayerBase provides your StreamPlayer implementation some capabilities common to all implementations. For example, StreamPlayerBase provides a mechanism for sending events to Flash Lite for the digital home. It also provides a mechanism for tracking buffer levels.

**IStreamPlayer and IStreamPlayerBase classes**

The IStreamPlayer abstract class provides the interfaces for implementing the factory for creating and destroying instances of your StreamPlayerBase subclass. Your IStreamPlayer implementation is a module that Flash Lite for the digital home loads when a Flash Lite instance indicates video decoding is needed. The IStreamPlayerBase abstract class derives from the IStreamPlayer abstract class. You derive your implementation from IStreamPlayerBase.
The IStreamPlayerBase class, along with the StreamPlayerBase class, provides your IStreamPlayerBase implementation the ability to track buffer levels.

**Class interaction**

An application running on your platform is a host application. The host application interacts with Flash Lite for the digital home to run Flash Lite applications. Specifically, the host application interacts with the IStagecraft module. Using this interface, the host application creates a StageWindow instance. The StageWindow instance contains an instance of Adobe® Flash® Lite® 3.1. Flash Lite loads the SWF file specified by the host application.

When the SWF content wants to play a video, the Flash Lite instance creates an instance of your IStreamPlayerBase subclass, and uses it to create an instance of your StreamPlayerBase subclass. The Flash Lite instance then passes the size and position of the video rectangle, the elementary audio and video streams, and the video control sequences to your StreamPlayerBase subclass instance.

When the Flash Lite instance no longer needs the video because, for example, the video has finished playing, the Flash Lite instance uses your IStreamPlayerBase subclass instance to destroy the StreamPlayerBase subclass instance.

**File locations**

<table>
<thead>
<tr>
<th>Class</th>
<th>Header file</th>
<th>Implementation file</th>
</tr>
</thead>
<tbody>
<tr>
<td>IStreamPlayer</td>
<td>include/ae/ddk/streamplayer/IStreamPlayer.h</td>
<td>Not applicable</td>
</tr>
<tr>
<td>IStreamPlayerBase</td>
<td>source/ae/ddk/streamplayer/IStreamPlayerBase.h</td>
<td>source/ae/ddk/streamplayer/IStreamPlayerBase.cpp</td>
</tr>
<tr>
<td>StreamPlayer</td>
<td>include/ae/ddk/streamplayer/StreamPlayer.h</td>
<td>Not applicable</td>
</tr>
<tr>
<td>StreamPlayerBase</td>
<td>source/ae/ddk/streamplayer/StreamPlayerBase.h</td>
<td>source/ae/ddk/streamplayer/StreamPlayerBase.cpp</td>
</tr>
</tbody>
</table>
Audio and video codecs

A Flash Lite instance requests your IStreamPlayerBase instance to create an instance of your StreamPlayerBase subclass. In the request, the Flash Lite instance passes the audio and video codecs required by the video. While any combination of the supported audio and video codecs is possible, the most common combinations are as follows:

<table>
<thead>
<tr>
<th>Video codec</th>
<th>Audio codec</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.264</td>
<td>AAC</td>
</tr>
<tr>
<td>H.264</td>
<td>MP3</td>
</tr>
<tr>
<td>Sorenson H.263</td>
<td>MP3</td>
</tr>
<tr>
<td>On2 VP6</td>
<td>MP3</td>
</tr>
<tr>
<td>No video stream</td>
<td>AAC or MP3</td>
</tr>
<tr>
<td>H.264, Sorenson H.263, or On2 VP6</td>
<td>No audio stream</td>
</tr>
</tbody>
</table>

Your IStreamPlayerBase instance determines if it can create a StreamPlayer object that supports the requested codec combination.

Falling back to software decoding

Your IStreamPlayerBase implementation determines if it can create a StreamPlayer object. One reason it can fail is that your StreamPlayerBase implementation cannot handle the requested codecs. Another reason is that your StreamPlayerBase implementation cannot handle another concurrent decoder.

Whatever the reason, Flash Lite for the digital home handles the failure to create a StreamPlayer by falling back to the appropriate internal software video and audio decoders. Flash Lite for the digital home provides software video decoders for Sorenson H.263 and On2 VP6 codecs. It provides software audio decoders for the mp3, PCM, and ADPCM codecs.

No fallback video decoder is available for the H.264 codec, and no software audio decoder is available for the AAC codec. Therefore, if the requested video codec is H.264, and your IStreamPlayerBase instance determines it cannot create a StreamPlayer object for H.264, Flash Lite for the digital home fails to play the video. However, consider the case where the requested audio codec is AAC, and the requested video codec is either Sorenson H.263 or On2 VP6, and your IStreamPlayerBase instance determines it cannot create a StreamPlayer object. Then Flash Lite for the digital home uses the software video decoder, but no audio decoder. In this case, the video is played but without sound.

When Flash Lite for the digital home falls back to a software video decoder, the software video decoder decodes each frame of video and passes the decoded frame back to the Flash Lite instance for rendering. Therefore, falling back to software video decoding can require much CPU processing and impact system performance.

Implementations included with source distribution

The source distribution for Flash Lite for the digital home includes two StreamPlayer implementations.

Updated 13 May 2009
StreamPlayer class details

StreamPlayer class definition
The abstract StreamPlayerBase class derives from the abstract StreamPlayer class. StreamPlayerBase provides implementations of some of the StreamPlayer methods. You implement a subclass of StreamPlayerBase and provide the remaining method implementations. If a method provided by StreamPlayerBase does not meet your needs, provide its implementation in your StreamPlayerBase subclass also.
StreamPlayer class hierarchy diagram

**Video display**

Flash Lite for the digital home passes your platform-specific StreamPlayer object a pointer to an output plane by calling its `SetOutputPlane()` method. The output plane maps to the display device where the StreamPlayer displays the video. For more information on output planes, see “Plane class” on page 5.

Flash Lite for the digital home also passes the size and position of the video region to the StreamPlayer object by calling `SetVideoRegion()`. The region is rectangular. The StreamPlayer decodes and displays the video in the specified region within the output plane. The StreamPlayer presents the video as fully opaque. If the SWF content repositions or resizes the video region, Flash Lite for the digital home calls `SetVideoRegion()` again.

The Flash Lite instance has no part in decoding and rendering the video. However, the Flash Lite instance can render other objects on top of the rectangular region. These objects include text, bitmaps, and vector graphics. The Flash Lite instance uses the alpha (transparency) value for the objects to appropriately blend the objects on top of the video.
Threading

Flash Lite for the digital home calls your platform-specific StreamPlayer object’s methods `SendESAudio()` and `SendESVideo()` to send each payload of elementary stream data to the StreamPlayer. Calls that Flash Lite for the digital home makes to `SendESVideo()` are in a separate thread from calls to `SendESAudio()`.

Furthermore, the StreamPlayer object can choose to create additional, separate threads for decoding each of the audio and video elementary streams. Typically, the StreamPlayer does create separate threads. Therefore, calls to `SendESAudio()` and `SendESVideo()` are asynchronous. The method returns, but a separate thread processes the elementary stream. The FFmpegStreamPlayer implementation provided with the source distribution has an example of separate decoding threads.

Payload format of elementary streams

The payload format of the elementary streams that Flash Lite for the digital home passes to a StreamPlayer object depends on the codec.

H.264

The first H.264 packets that Flash Lite for the digital home passes to the StreamPlayer object is a sequence parameter set (SPS) and a picture parameter set (PPS). The SPS and PPS syntax is defined in ISO 14496-10 section 7.3.1.2.

Subsequent data is sent as NAL units. The NAL units begin with a NAL unit start code prefix (defined in ISO 14496-10 3.130) followed by the NAL Unit payload. The NAL Unit syntax is defined in ISO 14496-10, section 7.3.1 and Annexure B.1.

The SPS and PPS data is sent again if a discontinuity in the stream occurs. A discontinuity can occur after a seek or dynamic bitrate adjustment by Flash lite for the digital home.

Sorenson H.263

Flash Lite for the digital home provides the Sorenson H.263 data as follows:

1 32 bits that contain the size in bytes of the payload data. This size does not include the 4 bytes that contain the size. For example, if the `bufferSize` parameter of `SendVideoES()` equals 1024, then the first 4 bytes contain the value 1020, and the remaining 1020 bytes contain the payload data.

2 The Sorenson H.263 data format, which is a proprietary format that is only available to Sorenson codec licensees.

On2 VP6

Flash Lite for the digital home provides the On2 VP6 data as follows:

1 32 bits that contain the size in bytes of the payload data. This size does not include the 4 bytes that contain the size. For example, if the `bufferSize` parameter of `SendVideoES()` equals 1024, then the first 4 bytes contain the value 1020, and the remaining 1020 bytes contain the payload data.

2 The On2 VP6 data format, which is a proprietary format that is only available to On2 VP6 codec licensees.

AAC

Flash Lite for the digital home provides the AAC data as follows:

1 An Audio Data Transport Stream (ADTS) packet as defined in ISO 14496-3 part 1 Annex 1.a and ISO 13818-7.

2 The ADTS packet is appended with the audio payloads as defined in ISO 14496-3, section 1.6.2.2.
Mp3
Flash Lite for the digital home provides mp3 data as a standard mp3 data audio stream. This standard is defined in the MPEG-1 Layer 3 specifications ISO/IEC 11172-3 and 13818-3.

Buffer management

Buffer allocation
Flash Lite for the digital home calls the StreamPlayer object's methods SendESAudio() and SendESVideo() to send each payload of elementary stream data to the StreamPlayer. Before calling one of these methods, Flash Lite for the digital home copies the payload data into a buffer. One of the parameters passed to SendESAudio() and SendESVideo() is a pointer to the buffer.

Your platform-dependent StreamPlayer object manages these buffers. Before each call to SendESAudio() and SendESVideo(), Flash Lite for the digital home calls the StreamPlayer object's GetBuffer() method to get a pointer to a buffer's memory. Therefore, the StreamPlayer object determines the memory requirements. For example, the StreamPlayer object sometimes keeps the buffers in a specific memory region which provides direct memory access to the hardware. If the StreamPlayer object has no special memory requirements, use AE_MALLOC() to allocate the memory. In this case, you can use the StreamPlayerBase implementation of GetBuffer().

Because SendESAudio() and SendESVideo() typically decode the elementary streams asynchronously, when these methods return, Flash Lite for the digital home does not immediately call the StreamPlayer object's ReleaseBuffer(). The StreamPlayer object sends an event to Flash Lite for the digital home when it no longer needs the buffer. When Flash Lite for the digital home receives the event, it calls the ReleaseBuffer() method to release the buffer's memory.

Buffer levels
The StreamPlayer object determines the high and low watermarks of its buffers. The high watermark is when the StreamPlayer determines that further calls to SendESVideo() and SendESAudio() could overflow its buffers. The low watermark is when the StreamPlayer object determines that it soon will have no further data to decode. The StreamPlayer object sends events to Flash Lite for the digital home whenever its buffer levels reach the high or low watermark.

Flash Lite for the digital home calls the StreamPlayer object's SetBufferLevels() method to set the high and low watermarks for the audio stream and the video stream. However, the StreamPlayer implementation considers these levels only as hints. The StreamPlayer implementation can use its own algorithm for determining the high and low watermarks.

The StreamPlayerBase and IStreamPlayerBase classes provide buffer tracking tools. In development builds, but not in release builds, these tracking tools display current buffer levels. For more information, see “Buffer level tracking tools” on page 48.

Prerolling buffer levels
The StreamPlayer object determines the size of the preroll buffer. The preroll buffer size is the number of audio and video bytes that the StreamPlayer caches before starting to decode. Flash Lite for the digital home calls the StreamPlayer object's SetPrerollSize() method to set the preroll size for the audio stream and the video stream. However, the StreamPlayer implementation considers these sizes only as hints. The StreamPlayer implementation can use its own algorithm for determining the preroll size. The StreamPlayer implementation can also use the preroll size in its algorithm for determining the high and low watermarks.
Events

A StreamPlayer object sends asynchronous events to Flash Lite for the digital home. The StreamPlayerBase class provides an implementation for sending events.

The types of events are defined in the EventType enumeration in StreamPlayer.h. Flash Lite for the digital home registers and unregisters to receive events by calling the StreamPlayer object's AddNotifier() and RemoveNotifier() methods.

A StreamPlayer object sends events as an Event structure in the Notifier class, also defined in StreamPlayer.h. When an event occurs, the StreamPlayer object fills in the appropriate values of an Event structure. Then the StreamPlayer object calls SendNotification(), implemented in StreamPlayerBase, to send the event.

The StreamPlayer object always assigns values to these members of the Event structure:

- **m_eventType**  The type of event. See the table below.
- **m_timestampOfEvent** The time the event occurred.
- **m_streamerType** Whether the event involves an audio stream or a video stream.

The following table lists the types of events, their descriptions, and the required and optional Event structure members for the events.

<table>
<thead>
<tr>
<th>EventType</th>
<th>Event members</th>
<th>Description</th>
</tr>
</thead>
</table>
| kBufferFull | Required: m_bufferLevel[streamType]  
streamType is kVideoStream or kAudioStream, depending on which type of stream’s buffers have reached the high watermark.  
Optional:  
m_bufferLevel[streamType]  
streamType is the other stream type. | This event is required.  
Send this event to notify Flash Lite for the digital home to stop sending data. Send this event each time the buffer level for the audio or video stream buffer reaches the high watermark. |
| kBufferLow | Required: m_bufferLevel[streamType]  
streamType is kVideoStream or kAudioStream, depending on which type of stream’s buffers have reached the low watermark.  
Optional:  
m_bufferLevel[streamType]  
streamType is the other stream type. | This event is required.  
Send this event to notify Flash Lite for the digital home to resume sending data. Send this event each time the buffer level for the audio or video stream buffer reaches the low watermark. |
| kBufferEmpty | Required: m_bufferLevel[streamType]  
streamType is kVideoStream or kAudioStream, depending on which type of stream’s buffers are empty.  
Optional:  
m_bufferLevel[streamType]  
streamType is the other stream type. | This event is required.  
Send this event to notify Flash Lite for the digital home to resume sending data because the buffers are empty.  
This error condition exists because the StreamPlayer object always sends a kBufferLow event when the low watermark is reached. If Flash Lite for the digital home does not send more data in response to kBufferLow, and it has not called NotifyEOF() to indicate the end of the stream, this error condition occurs. |
The following diagram gives an example of a call sequence showing the flow of events.

<table>
<thead>
<tr>
<th>EventType</th>
<th>Event members</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kReachedEOS</td>
<td>Required:</td>
<td>This event is required.</td>
</tr>
<tr>
<td></td>
<td>m_bufferLevel[streamType]</td>
<td>Send this event when the last sample in the stream has been decoded and presented. The StreamPlayer object sends this event only if Flash Lite for the digital home had previously called the NotifyEOF() method. Otherwise, when the last sample has been decoded, send kBufferEmpty to indicate an error has occurred.</td>
</tr>
<tr>
<td></td>
<td>Optional:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>m_bufferLevel[streamType]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>streamType is kVideoStream or kAudioStream, depending on which type of stream reached the end.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Optional:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>m_bufferLevel[streamType]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>streamType is the other stream type.</td>
<td></td>
</tr>
<tr>
<td>kReleaseBuffer</td>
<td>Required:</td>
<td>This event is required.</td>
</tr>
<tr>
<td></td>
<td>m_pBuffer</td>
<td>Send this event to release the buffer that was passed to SendAudioES() or SendVideoES(). If you do not send this event, the buffer is leaked. Send this event even if the StreamPlayer does not use the data.</td>
</tr>
<tr>
<td>kVDimAvailable</td>
<td>Required:</td>
<td>This event is required.</td>
</tr>
<tr>
<td></td>
<td>m_videoWidth and m_videoHeight</td>
<td>Send this event to indicate the decoded video dimensions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>This event occurs when the video dimensions in the video elementary stream itself change mid-stream. Because Flash Lite for the digital home does not parse the video elementary stream, the StreamPlayer sends this event.</td>
<td></td>
</tr>
<tr>
<td>kStateChanged</td>
<td>Required:</td>
<td>This event is required.</td>
</tr>
<tr>
<td></td>
<td>m_state</td>
<td>Send this event to indicate that the StreamPlayer object’s state has changed. The state enumeration is defined in StreamPlayer.h.</td>
</tr>
<tr>
<td>kPrerolling</td>
<td>Optional:</td>
<td>This event is optional.</td>
</tr>
<tr>
<td></td>
<td>m_bufferLevel[streamType]</td>
<td>Send this event while prerolling, before starting to decode and display the stream. In development builds, this event allows the buffer tracking tools to show the buffer levels changing before decoding starts. The buffer tracking tools, provided by StreamPlayerBase and IStreamPlayerBase, display the buffer levels.</td>
</tr>
<tr>
<td></td>
<td>streamType is kVideoStream or kAudioStream, depending on which type of stream’s buffers the prerolling event applies to.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Optional:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>m_bufferLevel[streamType]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>streamType is the other stream type.</td>
<td></td>
</tr>
<tr>
<td>kFrameUpdate</td>
<td>Optional:</td>
<td>This event is optional.</td>
</tr>
<tr>
<td></td>
<td>m_bufferLevel[streamType]</td>
<td>Send this event when the StreamPlayer has decoded another frame. In development builds, this event allows the buffer tracking tools to display the buffer levels. The buffer tracking tools, provided by StreamPlayerBase and IStreamPlayerBase, display the buffer levels.</td>
</tr>
<tr>
<td></td>
<td>streamType is kVideoStream or kAudioStream, depending on the type of stream the event is reporting.</td>
<td></td>
</tr>
<tr>
<td>kError</td>
<td>This event is optional.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Send this event to indicate an error in processing the stream. Flash Lite for the digital home does not use this event. However, the event can be used by a test program that drives your StreamPlayer.</td>
<td></td>
</tr>
</tbody>
</table>
Control sequences

Flash Lite for the digital home supports these control sequences: play at a normal speed, stop, pause, and play from a new position. Flash Lite for the digital home does not support slow motion, fast forward, or rewind. Your platform-specific StreamPlayer implements methods to handle the supported control sequences.

To play a video, Flash Lite for the digital home calls the `Play()` method of the StreamPlayer object. To pause, the `Pause()` method is called. To resume playing, the `Play()` method is called again.

To stop playing the video, Flash Lite for the digital home calls `Stop()` followed by `Flush()`. `Flush()` empties the buffers. Flash Lite for the digital home then calls `Play()` again, and calls `SendESVideo()` and `SendESAUDIO()` to reload data in the buffers.

Therefore, to seek and then play from a new position, Flash Lite for the digital home calls these methods:

1. `Stop()`
2. `Flush()`
3. `Play(decodeToTime, pauseAtDecodeTime)`
4. `SendVideoES()`
5. `SendAudioES()`

This overloaded `Play()` method's first parameter, `decodeToTime`, specifies the timestamp that playback is to resume from. However, Flash Lite for the digital home starts sending payload data starting with the closest previous video keyframe to the specified timestamp. The StreamPlayer object is responsible for decoding, but not displaying, the data up to the specified timestamp. Once the data reaches the specified timestamp, the StreamPlayer object resumes playing if `pauseAtDecodeTime` is false. If `pauseAtDecodeTime` is true, the StreamPlayer object pauses the video on the frame that matches the `decodeToTime` value.
**StreamPlayer methods**

For detailed definitions of return values and parameters of the StreamPlayer and StreamPlayerBase class methods, see include/ae/ddk/streamplayer/StreamPlayer.h and source/ae/ddk/streamplayer/StreamPlayerBase.h.

**AddNotifier() method**

The StreamPlayerBase class provides an implementation for this method. This method adds a Notifier object to your StreamPlayer object's list of Notifier objects. When an event occurs, such as kBufferFull or kBufferLow, a StreamPlayer object notifies all its Notifier objects. To notify the objects, the StreamPlayer object calls the OnEvent() method of each Notifier object. The Notifier class is defined in include/ae/ddk/streamplayer/StreamPlayer.h.

The StreamPlayerBase class provides an implementation of the StreamPlayer methods AddNotifier() and RemoveNotifier(). It also provides an implementation of the Notifier method OnEvent(). Finally, it provides a method called SendNotification(). Call SendNotification() whenever your StreamPlayer object has an event to send to Flash Lite for the digital home.

For more information, see “Events” on page 39.

**Flush() method**

Flash Lite for the digital home calls this method to empty the buffers of the audio and video streams. Flash Lite for the digital home calls the Flush() method after a call to Stop() if it wants to resume playback at a different presentation timestamp.

For more information, see “Control sequences” on page 41.

*Note: The StreamPlayerBase class does not provide an implementation of Flush(). Provide the implementation in the class you derive from StreamPlayerBase.*

**GetAudioVideoType() method**

The StreamPlayerBase class provides an implementation for this method. This method returns the audio type and the video type that a StreamPlayer object is playing.

**GetBuffer() method**

The StreamPlayerBase class provides a simple implementation for this method. This method returns a pointer to a memory block that Flash Lite for the digital home uses to provide payload data to the StreamPlayer.

The StreamPlayerBase implementation of GetBuffer() allocates memory using AE_MALLOC(). If your StreamPlayer implementation requires more specialized memory allocation, add an implementation of the GetBuffer() method to your platform StreamPlayer class that derives from StreamPlayerBase. For more information, see “Buffer management” on page 38.
**GetBufferLevels() method**

This method returns the current levels of the audio and video buffers that your StreamPlayer object manages. This method also returns the high and low watermarks for the buffers. Depending on your StreamPlayer implementation, the high and low watermarks provided by `GetBufferLevels()` do not have to be the same as the watermarks passed in a previous call to `SetBufferLevels()`. The reason for the difference is that your StreamPlayer implementation determines the watermarks based on its own requirements, which do not have to involve the values passed in `SetBufferLevels()`. For more information, see “Buffer management” on page 38.

*Note: The StreamPlayerBase class does not provide an implementation of GetBufferLevels(). Provide the implementation in the class you derive from StreamPlayerBase.*

**GetCurrentPTS() method**

This method returns the presentation timestamp of the video frame currently being displayed. For an audio-only stream, this method returns the presentation timestamp of the last audio packet played.

*Note: The StreamPlayerBase class does not provide an implementation of GetCurrentPTS(). Provide the implementation in the class you derive from StreamPlayerBase.*

**GetOutputPlane() method**

The StreamPlayerBase class provides an implementation for this method. This method returns a pointer to the Plane object, which is the output plane for the video display. For more information, see “SetOutputPlane() method” on page 45.

**GetStreamPlayerModule() method**

The StreamPlayerBase class provides an implementation for this method. This method returns a pointer to the IStreamPlayerBase object that created the StreamPlayer object. The method is used by the buffer level tracking tools that StreamPlayerBase and IStreamPlayerBase provide.

**GetVideoRegion() method**

The StreamPlayerBase class provides an implementation for this method. This method returns the current size and position of the video within the output plane.

For more information, see “SetVideoRegion() method” on page 45.

**NotifyEOF() method**

Flash Lite for the digital home calls this method to notify the StreamPlayer object that it is done playing the video. After calling `NotifyEOF()`, Flash Lite for the digital home makes no further calls to `GetBuffer()`, or to `SendVideoES()` and `SendAudioES()`. After receiving a call to `NotifyEOF()`, the StreamPlayer object sends the event `kReachedEOS` after decoding and presenting the last payload data in its buffers. For more information, see “Events” on page 39.

*Note: The StreamPlayerBase class does not provide an implementation of NotifyEOF(). Provide the implementation in the class you derive from StreamPlayerBase.*

**Pause() method**

Flash Lite for the digital home calls this method to pause decoding and displaying of the decompressed audio and video streams. Flash Lite for the digital home calls the `Play()` method to resume decoding and displaying.
For more information, see “Control sequences” on page 41.

Note: The StreamPlayerBase class does not provide an implementation of Pause(). Provide the implementation in the class you derive from StreamPlayerBase.

**Play() method**

Flash Lite for the digital home calls this method to start decoding and displaying the decompressed audio and video streams. Flash Lite for the digital home uses an overloaded Play() method to perform a seek operation. The overloaded Play() takes these parameters:

- `decodeToTime`. This parameter indicates to the StreamPlayer object to not display video until the presentation timestamp is greater than or equal to `decodeToTime`.
- `pauseAtDecodeTime`. A Boolean value. If true, then the StreamPlayer object pauses playback at the time specified by `decodeToTime`. If false, the StreamPlayer object resumes playback at the specified time.

For more information, see “Control sequences” on page 41.

Note: The StreamPlayerBase class does not provide an implementation of Play(). Provide the implementation in the class you derive from StreamPlayerBase.

**ReleaseBuffer() method**

The StreamPlayerBase class provides a simple implementation for this method. This method deallocates the memory previously allocated by a call to GetBuffer(). The StreamPlayerBase implementation of ReleaseBuffer() deallocates the memory using AE_FREE(). If your StreamPlayer implementation requires more specialized memory allocation and deallocation, add an implementation of the ReleaseBuffer() method to your platform StreamPlayer class that derives from StreamPlayerBase. For more information, see “Buffer management” on page 38.

**RemoveNotifier() method**

The StreamPlayerBase class provides an implementation for this method. This method removes a Notifier object from your StreamPlayer object’s list of Notifier objects. For more information, see “AddNotifier() method” on page 42 and “Events” on page 39.

**SendNotification() method**

The StreamPlayerBase class provides an implementation for this method. This method sends an event to each Notifier object in your StreamPlayer object’s list of Notifier objects. Call this method in your StreamPlayer implementation whenever you have an event to send to Flash Lite for the digital home. For more information, see “AddNotifier() method” on page 42 and “Events” on page 39.

**SendAudioES() method**

Flash Lite for the digital home calls this method to send a packet of the audio elementary stream payload to the StreamPlayer object. The parameters are as follows:

- A pointer to the buffer containing the compressed data. Flash Lite for the digital home calls GetBuffer() to retrieve the buffer pointer before calling SendAudioES().
- The number of bytes of data in the buffer.
- The presentation timestamp of the decompressed audio frame. Use the presentation timestamp to determine when to display the data.
**SendVideoES() method**

Flash Lite for the digital home calls this method to send a packet of the video elementary stream payload to the StreamPlayer object. The parameters are as follows:

- A pointer to the buffer containing the compressed data. Flash Lite for the digital home calls `GetBuffer()` to retrieve the buffer pointer before calling `SendVideoES()`.
- The number of bytes of data in the buffer.
- The presentation timestamp of the decompressed video frame. Use the presentation timestamp to determine when to display the data.
- The video frame encoding type. The `VideoFrameType` enumeration in `StreamPlayer.h` defines values for the `I-frame`, `P-frame`, and `B-frame` types.

*Note:* The `StreamPlayerBase` class does not provide an implementation of `SendVideoES()`. Provide the implementation in the class you derive from `StreamPlayerBase`.

**SetBufferLevels() method**

Flash Lite for the digital home calls this method to suggest values for the high and low watermarks for the audio and video buffers that your StreamPlayer object manages. For more information, see “Buffer management” on page 38.

*Note:* The `StreamPlayerBase` class does not provide an implementation of `SetBufferLevels()`. Provide the implementation in the class you derive from `StreamPlayerBase`.

**SetOutputPlane() method**

The `StreamPlayerBase` class provides an implementation for this method. Flash Lite for the digital home calls this method to provide the StreamPlayer object a pointer to an output plane. The output plane maps to the display device where the StreamPlayer displays the video. For more information on output planes, see “Plane class” on page 5.

**SetPrerollSize() method**

The `StreamPlayerBase` class provides an implementation for this method. Flash Lite for the digital home calls this method to provide the StreamPlayer the number of bytes to preroll before presenting the video. For more information, see “Buffer management” on page 38.

**SetVideoRegion() method**

The `StreamPlayerBase` class provides an implementation for this method. Flash Lite for the digital home calls this method to resize and reposition the video within the output plane.

`SetVideoRegion()` takes two parameters: `videoRect` and `planeRect`. The parameter `videoRect` specifies the rectangle or subrectangle of the video image to display. The parameter `planeRect` specifies the rectangle or subrectangle of the output plane in which to display the video.

The parameters are both `Rect` structures. The `Rect` structure is defined in `include/ae/stagecraft/StagecraftTypes.h`. The `x` and `y` members of the `videoRect` structure are relative to the video image’s upper leftmost corner. The `x` and `y` members of the `planeRect` structure are relative to the output plane’s upper leftmost corner. The upper leftmost corner has the `x, y` coordinates `0, 0`. Updated 13 May 2009
Stop() method
Flash Lite for the digital home calls this method to stop the playback of the audio and video streams.

For more information, see “Control sequences” on page 41.

*Note:* The StreamPlayerBase class does not provide an implementation of Stop(). Provide the implementation in the class you derive from StreamPlayerBase.

**IStreamPlayer class details**

**IStreamPlayer class definition**
The abstract IStreamPlayerBase class derives from the abstract IStreamPlayer class. IStreamPlayerBase provides implementations of some of the IStreamPlayer methods. You implement a subclass of IStreamPlayerBase, and provide the remaining method implementations. The main purpose of your IStreamPlayerBase object is to create and destroy your StreamPlayerBase object. The IStreamPlayerBase object also provides methods used for tracking buffer levels in a development build.

![IStreamPlayer class hierarchy diagram](image-url)
**IStreamPlayer class methods**

**CreateStreamPlayer() method**

This method creates a StreamPlayer object. When SWF content wants to play a video, the Flash Lite instance creates an instance of your IStreamPlayerBase subclass, and calls `CreateStreamPlayer()` to create an instance of your StreamPlayerBase subclass. The parameters passed to `CreateStreamPlayer()` are the audio and video codecs to be decoded. Return null if your StreamPlayerBase implementation cannot decode the specified codecs. Also return null if your IStreamPlayerBase implementation cannot create your StreamPlayerBase object for any reason.

For more information about the ramifications of returning null, see “Falling back to software decoding” on page 34.

**DestroyStreamPlayer() method**

This method destroys a StreamPlayer object. A pointer parameter specifies the StreamPlayer object to destroy. When the Flash Lite instance no longer needs the video because, for example, the video has finished playing, the Flash Lite instance calls `DestroyStreamPlayer()`.

**IsShowingBufferLevels() method**

The IStreamPlayerBase class provides an implementation of this method. This method returns true if the IStreamPlayerBase object is showing the buffer level tracking data. This tracking data is available only in development builds.

**ShowBufferLevels() method**

The IStreamPlayerBase class provides an implementation of this method. This method takes one Boolean parameter. If the parameter is true, the IStreamPlayerBase object shows the buffer level tracking data. This tracking data is available only in development builds.

**Creating files for your platform-specific graphics driver**

Put the header and source files for your platform-specific graphics driver in a subdirectory of the thirdparty-private/stagecraft-platforms directory. For information, see “Building platform-specific drivers and decoders” on page 79.

You can use the implementations provided by the source distribution without modification if they meet your needs. Otherwise, copy the files to use as a starting point for your own implementation. For more information on the source distribution implementations, see “Implementations included with source distribution” on page 34.

**Building your platform-specific overlay video driver**

For information about building your overlay video driver, see “Building platform-specific drivers and decoders” on page 79.
Buffer level tracking tools

The IStreamPlayerBase and StreamPlayerBase classes provide tools for tracking buffer levels. These tools are for Linux development builds only. By deriving your IStreamPlayer and StreamPlayer classes from IStreamPlayerBase and StreamPlayerBase, you can use these tools. The tools provide a graphical display of your audio and video stream buffer levels. A description of the graphical display is in include/ae/ddk/streamplayer/IStreamPlayer.h.

When Flash Lite for the digital home acquires the IStreamPlayer module, it adds a StreamPlayer related shell command to a set of shell commands. These shell commands are useful for testing Flash Lite for the digital home. The StreamPlayer related shell command is the following:

```
streambuffers [on|off]
```

Use this shell command to turn on and off the graphical display of buffer levels.
Chapter 4: The sound driver

Adobe® Flash® Lite® for the digital home plays SWF content. This content often outputs PCM sound samples for audio playback. To direct Flash Lite for the digital home to use the audio output hardware and APIs of your target platform, you implement the sound driver interfaces. The interfaces are abstract C++ classes. You also configure your platform’s PCM buffers and sample rate.

The source distribution provides a sound driver implementation for Linux® systems that use Advanced Linux Sound Architecture (ALSA). If your platform uses ALSA, your only task is to customize this ALSA sound driver. If your platform does not use ALSA, you have to implement the sound driver interfaces yourself.

Class and structure overview

The sound driver includes these classes and structures:

<table>
<thead>
<tr>
<th>Class or structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoundOutput</td>
<td>Abstract class that you implement. This class defines the interfaces that Flash Lite for the digital home uses to send its PCM samples to the audio output hardware. Flash Lite for the digital home also uses these interfaces to control and monitor the audio output hardware operations.</td>
</tr>
<tr>
<td>IFL31NativeSoundOutput</td>
<td>Abstract class, derived from IAEModule, that you implement to provide a singleton sound driver module. Flash Lite for the digital home uses this singleton object as a factory to create and destroy platform-specific SoundOutput objects. Flash Lite for the digital home also uses the singleton object to configure and manage the SoundOutput objects. The &quot;FL31&quot; in the class name indicates that the class is designed specifically for how Adobe® Flash® Lite® 3.1 handles sound.</td>
</tr>
<tr>
<td>SoundOutputBase</td>
<td>Abstract class that derives from SoundOutput. When you implement a platform-specific SoundOutput class, derive it from SoundOutputBase. SoundOutputBase defines a public destructor.</td>
</tr>
<tr>
<td>AudioParameters</td>
<td>Structure in which you configure your platform’s PCM buffers and sample rate.</td>
</tr>
</tbody>
</table>

The SoundOutput and IFL31NativeSoundOutput classes are defined in `include/ae/ddk/fl31nativesoundoutput/IFL31NativeSoundOutput.h`.

The SoundOutputBase class and the AudioParameters structure are defined in the directory `source/ae/ddk/fl31nativesoundoutput` in files `SoundOutputBase.h`, `AudioParameters.h`, and `AudioParameters.cpp`.

Class interaction

The host application is defined in "Running Flash Lite for the digital home” on page 3. The host application interacts with Flash Lite for the digital home to run Flash Lite applications. Specifically, the host application interacts with the IStagecraft module. Using this interface, the host application creates a StageWindow instance. The StageWindow instance contains an instance of Adobe® Flash® Lite® 3.1. The Flash Lite instance loads the SWF file specified by the host application.
The sound driver

The following diagram and sections summarize the interactions between StageWindow instances, their Flash Lite instances, the IFL3NativeSoundOutput singleton, the SoundOutput instances, and the hardware.

**Sound driver architecture diagram**

In the diagram, the asterisks and numbers indicate whether the association between components is one-to-many or one-to-one.

**Acquiring the sound driver module**

When a StageWindow instance initializes, it acquires the sound driver module -- the singleton instance of your platform-specific IFL3NativeSoundOutput class. The StageWindow instance queries the IFL3NativeSoundOutput singleton for audio configuration parameters, such as buffer sizes, and configures the Flash Lite instance accordingly.

**Creating a SoundOutput instance**

When the SWF content has sound output to play, the StageWindow instance asks the IFL3NativeSoundOutput singleton to create an instance of your platform-specific SoundOutput class. Each StageWindow instance has at most one SoundOutput instance. The StageWindow instance asks the IFL3NativeSoundOutput singleton to destroy the SoundOutput instance when the SWF content no longer has sound output to play. For example, when the SWF movie ends, the IFL3NativeSoundOutput singleton destroys the SoundOutput instance.

**Flash Lite instance and SoundOutput instance interaction**

The Flash Lite instance interacts with the SoundOutput instance to do the following:

- Initialize so it is ready to accept PCM sample buffers from the Flash Lite instance for immediate transfer to the audio output hardware.
- Send PCM samples that SWF content generates to the audio output hardware. The Flash Lite instance manages the buffers which contain the PCM samples.
- Pause and resume playing audio.
- Query how many buffers have been played.
- Shut down, releasing its resources and its connection to the audio output hardware.
**PCM sample format**

The Flash Lite instance sends PCM samples to the SoundOutput instance. The PCM samples are 16-bit little endian, mono or stereo, and left-right sample interleaved in the case of stereo.

**Enabling sound focus**

The IFL3NativeSoundOutput singleton also interacts with the SoundOutput instances. The IFL3NativeSoundOutput singleton specifies which SoundOutput instance has control of the hardware. Although each StageWindow instance has only one SoundOutput instance, multiple StageWindow instances can co-exist. However, Flash Lite for the digital home assumes that the platform has only one audio output device.

Therefore, only one SoundOutput instance at a time sends its PCM samples to the audio hardware. Flash Lite for the digital home tells the IFL3NativeSoundOutput singleton which StageWindow instance has the focus. The IFL3NativeSoundOutput singleton then directs the appropriate SoundOutput instance to send its PCM samples to the audio output hardware. The IFL3NativeSoundOutput singleton directs the other SoundOutput instances to stop sending their PCM samples to the hardware.

Each Flash Lite instance does not know whether its SoundOutput instance has the sound focus. The behavior of the Flash Lite instance remains the same when its SoundOutput instance has the sound focus or does not have the sound focus. That is, the Flash Lite instance continues its interactions with the SoundOutput instance, such as sending it PCM samples that SWF content generates. The SoundOutput instance, therefore, must not change its behavior with respect to its interactions with the Flash Lite instance, regardless whether it has the focus.

For more information, see “EnableOutput() method” on page 55 and “SetFocus() method” on page 57.

**Mixing audio inputs**

Flash Lite for the digital home supports two inputs to your platform’s mixer interface. One input is the PCM samples from the SoundOutput instance that has the sound focus. The other input is from the audio stream of a video that the Flash Lite instance of some StageWindow instance is playing. Flash Lite for the digital home passes the compressed audio stream to a StreamPlayer object. The StreamPlayer object does hardware accelerated audio decoding. The resulting decoded stream is input to your platform’s mixer interface. For more information, see “The overlay video driver” on page 31.

Using the IFL3NativeSoundOutput singleton, Flash Lite for the digital home can set output volume levels of the two inputs. Flash Lite for the digital home has default level settings, but SWF content can change the levels. Also, the host application can change the levels.

**Setting audio parameters**

When SWF content has audio output, the Flash Lite instance buffers the PCM samples. The Flash Lite instance passes a pointer to a filled buffer to a SoundOutput object. The number of buffers and the size of each buffer is platform-dependent. You specify these values in an AudioParameters structure. The Flash Lite instance allocates and owns these buffers, filling them and passing them in turn to the SoundOutput instance. The Flash Lite instance fills each buffer with PCM samples, adding silent padding if needed.

The AudioParameters structure also specifies the desired sample rate of the audio output hardware. The SoundOutput object uses this sample rate if it cannot otherwise determine the sample rate of the audio output hardware.

Updated 13 May 2009
Implementations included with source distribution

The source distribution for Flash Lite for the digital home includes two sound driver implementations.

<table>
<thead>
<tr>
<th>Sound driver implementation</th>
<th>File location and description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>A generic sound driver implementation is in the directory source/ae/ddk/fl31nativesoundoutput/generic. This implementation simply discards PCM samples. Copy these files as a basis for your platform-specific sound driver implementation.</td>
</tr>
<tr>
<td>ALSA</td>
<td>An ALSA sound driver implementation is in the directory source/ae/ddk/fl31nativesoundoutput/alsa. For more information about customizing this sound driver implementation for your ALSA platform, see &quot;ALSA implementation&quot; on page 58.</td>
</tr>
</tbody>
</table>

Implementation tasks overview

To implement a platform-specific sound driver on a platform which does not use ALSA, do the following high-level tasks:

- Implement a class that derives from the SoundOutputBase class.
- Implement a class that derives from the IFL3NativeSoundOutput class.
- Copy the AudioParameters.h and AudioParameters.cpp files and modify the data member initializations as appropriate for your platform.

If your platform does use ALSA, do the following high-level tasks:

- Implement the AlsaParameters structure.
- Implement the AlsaMixerElem class.

SoundOutput and SoundOutputBase class hierarchy

The abstract SoundOutputBase class derives from the abstract SoundOutput class. SoundOutputBase adds only a public destructor. You implement a subclass of SoundOutputBase. The SoundOutput class hierarchy and methods are shown in the following illustration:
With three exceptions, the methods of a SoundOutput object are called by its corresponding Flash Lite instance. The three exceptions are the constructor, destructor, and `EnableOutput()`. The IFL31NativeSoundOutput singleton instance calls these three methods.

### SoundOutput methods

The Flash Lite instance calls all the SoundOutput methods directly or indirectly. Because the Flash Lite instance provides real-time output, it depends on SoundOutput methods to return quickly.

Some methods perform complex tasks that interact with the audio output hardware. These methods are `OpenStreamSoundDevice()`, `CloseStreamSoundDevice()`, `PlayStreamSoundBuffer()`, `PauseSound()`, and `ResumeSound()`. Except for `CloseStreamSoundDevice()`, implement all these methods to return quickly to the calling Flash Lite instance.

The method `CloseStreamSoundDevice()` is the exception. When `CloseStreamSoundDevice()` returns, the Flash Lite instance often overwrites or deletes the PCM buffers that it shares with the SoundOutput object. Therefore, make sure `CloseStreamSoundDevice()` has finished with the buffers before it returns.

The Flash Lite instance also depends on the return value from `GetNumBuffersPlayed()` to do subsequent processing. Specifically, the Flash Lite instance uses the return value to manage its allocation of buffers.

Another dependency on a return value involves the IFL31NativeSoundOutput singleton. It depends on the return value from `EnableOutput()`. The singleton uses the return value to keep its list of SoundOutput objects updated regarding which SoundOutput object has the focus.

For detailed definitions of return values and parameters of the SoundOutput and SoundOutputBase class methods, see include/ae/ddk/fl31nativesoundoutput/SoundOutput.h and source/ae/ddk/fl31nativesoundoutput/SoundOutputBase.h.
**OpenStreamSoundDevice() method**
This method initializes the platform-specific SoundOutput object. Initialize the SoundOutput object so it is ready to accept PCM sample buffers from the Flash Lite instance for immediate transfer to the audio output hardware.

The Flash Lite instance calls this method before calling any other SoundOutput method. If the Flash Lite instance calls CloseStreamSoundDevice(), then it calls OpenStreamSoundDevice() again before calling any other SoundOutput method.

**CloseStreamSoundDevice() method**
This method uninitializes the platform-specific SoundOutput. Close connections to the audio output hardware and release any allocated resources. Flush all pending PCM sample buffers through to the audio output hardware. Return from CloseStreamSoundDevice() only after all submitted buffers have been fully played.

**PlayStreamSoundBuffer() method**
This method accepts a PCM sample buffer for transfer to the audio output hardware. Transfer buffers to the audio output hardware in the order they are received. Buffers are always the size specified in AudioParameters.cpp. The Flash Lite instance always completely fills each buffer, adding silent padding if needed.

**PauseSound() method**
This method is a placeholder for a future feature to request the hardware to pause the audio output. Therefore, implement PauseSound() and ResumeSound() as stubs.

**ResumeSound() method**
This method is a placeholder for a future feature to request the hardware to resume the audio output. Therefore, implement PauseSound() and ResumeSound() as stubs.

**GetNumBuffersPlayed() method**
This method returns the number of submitted PCM sample buffers that the audio output hardware played to completion since the last call to this method. The Flash Lite instance calls this method for two reasons: buffer management and audio/video synchronization.

The Flash Lite instance owns and manages the PCM sample buffers. To know when it can reuse a buffer, the Flash Lite instance must recognize when the audio output hardware has played a buffer’s PCM samples. The audio output hardware plays the PCM samples in the order they are received. Therefore, the Flash Lite instance uses the value returned by GetNumBuffersPlayed() to determine which buffers are available. Return the number of buffers for which the hardware has finished playing all the PCM samples. If the hardware has only played some of the PCM samples in a buffer, do not include that buffer in the count. If you do, the Flash Lite instance sometimes overwrites the buffer before its playback is complete.

The Flash Lite instance also uses GetNumBuffersPlayed() to do audio/video synchronization. The Flash Lite instance depends on the return value to be as close as possible to the current number of played buffers. A played buffer is one for which all the PCM samples have actually exited the speakers. Any discrepancy results in audio/video skew.
EnableOutput() method

This method toggles the sound focus of the SoundOutput object. As described in “Enabling sound focus” on page 51, the IFL3NativeSoundOutput singleton calls this method of affected SoundOutputs object when the sound focus changes to a different StageWindow instance. This method takes a Boolean parameter. The value true indicates the SoundOutput object is receiving the sound focus. The value false indicates it is losing the focus. This method returns true when the SoundOutput has the focus. Otherwise it returns false.

**Note:** If the input parameter indicates receiving the focus, but the method fails to successfully set the focus, the return value is false.

When losing the focus, this method performs these tasks:

- Switches the destination of submitted PCM sample buffers from the audio output hardware to a null device.
- Releases the system’s only PCM audio output device for use by other SoundOutput objects.

When receiving the focus, this method performs these tasks:

- Switches the destination of submitted PCM sample buffers from a null device to the audio output hardware.
- Takes ownership of the system’s only PCM audio output device.

The method performs these tasks regardless of the state of the SoundOutput object. For example, the SoundOutput object can be either currently playing audio, paused, or idle in the stop state.

Furthermore, whether the SoundOutput object has the sound focus, it does not change its behavior with respect to its interaction with its Flash Lite instance. Specifically, the Flash Lite instance depends on `GetNumBuffersPlayed()` to return an accurate number even when the SoundOutput object does not have the sound focus. The Flash Lite instance depends on this number for buffer management and audio/video synchronization. Therefore, make sure the SoundOutput object accounts for each PCM sample buffer that the Flash Lite instance submitted to the SoundOutput object in `PlayStreamSoundBuffer()`.

For example, in the ALSA implementation, when the SoundOutput loses the sound focus, it first waits for the submitted PCM sample buffers to play to underflow. Then, the SoundOutput object switches to sending all subsequent buffers to null output. This “buffer boundary” implementation makes tracking the buffers simpler, because the hardware plays all the submitted PCM sample buffers before the switch.

Alternatively, an implementation can track the submitted PCM samples more closely. For example, an implementation can switch to sending samples to null output on a sample boundary instead of a buffer boundary. This implementation minimizes latency in switching to null output, but increases implementation complexity. The complexity increases because you still have to ensure that `GetNumBuffersPlayed()` returns an accurate value.

After the SoundOutput object is switched to null output, it discards samples submitted after the switch. However, it must simulate the playback of submitted data, ensuring that `GetNumberBuffersPlayed()` returns a correct value with the correct timing. Otherwise the Flash Lite instance encounters synchronization problems.

IFL31NativeSoundOutput class details

**IFL31NativeSoundOutput class definition**

The abstract IFL31NativeSoundOutput class derives from the abstract IAEModule class. You implement a subclass of IFL31NativeSoundOutput. Flash Lite for the digital home creates a singleton object of your IFL31NativeSoundOutput subclass.

The IFL31NativeSoundOutput class hierarchy and methods are given in the following illustration:
The responsibilities of your IFL31NativeSoundOutput singleton object are:

- Creating and destroying instances of your SoundOutput object.
- Enabling and disabling the sound focus of all the SoundOutput objects.
- Setting the output level for PCM sample output and StreamPlayer output.

Thread-safety considerations

Flash Lite for the digital home creates your IFL31NativeSoundOutput subclass as a singleton object. Therefore, calls to its methods come from different threads. For example, calls to its CreateSoundOutput() method come from different Flash Lite instances. Therefore, use a thread-safe implementation for your subclass of IFL31NativeSoundOutput.

FL31NativeSoundOutput class methods

IsSampleRateSupported() method

This method returns true if the audio output hardware supports the sample rate passed as a parameter. Flash Lite for the digital home uses this function to determine the set of rates that Flash Lite and the audio output hardware support. Flash Lite for the digital home then chooses the highest frequency supported by both.

The implementation of this method can involve using your hardware interface to determine if the rate is supported. Alternatively, the method can return true for the rate you want for your platform, without a query to the hardware interface.

Flash Lite for the digital home calls this method when it creates the Flash Lite instance for a StageWindow instance.
GetBufferInfo() method
This method returns the following as output parameters:

- The number of PCM sample buffers that the Flash Lite instance allocates.
- The size in frames of each PCM sample buffer.

A frame is defined as a unit containing one PCM sample for each channel. For example, with 16-bit samples, a frame of a mono signal is 16 bits, whereas a stereo frame is 32 bits. The Flash Lite instance uses these values to allocate the PCM sample buffers which it uses to transfer samples to its SoundOutput object. Typically, GetBufferInfo() returns the values that are hard coded in the AudioParameters structure.

Flash Lite for the digital home calls this method when it creates the Flash Lite instance for a StageWindow instance.

CreateSoundOutput() method
This method creates a SoundOutput object. A Flash Lite instance calls it only when the SWF content begins to generate PCM audio data. The parameters to CreateSoundOutput() specify:

- the audio sample rate.
- the number of bits in a single audio sample.
- the number of audio channels (one or two).

The IFLNativeSoundOutput singleton keeps a list of all the SoundOutput objects it creates. This list is necessary to support setting the sound focus. See “SetFocus() method” on page 57. The CreateSoundOutput() method adds the newly created SoundOutput object to the list.

DestroySoundOutput() method
This method destroys a SoundOutput object. A Flash Lite instance calls it when the SWF content no longer generates PCM audio data. A typical example is when Flash Lite for the digital home destructs the associated StageWindow instance.

This method deletes the SoundOutput instance, which calls the SoundOutput destructor. Therefore, be sure to derive the platform-specific SoundOutput subclass from SoundOutputBase because SoundOutputBase defines a public destructor.

This method removes the deleted SoundOutput object from the IFL31NativeSoundOutput singleton’s list of SoundOutput objects. The IFLNativeSoundOutput singleton keeps this list to support setting the sound focus. See “SetFocus() method” on page 57. The DestroySoundOutput() method removes the deleted SoundOutput object from the list. The method also optionally moves the focus to some other existing SoundOutput instance.

SetFocus() method
This method causes the SoundOutput object with the sound focus to give up the focus, and causes the specified SoundOutput object to take the sound focus.

As described in “Enabling sound focus” on page 51, Flash Lite for the digital home assumes that the platform has only one audio output device. When a StageWindow instance receives the focus, its Flash Lite instance calls SetFocus().
The IFL31NativeSoundOutput singleton keeps a list of existing SoundOutput objects. It also tracks which SoundOutput object has the sound focus. It calls the `EnableOutput()` method of the SoundOutput object with the focus (if one has the focus) to disable output to the audio output hardware. Then it calls the `EnableOutput()` method of the SoundOutput object specified in `SetFocus()` to enable output to the audio output hardware. For more information, see “EnableOutput() method” on page 55.

**GetLevel() method**

This method gets the volume level of the hardware mixer’s PCM sample input or the hardware mixer’s StreamPlayer input. The method returns a value between a 0 and 100, because that is the range that the Flash Lite instance uses. If necessary, scale your hardware’s level to fit into this range. A scaling example is in `source/ae/ddk/fl31nativesoundoutput/alsa/AlsaMixer.cpp`.

For more information, see “Mixing audio inputs” on page 51.

**SetLevel() method**

This method sets the volume level of the hardware mixer’s PCM sample input or the hardware mixer’s StreamPlayer input. The method sets a value between a 0 and 100, because that is the range that the Flash Lite instance uses. If necessary, scale your hardware’s level to fit into this range. A scaling example is in `source/ae/ddk/fl31nativesoundoutput/alsa/AlsaMixer.cpp`.

For more information, see “Mixing audio inputs” on page 51.

**ALSA implementation**

Flash Lite for the digital home provides with its source distribution complete implementations of IFL31NativeSoundOutput and SoundOutput based on the ALSA PCM and Simple Mixer interfaces. The source is in `source/ae/ddk/fl31NativeSoundOutput/alsa`.

*Note:* The ALSA implementation assumes multiple processes of Flash Lite for the digital home can concurrently exist. However, Flash Lite for the digital home does not yet support this feature. Flash Lite for the digital home supports only multiple StageWindow instances in one process.

To use the ALSA implementation, provide the following customizations.

**AlsaParameters structure**

The file `source/ae/ddk/fl31NatveSoundOutput/alsa/AlsaParameters.h` defines the AlsaParameters structure. Provide an implementation for this structure. The implementation sets the values of the structure members. See `AlsaParameters.h` for more information about what to set the values to. `AlsaParametersX86.cpp` in the same directory shows an example implementation.

**AlsaMixerElem class**

The file `source/ae/ddk/fl31NatveSoundOutput/alsa/AlsaMixerElem.h` defines the AlsaMixerElem class. Provide an implementation for this class. The implementation provides details on how to initialize and operate the ALSA Simple Mixer device on your platform. For example, the AlsaMixerElem implementation includes:

- the card name of the ALSA device
- logic to find the handles to the required mixer elements
- logic to initialize the device and set the device to its default state.
See AlsaMixerElem.h for more information about the AlsaMixerElem class. AlsaMixerELEMX86.cpp in the same directory shows an example implementation.

**Creating files for your platform-specific sound driver**

Put the header and source files for your platform-specific sound driver in a subdirectory of the thirdparty-private/stagecraft-platforms directory. For information, see “Building platform-specific drivers and decoders” on page 79.

You can use the implementations provided by the source distribution without modification if they meet your needs. Otherwise, copy them to use as a starting point for your own implementation. For more information on the source distribution implementations, see “Implementations included with source distribution” on page 52.

**Building your platform-specific sound driver**

For information about building your sound driver, see “Building platform-specific drivers and decoders” on page 79.
Chapter 5: The image decoder

Adobe® Flash® Lite® for the digital home plays SWF content. This content often includes JPEG and PNG images. Flash Lite for the digital home provides software decoders that decode these images. However, decoding these images with a dedicated hardware decoder is faster. This speed can make the SWF content start up faster and provide better response time.

To direct Flash Lite for the digital home to use a hardware image decoder and associated APIs of your target platform, you implement the image decoder interfaces. The interfaces are abstract C++ classes.

Class overview

The image decoder includes these classes, which are defined in include/ae/ddk/imagedecoder/IImageDecoder.h:

<table>
<thead>
<tr>
<th>Class or structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageDecoder</td>
<td>Abstract class that you implement that defines the interfaces that Flash Lite for the digital home uses to decode a JPEG or PNG image.</td>
</tr>
<tr>
<td>IImageDecoder</td>
<td>Abstract class, derived from IAEModule, that you implement. The IImageDecoder subclass is the factory for creating and destroying your platform-specific image decoder.</td>
</tr>
<tr>
<td>DecodeRequest</td>
<td>Abstract class that the Flash Lite module implements to provide information about which type of image decoder to create, and how to decode the image.</td>
</tr>
</tbody>
</table>

The methods you implement for your platform-specific image decoder module (IImageDecoder subclass) are:

- CreateImageDecoder()
- DestroyImageDecoder()

The methods you implement for your platform-specific image decoder (ImageDecoder subclass) are:

- DecodeImageHeader()
- DecodeImageData()
- AbortDecode()
- GetAdjustedScaleDims()

Class interaction and logic flow

The host application is defined in “Running Flash Lite for the digital home” on page 3. The host application interacts with Flash Lite for the digital home to run Flash Lite applications. Specifically, the host application interacts with the IStagecraft module. Using this interface, the host application creates a StageWindow instance. The StageWindow instance contains an instance of Adobe® Flash® Lite® 3.1. The Flash Lite instance loads the SWF file specified by the host application.

The SWF content can include PNG and JPEG images. The Flash Lite instance and your platform-specific image decoder module and image decoder interact to decode the images.
Acquiring the image decoder module

When the Flash Lite instance renders an image from the SWF content, the Flash Lite instance acquires the image decoder module. The image decoder module is an instance of your platform-specific IImageDecoder subclass.

Setting up the decode request

The Flash Lite instance asks the image decoder module to create an image decoder. But first, it creates and initializes an instance of a class that implements the DecodeRequest interface. The Flash Lite instance initializes the DecodeRequest object with the following information:

- The type of the source image: JPEG or PNG.
- A pointer to the encoded image data for the source image.
- The size of the encoded image data.
- A pointer to a MemoryWatchdog object.

Creating the image decoder

After creating and initializing the DecodeRequest object, the Flash Lite instance asks the image decoder module to create an image decoder. To do so, the Flash Lite instance calls the CreateImageDecoder() method of the image decoder module. The Flash Lite instance passes a pointer to the DecodeRequest object as a parameter in the method call.

In your implementation of CreateImageDecoder(), use the type of the source image in the DecodeRequest object to determine how to create your platform-specific image decoder. In your newly created image decoder, save the pointer to the DecoderRequest object. The image decoder uses the methods of the DecoderRequest object throughout its lifecycle.

Decoding the image header

After creating the image decoder, the Flash Lite instance asks it to decode the image header. To do so, the Flash Lite instance calls the DecodeImageHeader() method of your platform-specific image decoder. Your implementation of DecodeImageHeader() includes the following tasks:

1. Use these DecodeRequest object methods to get information about the image: GetSourceImageType(), GetSourceData(), GetSourceDataSize(). Use the DecodeRequest object method GetMemoryWatchdog() to get a pointer to a MemoryWatchdog object for allocating and deallocating system memory.

   **Note:** Do not call any other methods of the DecodeRequest object at this point in the lifecycle of the image decoder. Other methods do not yet have valid data to return.

2. Decode the header information. If not null, use the MemoryWatchdog pointer for system memory allocation and deallocation.

3. Call the NotifyImageHeaderDecodeComplete() method of the DecodeRequest object. You pass this method the height, width, and color format of the image. You also pass this method whether the image has transparency (alpha) data. The DecodeRequest object stores this information.
Preparation to decode the image data

After your image decoder has decoded the header data, the Flash Lite instance prepares to decode the image data. This preparation includes some tasks that affect your image decoder. Specifically, the Flash Lite instance:

- Prepares a Plane object. Your image decoder decodes the image data into this Plane object. The Flash Lite instance stores a pointer to the Plane object in the DecodeRequest object. Later, your image decoder uses the `GetDecodeTargetPlane()` method of the DecodeRequest object to get the pointer.
- Determines the scale factor. For JPEG images, your image decoder uses this factor to downscale the image during decoding. For images which are not JPEG images, the Flash Lite instance sets this factor to 1. The Flash Lite instance stores the scale factor in the DecodeRequest object. Later, your image decoder uses the `GetScaleFactor()` method of the DecodeRequest object to get the scale factor.
- Adjusts the image dimensions (for JPEG images only). Your image decoder provides the `GetAdjustedScaledDims()` method to assist with this task.

Decoding the image data

After making preparations for decoding the image data, the Flash Lite instance calls the `DecodeImageData()` method of your platform-specific image decoder. Your implementation of `DecodeImageData()` includes the following tasks:

1. Use these DecodeRequest object methods to get information about the image: `GetSourceImageType()`, `GetSourceData()`, `GetSourceDataSize()`.
2. Use these DecodeRequest object methods to get information decoded in the previous call to `DecodeHeaderData()`: `GetHeaderImageWidth()`, `GetHeaderImageHeight()`, and `GetHeaderImageColorFormat()`.
3. Use the DecodeRequest object method `GetTargetDecodePlane()` to get a pointer to the plane in which to decode the image.
4. Call the Plane object’s `Lock()` method to get a pointer to the bitmap of the plane. Don’t forget to call the Plane object’s `Unlock()` method when decoding is complete.
5. Use the Plane object’s `GetClassName()` method to determine the type of the Plane object. Determining the type is necessary if your `DecodeImageData()` implementation accesses publicly accessible methods for that specific Plane type. Cast the Plane object pointer to the more specific type. For example:
   ```cpp
   ae::stagecraft::Plane * pPlane = m_pDecodeRequest->GetDecodeTargetPlane();
   if (strcmp(pPlane->GetClassName(), MY_PLATFORM_PLANE_CLASS_NAME) == 0)
   {
     MyPlatformPlane * pMyPlatformPlane = (MyPlatformPlane *) pPlane;
     pMyPlatformPlane->MyPublicMethod();
   }
   ```
6. Decode the image data. If not null, use the MemoryWatchdog pointer for system memory allocation and deallocation. The DecodeRequest object’s `GetMemoryWatchdog()` returns this pointer.
7. Call the Plane object’s `Unlock()` method if you previously called the `Lock()` method.

The Flash Lite instance now has the decoded image in the bitmap from the Plane object. The Flash Lite instance deletes the DecodeRequest object. It also destroys the image decoder by calling the `DestroyImageDecoder()` method of the image decoder module.
Aborting an image decode request

Sometimes the Flash Lite instance attempts to cancel a request it previously made to decode an image. This attempt occurs when, for example, the StageWindow instance for the Flash Lite instance exits before the SWF content finishes playing. The Flash Lite instance calls your ImageDecoder object’s `AbortDecode()` method. In `AbortDecode()`, provide the logic to stop decoding the image header or data, depending on the state of your ImageDecoder. Depending on your implementation, this method can do nothing.

*Note:* The thread calling `AbortDecode()` is not the same as the thread which called your ImageDecoder object’s `DecodeImageHeader()` and `DecodeImageData()` methods. Therefore, code `AbortDecode()` in a thread-safe manner.

Synchronous or asynchronous implementation

Your implementations of the image decoder methods `DecodeImageHeader()` and `DecodeImageData()` can be either synchronous or asynchronous. If you implement them synchronously, call `NotifyImageHeaderDecodeComplete()` or `NotifyImageDataDecodeComplete()` before returning. If you implement them asynchronously, return without calling the notification method. However, as soon as the header or image data has been decoded, call the appropriate notification method.

Implementations included with source distribution

The source distribution for Flash Lite for the digital home include a JPEG image decoder software implementation and a PNG image decoder software implementation. The files are in source/ae/ddk/imagedecoder. You can copy these files as a basis for your own implementations.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageDecoderImpl.h</td>
<td>An implementation of the ImageDecoder module. This image decoder module creates either a JpegImageDecoder object or a PngImageDecoder object, depending on the type of image.</td>
</tr>
<tr>
<td>ImageDecoderImpl.cpp</td>
<td>An abstract implementation of the ImageDecoder class. The JpegImageDecoder and PngImageDecoder classes derive from ImageDecoderImpl. The ImageDecoderImpl class provides implementations of methods common to the JpegImageDecoder and PngImageDecoder classes.</td>
</tr>
<tr>
<td>JpegImageDecoder.h</td>
<td>An implementation of the ImageDecoderImpl class for decoding JPEG images.</td>
</tr>
<tr>
<td>JpegImageDecoder.cpp</td>
<td>An implementation of the ImageDecoderImpl class for decoding JPEG images.</td>
</tr>
<tr>
<td>PngImageDecoder.h</td>
<td>An implementation of the ImageDecoderImpl class for decoding PNG images.</td>
</tr>
<tr>
<td>PngImageDecoder.cpp</td>
<td>An implementation of the ImageDecoderImpl class for decoding PNG images.</td>
</tr>
</tbody>
</table>

Creating files for your platform-specific image decoder

Put the header and source files for your platform-specific image decoder in a subdirectory of the thirdparty-privatestagecraft-platforms directory. For information, see “Building platform-specific drivers and decoders” on page 79.
You can use the implementations provided by the source distribution without modification if they meet your needs. Otherwise, copy them to use as a starting point for your own implementation. For more information on the source distribution implementations, see “Implementations included with source distribution” on page 63.

**Building your platform-specific image decoder**

For information about building your image decoder, see “Building platform-specific drivers and decoders” on page 79.
Chapter 6: The audio decoder

Adobe® Flash® Lite® for the digital home plays SWF content. This content includes compressed audio data. Flash Lite for the digital home provides audio software decoders that decompress the data into PCM samples. These software decoders decompress mp3, PCM, and ADPCM codecs. If your platform provides hardware that can decompress the audio data faster, you can direct Flash Lite for the digital home to utilize your hardware. To do so, you implement the audio decoder interfaces. The interfaces are abstract C++ classes.

Before implementing a hardware audio decoder, use the provided software decoders to see if the performance is sufficient. Unlike the sound driver, which you must implement for playing audio output, replacing the software decoders is optional. You can replace them with platform-specific hardware audio decoders for decoding the audio.

The audio decoders pass the decoded PCM samples back to the Flash Lite instance. A software mixer inside the Flash Lite instance mixes multiple audio decoder outputs and sends them to the output mixer as one output.

*Note:* The StreamPlayer sends another audio output to the hardware mixer.

Class overview

The audio decoder includes these classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AudioDecoder</td>
<td>Abstract class you implement that defines the interfaces Flash Lite for the digital home uses to interact with hardware that decodes mp3, PCM, or ADPCM codecs.</td>
</tr>
<tr>
<td>IAudioDecoder</td>
<td>Abstract class, derived from IAEModule, that you implement. The IAudioDecoder subclass is the factory for creating and destroying your platform-specific audio decoder.</td>
</tr>
</tbody>
</table>

The AudioDecoder and IAudioDecoder classes are defined in the files AudioDecoder.h and IAudioDecoder.h in the directory include/ae/ddk/audiodecoder.

Class interaction and logic flow

The host application is defined in “Running Flash Lite for the digital home” on page 3. The host application interacts with Flash Lite for the digital home to run Flash Lite applications. Specifically, the host application interacts with the IStagecraft module. Using this interface, the host application creates a StageWindow instance. The StageWindow instance contains an instance of Adobe® Flash® Lite® 3.1. The Flash Lite instance loads the SWF file specified by the host application.

The SWF content sometimes includes compressed audio. The Flash Lite instance and your platform-specific audio decoder module (IAudioDecoder subclass) and audio decoder (AudioDecoder subclass) interact to decode the audio.

When the Flash Lite instance loads a SWF file, the Flash Lite instance acquires the audio decoder module. When the Flash Lite instance prepares to decompress audio in the SWF content, it asks the audio decoder module to create an audio decoder. The audio decoder module uses a parameter in the request to determine what type of AudioDecoder object to create. For example, if the codec is mp3, the audio decoder module has logic to create an object for which the type is the AudioDecoder subclass for mp3 decoding.
Then, the Flash Lite instance uses the newly created audio decoder to decode the audio data. The Flash Lite instance provides the audio decoder a pointer to a buffer containing the compressed data. It also provides a pointer to another buffer in which to put the decoded data. After the audio decoder has finished decoding the audio data, the Flash Lite instance asks the audio decoder module to destroy the audio decoder. The Flash Lite instance possibly mixes the decoded audio data with other decoded audio data. It then uses the sound driver module to play the audio output.

### Implementations included with source distribution

The source distribution for Flash Lite for the digital home includes AudioDecoder implementations for mp3, PCM, and ADPCM codecs. The files are in source/ae/ddk/audiodecoder. You can copy these files as a basis for your own implementation.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAudioDecoderImpl.h</td>
<td>This IAudioDecoder factory implementation creates and destroys the provided audio decoders: AudioDecoderPcm, AudioDecoderAdpcm, and AudioDecoderMp3. The files are in the directory source/ae/ddk/audiodecoder/platform/generic.</td>
</tr>
<tr>
<td>IAudioDecoderImpl.cpp</td>
<td></td>
</tr>
<tr>
<td>AudioDecoderBase.h</td>
<td>The AudioDecoderBase abstract class derives from AudioDecoder. The AudioDecoderBase class provides logic common to the AudioDecoderPcm, AudioDecoderAdpcm, and AudioDecoderMp3 classes. The files are in the directory source/ae/ddk/audiodecoder.</td>
</tr>
<tr>
<td>AudioDecoderBase.cpp</td>
<td></td>
</tr>
<tr>
<td>AudioDecoderPcm.h</td>
<td>The AudioDecoderPcm class derives from AudioDecoderBase. It is the AudioDecoder implementation for decoding compressed PCM data. The files are in the directory source/ae/ddk/audiodecoder/pcm/software.</td>
</tr>
<tr>
<td>AudioDecoderPcm.cpp</td>
<td></td>
</tr>
<tr>
<td>AudioDecoderAdpcm.h</td>
<td>The AudioDecoderAdpcm class derives from AudioDecoderBase. It is the AudioDecoder implementation for decoding compressed ADPCM data. The files are in the directory source/ae/ddk/audiodecoder/adpcm/software.</td>
</tr>
<tr>
<td>AudioDecoderAdpcm.cpp</td>
<td></td>
</tr>
<tr>
<td>AudioDecoderMp3.h</td>
<td>The AudioDecoderMp3 class derives from AudioDecoderBase. It is the AudioDecoder implementation for decoding compressed Mp3 data. The directory source/ae/ddk/audiodecoder/mp3/software/mad contains the MPEG Audio Decoder library. The AudioDecoderMp3 implementation uses this library for decoding.</td>
</tr>
<tr>
<td>AudioDecoderMp3.cpp</td>
<td></td>
</tr>
</tbody>
</table>

### AudioDecoder methods

For detailed definitions of return values and parameters of the AudioDecoder class methods, see include/ae/ddk/audiodecoder/AudioDecoder.h.

#### Setup() method

This method configures an AudioDecoder object with the information it requires to decode a buffer of compressed data. The Flash Lite instance calls `Setup()` at least once before each call it makes to the `Decompress()` method of the AudioDecoder object. When the Flash Lite instance calls `Setup()` multiple times before calling `Decompress()`, each call adds more data to the pool of compressed sound data to decode.
The parameters to Setup() are:

- **pSrcBuf** - a pointer to the source buffer. This buffer contains the compressed audio data for the next call to Decompress() to use. The audio data format depends on the type of the audio decoder.
- **nBytes** - the number of bytes of compressed data to decode in the next call to Decompress().
- **delaySamples** - the number of delay samples. This number indicates how many decoded PCM samples to skip when copying decoded samples to the destination buffer in the next call to Decompress(). Therefore, the decoder decodes this many samples, but does not output them for playback.
- **reset** - a Boolean flag for resetting the AudioDecoder object. When this flag is true, ignore any previous audio data submitted. Reset the AudioDecoder object to its initialized state.

**Decompress() method**

This method decodes the compressed audio data provided in the preceding call to Setup(). The Flash Lite instance calls this method one or more times following a call to Setup(). In each call, the Flash Lite instance passes these parameters:

- **pDstBuf** - a pointer to a buffer to contain the decoded PCM samples.
- **nSamples** - the number of samples requested. The Decompress() method decodes at most this many samples. However, Decompress() decodes fewer samples than nSamples if it reaches the end of the source buffer.

The Decompress() method returns the actual number of samples it decoded. If an error occurs, it returns -1. The Flash Lite instance calls this method as many times as needed to consume the compressed data submitted by Setup().

**NumBytesPerSample() method**

This method returns the number of bytes in a decoded PCM sample. The value depends on the sample size and number of channels in the decoded audio stream. This method performs no decoding, but provides a convenient interface for the Flash Lite instance to convert the size of a buffer from samples to bytes. With this method, the Flash Lite instance can do this conversion without needing to know the number of bytes per sample or number of channels.

**IAudioDecoder class methods**

For detailed definitions of return values and parameters of the IAudioDecoder class methods, see include/ae/ddk/audiodecoder/IAudioDecoder.h.

**GetSupportedFormats() method**

This method returns a list of audio input codecs for which the IAudioDecoder subclass can create an AudioDecoder object. The list is an array of values of the DecoderType enumeration, defined in IAudioDecoder.h.

**CreateDecoder() method**

This method creates an AudioDecoder object. It uses the parameters to determine what type of AudioDecoder object to create. The parameters are:

- **type** - the type of decoder to create. This parameter is one of the values of the DecoderType enumeration.
- **sampleRateHz** - the audio sample rate in hertz.
- **sampleSizeBits** - the audio sample size in bits.
• `nChannels` - the number of channels.

If your platform-specific AudioDecoder automatically determines the sample rate, the sample size, and the number of channels, then ignore these parameters.

`CreateDecoder()` returns a pointer to the newly created AudioDecoder object. If `CreateDecoder()` fails to create an AudioDecoder object, it returns `NULL`. Reasons for failure include not being able to support the requested codec.

**DestroyDecoder() method**

This method destroys an AudioDecoder object. A pointer to the object to destroy is passed as a parameter.

**Creating files for your platform-specific audio decoder**

Put the header and source files for your platform-specific audio decoder in a subdirectory of the thirdparty-private/stagecraft-platforms directory. For information, see “Building platform-specific drivers and decoders” on page 79.

You can use the implementations provided by the source distribution without modification if they meet your needs. Otherwise, copy them to use as a starting point for your own implementation. For more information on the source distribution implementations, see “Implementations included with source distribution” on page 66.

**Building your platform-specific audio decoder**

For information about building your audio decoder, see “Building platform-specific drivers and decoders” on page 79.
Chapter 7: The video decoder

Adobe® Flash® Lite® for the digital home plays SWF content. This content sometimes includes video. The video can be embedded in the SWF file, or can be progressively downloaded from file:// or http:// URLs. Alternatively, the video can be streamed from an Adobe® Flash® Media Server.

Typically, platform-specific StreamPlayers decode and render progressively downloaded and streamed video. For more information, see "The overlay video driver" on page 31. On the other hand, embedded video is decoded and rendered by a Flash Lite instance using internal software decoders. These internal software decoders can also decode and render video that is not embedded. However, typically, using the internal software decoders is too slow to be acceptable in platforms using Flash Lite for the digital home.

An alternative to the internal software decoders is to use platform-specific hardware video decoders to decode the video and render each frame of embedded videos. However, unlike StreamPlayers, the video decoder returns each frame to the Flash Lite instance for compositing with other layers on the Stage. The hardware decoder alternative is typically too slow for platforms using Flash Lite for the digital home, unless the video involved is a small thumbnail video.

However, your platform might have a use case for video hardware decoders. For example, the video hardware decoder might use your platform’s assembly instructions to improve processing time. Therefore, Flash Lite for the digital home provides interfaces to decode videos using hardware video decoders. The supported codecs are On2 VP6, Sorenson H.263, and H.264. To direct Flash Lite for the digital home to utilize your hardware video decoders, implement the video decoder interfaces. The interfaces are abstract C++ classes.

Class overview

The video decoder includes these classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VideoDecoder</td>
<td>Abstract class that you implement. Defines the interfaces Flash Lite for the digital home uses to interact with hardware that decodes and renders either On2 VP6, Sorenson H.263, or H.264 codecs.</td>
</tr>
<tr>
<td>IVideoDecoder</td>
<td>Abstract class, derived from IAEModule, that you implement. The IVideoDecoder subclass is the factory for creating and destroying your platform-specific video decoder.</td>
</tr>
</tbody>
</table>

The VideoDecoder and IVideoDecoder classes are defined in the files VideoDecoder.h and IVideoDecoder.h in the directory include/ae/ddk/videodecoder.

Class interaction and logic flow

The host application is defined in "Running Flash Lite for the digital home" on page 3. The host application interacts with Flash Lite for the digital home to run Flash Lite applications. Specifically, the host application interacts with the IStagecraft module. Using this interface, the host application creates a StageWindow instance. The StageWindow instance contains an instance of Flash Lite 3.1. The Flash Lite instance loads the SWF file specified by the host application.
The SWF content sometimes includes embedded compressed video. The Flash Lite instance and your platform-specific video decoder module (IVideoDecoder subclass) and video decoder (VideoDecoder subclass) interact to decode and render the video.

When the Flash Lite instance loads a SWF file, the Flash Lite instance acquires the video decoder module. When the Flash Lite instance prepares to decompress video embedded in the SWF content, it asks the video decoder module to create a video decoder. The video decoder module uses a parameter in the request to determine which type of VideoDecoder object to create. For example, if the codec is Sorenson H.263, the video decoder module creates an object for which the type is the VideoDecoder subclass for Sorenson H.263 decoding.

Then, the Flash Lite instance uses the newly created video decoder to decode and render the video data. The Flash Lite instance provides the video decoder a pointer to a buffer containing the compressed data. The video decoder decompresses the data into its own buffers. Then, the Flash Lite instance provides pointers to buffers into which the video decoder blits the decompressed data. The Flash Lite instance composites each decompressed video frame with other layers on the Stage. Once the video has been rendered, the Flash Lite instance asks the video decoder module to destroy the video decoder.

### Implementations included with source distribution

The source distribution for Flash Lite for the digital home includes VideoDecoder implementations for Sorenson H.263 and On2 VP6 codecs. The files are in source/ae/ddk/videodecoder. You can copy these files as a basis for your own implementation.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVideoDecoderImpl.h</td>
<td>This IVideoDecoder factory implementation creates and destroys the provided software video decoders: VideoDecoderSorenson and VideoDecoderOn2VP6. The files are in the directory source/ae/ddk/videodecoder.</td>
</tr>
<tr>
<td>IVideoDecoderImpl.cpp</td>
<td></td>
</tr>
<tr>
<td>VideoDecoderSorenson.h</td>
<td>The VideoDecoderSorenson class derives from VideoDecoder. It is the VideoDecoder implementation for decoding and rendering the Sorenson H.263 codec. The files are in the directory source/ae/ddk/videodecoder/sorenson/software.</td>
</tr>
<tr>
<td>VideoDecoderSorenson.cpp</td>
<td></td>
</tr>
<tr>
<td>VideoDecoderOn2VP6.h</td>
<td>The VideoDecoderOn2VP6 class derives from VideoDecoder. It is the VideoDecoder implementation for decoding and rendering the On2 VP6 codec. The files are in the directory source/ae/ddk/videodecoder/on2vp6/software.</td>
</tr>
<tr>
<td>VideoDecoderOn2VP6.cpp</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The source distribution provides a stub for video decoder for H.264.

### VideoDecoder structures

The structures are defined in include/ae/ddk/videodecoder/VideoDecoder.h.

#### DecoderMessage struct

The DecoderData struct contains a DecoderMessage struct field. The Flash Lite instance and a VideoDecoder object use these structures to exchange information.

**Note:** Do not change the type, order, or number of fields in DecoderMessage struct.

Updated 13 May 2009
struct DecoderMessage
{
    // Reserved. Do not use.
    char reserved1;

    // Pointer to the compressed bitstream buffer.
    // The Flash Lite instance uses this field to provide the compressed data
    // to a VideoDecoder object.
    u8 * data;

    // Reserved. Do not use.
    u32 reserved2;

    // Length in bytes of the bitstream buffer.
    // The Flash Lite instance uses this field to indicate to the Flash Lite instance
    // the length of the compressed data pointed to by the data field.
    u32 length;
};

DecoderData struct
The Flash Lite instance and a VideoDecoder object use the DecoderData structure to exchange information.

struct DecoderData
{
    // The Flash Lite instance uses this field to provide the compressed data to a
    // VideoDecoder object.
    DecoderMessage decodeMsg;
    // The isEmbeddedVideo field has the value true when the compressed frame
    // is part of a video embedded in the timeline of the SWF content. The value false
    // indicates the frame is part of a streamed video.
    //
    // When the flag is false (streaming video), the VideoDecoder object uses
    // the header at the beginning of the data stream to get the height and width
    // of the image, as well as padding information. The VideoDecoder object then
    // discards the header before decoding the frame.
    //
    // When the flag is true (embedded video), the VideoDecoder object assumes no
    // header is present. The VideoDecoder object gets the height and
    // width from fields in this DecoderData struct.
    bool isEmbeddedVideo;

    // The height and width in pixels of the video frame.
    //
    // When the Flash Lite instance calls the DecompressFrame() method of the
OPTIMIZING FLASH LITE FOR THE DIGITAL HOME

The video decoder

// VideoDecoder object, and isEmbeddedVideo is true, the Flash Lite instance fills
// in the height and width values.
//
// When the Flash Lite instance calls the GetFrameDimensions() method of the
// VideoDecoder object, the VideoDecoder object fills in the height and width
// values.
//
uint32 height;
uint32 width;

// The Boolean canAcceptMoreData is true when the VideoDecoder decoded all the
// data passed in the DecoderMessage field. The VideoDecoder object sets
// this field when the Flash Lite instance calls DecompressFrame(). The VideoDecoder
// object sets this field to false when it cannot accept more data for now. When
// false, the Flash Lite instance sends the same data again later.
//
bool canAcceptMoreData;
// Reserved. Do not use.
DecoderMessage* reserved1;
};

VideoDecoder methods

For detailed definitions of return values and parameters of the VideoDecoder class methods, see
include/ae/ddk/videodecoder/VideoDecoder.h.

GetDecoderType() method

This method returns the type of codec that the VideoDecoder object decodes. The return value is one of the values of
the DecoderType enumeration.

IsKeyFrame() method

This method returns true if the pointer passed as a parameter is pointing to compressed data that is a keyframe. Otherwise, this method returns false.

GetFrameDimensions() method

This method returns the width and height of the keyframe passed as a parameter. The data field in the
DecoderMessage structure in the DecoderData parameter points to the compressed keyframe. The method fills in the
width and height fields of the DecoderData structure. The return value is true when the method is successful.
Otherwise, the return value is false.

The Flash Lite instance calls GetFrameDimensions() to determine the destination frame size. Then, the Flash Lite
instance calls DecompressFrame(). In GetFrameDimensions(), do not modify the data pointed to by the data field.

DecompressFrame() method

This method returns true if it successfully decompresses and renders the specified compressed frame. Otherwise, it
returns false. The data field in the DecoderMessage structure in the DecoderData parameter points to the
compressed frame.

Updated 13 May 2009
**BlitRow() method**

This method returns `true` if it successfully blits the requested row (or portion of a row) of the decompressed frame into a destination buffer. Otherwise, it returns `false`. The Flash Lite instance calls `DecompressFrame()`, followed by multiple calls to `BlitRow()`.

The parameters to `BlitRow()` are:

- `column` - the column number at which blitting is to begin. The column number is in 16.16 format, and ranges in value from 0 to the width of a frame.
- `row` - the row number at which blitting is to begin. The row number is in 16.16 format, and ranges in value from 0 to the height of a frame.
- `nPixels` - the number of pixels to blit. This number is less than or equal to the width of a frame.
- `dest` - a pointer to a buffer to blit into. `BlitRow()` can assume that the destination buffer has enough room. The destination format is `ARGB8888`.

`BlitRow()` blits `nPixels` pixels into the `dest` buffer. `BlitRow()` blits the specified row of the last decompressed frame, starting at the specified column.

**Flush() method**

This method returns `true` if it successfully clears the VideoDecoder object’s pipeline and image buffer. Returning `true` indicates the VideoDecoder object has reinitialized its internal data and is ready for a new keyframe. Otherwise, return `false`.

**IVideoDecoder class methods**

For detailed definitions of return values and parameters of the IVideoDecoder class methods, see `include/ae/ddk/videodecoder/IVideoDecoder.h`.

**CreateDecoder() method**

This method creates a VideoDecoder object. It uses the parameter, a `DecoderType` enumeration value, to determine which VideoDecoder subclass to create.

`CreateDecoder()` returns a pointer to the newly created VideoDecoder object. If `CreateDecoder()` fails to create an VideoDecoder object, it returns `NULL`. Reasons for failure include not being able to support the requested codec.

**DestroyDecoder() method**

This method destroys a VideoDecoder object. A pointer to the object to destroy is passed as a parameter.

**Creating files for your platform-specific video decoder**

Put the header and source files for your platform-specific video decoder in a subdirectory of the thirdparty-private/stagecraft-platforms directory. For information, see “Building platform-specific drivers and decoders” on page 79.
You can use the implementations provided by the source distribution without modification if they meet your needs. Otherwise, copy them to use as a starting point for your own implementation. For more information on the source distribution implementations, see "Implementations included with source distribution" on page 70.

**Building your platform-specific video decoder**

For information about building your video decoder, see “Building platform-specific drivers and decoders” on page 79.
Chapter 8: Coding, building, and testing

The Adobe® Flash® Lite® for the digital home source distribution provides Linux® and Win32 implementations of commonly used programming types, macros, and functions. If your platform uses a different operating system, a system developer for your platform provides the implementations. You, as a platform driver or decoder developer, however, only work with the definitions and interfaces of these programming constructs.

Flash Lite for the digital home also provides a framework for using the make utility to build your driver or decoder. The build process for Flash Lite for the digital home requires you to place your source and header files in specific directories. The build process also depends on configuration files you provide.

Lastly, Flash Lite for the digital home provides a utility to measure the performance of your drivers and decoders.

Common types and macros

The file AETypes.h is in stagecraft/include/ae. (In these examples, stagecraft is the installation directory of Flash Lite for digital home.) AETypes.h provides typedefs and macros based on the toolchain or operating system. This file is the only file that contains #ifdefs based on the operating system. The AETypes.h file in the source distribution provides the implementation for Linux platforms, as well as for a Win32 development environment. If your platform does not use Linux, a system developer for your platform provides the implementation.

Use these typedefs and macros to keep your platform-specific drivers portable. In your platform-specific driver, use the following typedefs and macros.

- **Typedefs for common character and integer types.** For example:
  ```c
  typedef unsigned char u8;
  ```

- **Macros for swapping bytes.** For example:
  ```c
  #define AE_SWAP16(n) ( (((n) >> 8) & 0x00FF) | (((n) << 8) & 0xFF00) )
  ```

- **Macros for converting numbers between big endian and little endian order.** For example:
  ```c
  #define AE_BE32(n)          AE_SWAP32((n))
  ```

- **Macros for memory allocation and deallocation.** For example:
  ```c
  #define AE_NEW ::new(NULL, 0, (AEMem_Selector_AE_NEW_DELETE *)0)
  ```

The file also includes debug versions of the memory macros. The debug versions provide additional information about the memory manipulation.

Kernel functionality

Flash Lite for the digital home architecture includes a kernel called the Adobe Electronics Kernel. The kernel provides Flash Lite for the digital home some fundamental functionality. For example, the kernel provides the ability to load modules, to work with threads, and to do string processing. The source distribution provides the kernel implementation for Linux platforms, as well as for a Win32 development environment. If your platform does not use Linux, a system developer for your platform provides the kernel implementation.
As a developer of platform-specific drivers, you also use some of the kernel functionality. The public interface file is in stagecraft/include/ae/IAEKernel.h.

To access kernel methods, use the static GetKernel() method. For example:

IAEKernel::Thread *pThread = IAEKernel::GetKernel()->CreateThread();

### Fixed-point numbers

The kernel provides the FixedPoint class. Use this class to do fixed-point arithmetic and comparisons using integer numerators and denominators.

### Time

Your platform’s kernel implementation provides a set of methods related to time. For example, these methods include GetTimeGMT(), TimeToCalendarTime(), SetTimer(), and Sleep(). For a complete list of time-related methods, see IAEKernel.h.

These methods work with objects derived from the Time abstract class. A Time class implementation works with time in nanoseconds, and provides the following:

- Get and Set methods for number of nanoseconds, microseconds, milliseconds, and seconds.
- Arithmetic and comparison operators.
- Methods for setting the Time to a constant representing forever and for checking for forever.

Some of the kernel methods also use a CalendarTime object. The CalendarTime class works with the date and time, given as the year, month, day, hour, minute, and second.

The kernel also provides a CountdownTimer class. Use this class for setting a time duration and checking on its progress.

### Threads

Use your platform’s kernel implementation of CreateThread(), DestroyThread(), and GetCurrentThread() for thread manipulation. These methods work with objects derived from the Thread abstract class. Your platform’s kernel also provides an implementation of the Thread class.

### Locks

Your platform’s kernel implementation provides mutex functionality with the CreateMutex() and DestroyMutex() methods. The kernel also provides implementations of the following classes that relate to mutex functionality:

- **Lockable**  An abstract class providing Lock() and Unlock() methods.
- **Mutex**  An abstract class derived from Lockable() that adds a TryLock() method.
- **ScopedLock**  A class to ensure that a Mutex object is unlocked when the ScopedLock object goes out of scope.

### Events

Your platform’s kernel implementation provides the Event class. Use the Event class for setting, clearing, and waiting on events. The kernel provides the methods CreateEvent() and DestroyEvent().
Messages and message queues
Your platform’s kernel implementation provides the Message and MessageQueue classes. The kernel also provides the methods CreateMessageQueue() and DestroyMessageQueue(). Use these classes and methods to send and receive messages.

Memory and string manipulation
Your platform’s kernel implementation provides a set of memory and string manipulation functions. For example, the kernel provides an implementation of memcpy(), strcmp(), and strcat(). For the complete list, see IAEKernel.h.

Operating system functionality
Flash Lite for the digital home includes modules for interacting with the operating system of your platform. These modules are the following:

IFileSystem  Provides file system operations.
IProcess     Provides interprocess locks and shared memory.
ISocket      Provides socket operations.

The source distribution provides the implementation of these modules for Linux platforms, as well as for a Win32 development environment. If your platform does not use Linux, a system developer for your platform provides the implementations.

The public interface files are in stagecraft/include/ae/os.

File manipulation
The public interface files for file manipulation are in stagecraft/include/ae/os/filesystem in IFileSystem.h and File.h.

The IFileSystem.h file contains the definition of the IFilesystem module. Use this module to create and destroy files.

The File.h file contains the definition of the File class. This class contains the methods you need for file manipulation. For example, use this class to do the following:

- Open and close files.
- Read and write to files.
- Seek to a position in a file.
- Create directories.

For a complete list of methods, see IFileSystem.h and File.h.

Interprocess locks and shared memory
The public interface files for interprocess locks and shared memory are in stagecraft/include/ae/os/process in IProcess.h, NamedLock.h, and SharedMemory.h.

The IProcess.h file contains the definition of the IProcess module. Use this module to create and destroy named locks and shared memory. Use NamedLock objects to provide mutual exclusion among operating system processes. Use SharedMemory objects, which derives from NamedLock, to share a region of memory among operating system processes.
Note: These objects are not thread-safe. To provide thread-safety, create a separate NamedLock object or SharedMemory object in each thread. Then, these objects provide locking or safe memory access among threads as well as among processes.

The NamedLock.h file contains the definition of the NamedLock class. This class contains methods such as Lock(), Unlock(), and IsLocked(). For a complete list of methods, see NamedLock.h. The SharedMemory.h file contains the definition of the SharedMemory class. This class contains the methods GetAddress() and GetSize(). For details, see SharedMemory.h.

Sockets

The public interface files for sockets are in stagecraft/include/ae/os/socket in ISocket.h and Socket.h.

The ISocket.h file contains the definition of the ISocket module. Use this module to create and destroy sockets. The ISocket module also provides methods for converting between Internet Protocol version 4 (IPv4) 32-bit addresses and dot notation. Another method resolves a host name to an IPv4 32-bit address. For details, see ISocket.h.

The Socket.h file contains the definition of the Socket class. Use this class to open and close sockets, and to send and receive data on a socket. For details, see Socket.h.

The memory watchdog

Each StageWindow instance creates a StageWindowMemoryWatchdog object. Use this object to allocate large blocks of system memory. Doing so tracks the memory allocated with all the memory allocated within the StageWindow instance. This memory tracking is useful during your development process.

The memory limit is specified as a parameter to the StageWindow instance. For example, the host application stagecraft.exe, provided with the source distribution in stagecraft_main.cpp, takes a command-line parameter called memlimit. Stagecraft.exe passes this value to the IStagecraft interface. The implementation of the IStagecraft interface passes the value to the StageWindow instance that it creates. If the IStagecraft implementation passes 0 to the StageWindow instance, the StageWindow instance does not track memory. No memory tracking is the default behavior.

A StageWindowMemoryWatchdog object does the following:

- Allocates and frees blocks of memory.
- Tracks how much memory has been allocated.
- Keeps a data member indicating the maximum amount of allocated memory. The host application passes this value to the StageWindow instance.
- Checks whether a memory allocation exceeds the maximum amount. If so, the StageWindowMemoryWatchdog object terminates the Flash Lite instance.

Access the StageWindow instance’s StageWindowMemoryWatchdog object with the following line of code.

    ae::stagecraft::MemoryWatchdog * pMemoryWatchdog = pStageWindow->GetMemoryWatchdog();

Placing code in the directory structure

When you develop a platform-specific driver or decoder, create a subdirectory for your platform in the following directory:

Updated 13 May 2009
stagecraft/thirdparty-private/<yourCompany>/stagecraft-platforms

Note: The stagecraft directory in these examples is the directory into which you installed the source distribution for Flash Lite for the digital home.

Substitute your company name for <yourCompany>. For example, create the following subdirectory for your platform development:

stagecraft/thirdparty-private/CompanyA/stagecraft-platforms/yourPlatform

Put your header and source files in the yourPlatform directory or subdirectories of the yourPlatform directory.

Building platform-specific drivers and decoders

Setting build-related environment variables

The build process for Flash Lite for the digital home uses two environment variables.

SC_PLATFORM This environment variable indicates which platform to build. The platform corresponds to a subdirectory of stagecraft/thirdparty-private. However, Adobe recommends you use a subdirectory under stagecraft/thirdparty-private/<yourCompany>/stagecraft-platforms. Set this environment variable to the full path of your platform subdirectory.

Note: Providing the full path name is different than when building the platforms provided with the source distribution. For the platforms provided with the source distribution, set SC_PLATFORM to the name of the appropriate platform directory under stagecraft/build/linux/platforms. For example, set SC_PLATFORM to x86Desktop.

SC_BUILD_MODE This environment variable indicates whether to build a release or debug version of Flash Lite for the digital home. The two values are debug and release.

Set these environment variables before running the make utility. If you do not, the make utility prompts you for them. When prompting you for the SC_PLATFORM value, the make utility lists the full path names of the subdirectories under stagecraft/thirdparty-private that contain a Makefile.config file. In addition, the make utility lists the platforms that the source distribution provides. These platforms are in subdirectories under stagecraft/build/linux/platforms. For the provided platforms, the make utility prompt includes only the name of the subdirectory, not the full path name.

Creating your platform Makefile.config file

The Makefile.config file specifies variables that the make utility uses. To create your platform’s Makefile.config, do the following:

1 Copy the Makefile.config from the directory stagecraft/build/linux/platforms/generic to the stagecraft/thirdparty-private/<yourCompany>/stagecraft-platform subdirectory for your platform. For example:

```
  cd stagecraft/thirdparty-private/<yourCompany>/stagecraft-platform/yourPlatform
  cp ../../build/linux/platforms/generic/Makefile.config .
```

2 Edit the Makefile.config in your platform directory. Modify the required variables as appropriate for your platform. You can also provide values for the optional variables, and you can add variables.

Note: Take care when editing Makefile.config (or any makefile) to use an editor that does not replace the tabs with spaces.

The following table describes the variables you set in Makefile.config. Provide values for the required variables. The Makefile in stagecraft/linux/platforms provides the default values for the optional variables.
Creating your .mk file

Each driver or decoder has a .mk file. The primary purpose of the .mk file is to specify the source files to build.

Create a .mk file for your platform-specific driver or decoder. To create the .mk file, copy the appropriate .mk file from the stagecraft/build/linux/modules directory to the subdirectory for your platform under stagecraft/thirdparty-private/<yourCompany>/stagecraft-platform. This directory is the same one in which you put the Makefile.config file for your platform. Use the name of the copied file for the file you create.

The following table shows the .mk file to copy for each driver or decoder:
After you create the .mk file in your platform subdirectory, edit it as follows:

1. Specify the kinds of target to build. You specify any combination of these three kinds of targets: shared libraries, static libraries, or executables. For example:

   SC_MODULE_BUILD_SHARED_LIB:= yes
   SC_MODULE_BUILD_STATIC_LIB:= yes
   SC_MODULE_BUILD_EXECUTABLE:= no

   These variables are already specified from copying the .mk file from the stagecraft/build/linux directory. Edit the values as required by your platform. Typically, you do not need to edit these variables.

2. Specify the module directory and the module source files to build in the variables SC_MODULE_SOURCE_DIR and SC_MODULE_SOURCE_FILES. These variables are already specified from copying the .mk file from the stagecraft/build/linux directory. Typically, you do not add to the list of module source files. However, sometimes you delete some filenames. For example, the following lines show these variables in the IStreamPlayer.mk in stagecraft/build/linux/modules:

   SC_MODULE_SOURCE_DIR:= $(SC_SOURCE_DIR_DDK)/streamplayer
   SC_MODULE_SOURCE_FILES := \
    IStreamPlayerBase.cpp \ 
    StreamPlayerBase.cpp \ 
    ShellCommands.cpp \ 
    software/IStreamPlayerImpl.cpp \ 
    software/SoftwareStreamPlayer.cpp

   Because you provide your own StreamPlayer implementation, you do not want to build the software StreamPlayer implementations provided with the source distribution. Therefore, modify the IStreamPlayer.mk for your platform as follows:

   SC_MODULE_SOURCE_DIR:= $(SC_SOURCE_DIR_DDK)/streamplayer
   SC_MODULE_SOURCE_FILES := \ 
    IStreamPlayerBase.cpp \ 
    StreamPlayerBase.cpp \ 
    ShellCommands.cpp

3. Add these variables to the .mk file: SC_PLATFORM_SOURCE_DIR and SC_PLATFORM_SOURCE_FILES. These variables specify the platform directory and the platform source files to build. For example:

   SC_PLATFORM_SOURCE_DIR:= $(SC_SOURCE_DIR_DDK)/streamplayer
   SC_PLATFORM_SOURCE_FILES := \ 
    YourPlatformIStreamPlayerImpl.cpp \ 
    YourPlatformStreamPlayer.cpp

   **Note:** The Makefile in stagecraft/build/linux automatically creates the variable SC_PLATFORM_MAKEFILE_DIR. The Makefile sets this variable to the value of the SC_PLATFORM environment variable.
In **SC_PLATFORM_SOURCE_FILES**, list all the source files for your platform-specific driver or decoder. Provide the path relative to the **SC_PLATFORM_SOURCE_DIR** directory. For example, if you have a file helperClass.cpp in the subdirectory helpers in stagecraft/thirdparty-private/yourCompany/stagecraft-platforms/yourPlatform/streamplayer, set **SC_PLATFORM_SOURCE_FILES** as follows:

```
SC_PLATFORM_SOURCE_FILES := \
    YourPlatformIStreamPlayerImpl.cpp \
    YourPlatformStreamPlayer.cpp \ 
    helpers/helperClass.cpp
```

4 Specify the value for **SC_ADDITIONAL_MODULE_OBJ_SUBDIRS**. This variable specifies any subdirectories of **SC_MODULE_SOURCE_DIR** that contain module source files. This variable is already specified from copying the .mk file from the stagecraft/build/linux directory. Typically, it specifies subdirectories containing software implementations of drivers and decoders which you are replacing. In that case, comment it out. However, if you are using a software implementation, make sure that it has the correct value. For example, if you are using the I2D software implementation during initial graphics driver testing, your IGraphicsDriver.mk file contains the following:

```
SC_MODULE_SOURCE_DIR:= $(SC_SOURCE_DIR_DDK)/graphicsdriver
SC_MODULE_SOURCE_FILES:= \
    GraphicsDriver.cpp \
    host/I2DImpl.cpp

SC_ADDITIONAL_MODULE_OBJ_SUBDIRS := host
```

5 Specify the value for the following variables if you want to override the values specified in Makefile.config:

- **SC_CFLAGS_GENERIC**
- **SC_CFLAGS_DEBUG**
- **SC_CFLAGS_RELEASE**
- **SC_CXXFLAGS_GENERIC**
- **SC_CXXFLAGS_DEBUG**
- **SC_CXXFLAGS_RELEASE**
- **SC_LDFLAGS_SHAREDLIB**
- **SC_LDFLAGS_EXECUTABLE**
- **SC_ARFLAGS_STATICLIB**

6 Create and set the following **SC_PLATFORM_**variables if you have additional flags for building the files you listed in **SC_PLATFORM_SOURCE_FILES**:

- **SC_PLATFORM_CFLAGS_GENERIC**
- **SC_PLATFORM_CFLAGS_DEBUG**
- **SC_PLATFORM_CFLAGS_RELEASE**
- **SC_PLATFORM_CXXFLAGS_GENERIC**
- **SC_PLATFORM_CXXFLAGS_DEBUG**
- **SC_PLATFORM_CXXFLAGS_RELEASE**
- **SC_PLATFORM_LDFLAGS_SHAREDLIB**
- **SC_PLATFORM_LDFLAGS_EXECUTABLE**
- **SC_PLATFORM_ARFLAGS_STATICLIB**
Note: The Makefile applies these flags only to building the files specified in SC_PLATFORM_SOURCE_FILES. The Makefile does not apply these flags to the files listed in SC_MODULE_SOURCE_FILES.

Running the make utility

Before building Flash Lite for the digital home, install any third-party libraries that your platform depends on. See Third-party libraries in Getting Started with Adobe Flash Lite for the Digital Home. To build Flash Lite for the digital home, including your platform-specific drivers and decoders, do the following:

1. Make sure the environment variables SC_BUILD_MODE and SC_PLATFORM are set.
2. Change to the directory stagecraft/build/linux.
3. Enter the following command:
   
   `make`

To build a specific driver or decoder, do the following:

1. Make sure the environment variables SC_BUILD_MODE and SC_PLATFORM are set.
2. Change to the directory stagecraft/build/linux.
3. Enter the following command:

   `make <driver name>`

   For `<driver name>` use one of the following:

   - IGraphicsDriver
   - IStreamPlayer
   - IFL31NativeSoundOutput
   - IImageDecoder
   - IAudioDecoder
   - IVideoDecoder

If you want to force a rebuild of a target, rather than build according to dependency rules, use the following commands:

```bash
## Rebuild all modules
make rebuild

## Rebuild individual modules
make rebuild-IGraphicsDriver
make rebuild-IStreamPlayer
make rebuild-IFL31NativeSoundOutput
make rebuild-IImageDecoder
make rebuild-IAudioDecoder
make rebuild-IVideoDecoder
```

To remove all output files resulting from running the make utility, use the following commands:

```bash
rm -rf stagecraft/build/linux/*
```
## Clean all modules
make clean

## Clean individual modules
make clean-IGraphicsDriver
make clean-IStreamPlayer
make clean-IFL31NativeSoundOutput
make clean-IImageDecoder
make clean-IAudioDecoder
make clean-IVideoDecoder

Other parameters available in the `make` utility include:

- `quiet` Use this parameter to reduce diagnostic output from the `make` utility. For example, the following command builds all modules with reduced diagnostic output:

  ```
  make quiet
  ```

- `doxydocs` Use this parameter to generate the Doxygen documents. These documents provide the class and method comments in an accessible format. For example, the following command builds the Doxygen output for all of Flash Lite for the digital home:

  ```
  make doxydocs
  ```

---

## Measuring performance

One way to measure the performance of your platform is to determine the speed at which it renders SWF content. A SWF movie is structured as one or more datasets known as a frame. Like the frames of a film in the physical world, the frames of a SWF movie execute in start-to-finish order. Frames usually contain content that is displayed visibly or auditorily to the end user. Each frame can also contain Adobe® ActionScript® content to execute. A frame can actually contain visual content, ActionScript content, both, or neither.

You can pass your host application a command-line parameter to trace the performance of the Flash Lite instance with regard to frame updates. Your host application interacts with an IStagecraft interface, and passes parameters from its command line to StageWindow instances. An example is in the host application provided with the source distribution: `stagecraft/source/executables/stagecraft/stagecraft_main.cpp`. One of the command-line parameters is `--tracefps`. Use `--tracefps` to print frames-per-second statistics to the command shell.

For example, the host application provided with the source distribution is an executable called `stagecraft.exe`. The command to print frames-per-second statistics is the following:

```
./stagecraft --tracefps pathToSWF  ## debug builds only
```

Output from the `--tracefps` option looks like the following example:

```
5010 ms: FlashFrames = 324 (64.7 FPS), FrameBufUpdates = 325 (64.9 FPS)
5010 ms: FlashFrames = 310 (61.9 FPS), FrameBufUpdates = 310 (61.9 FPS)
5010 ms: FlashFrames = 302 (60.3 FPS), FrameBufUpdates = 302 (60.3 FPS)
5000 ms: FlashFrames = 312 (62.4 FPS), FrameBufUpdates = 0 (0.0 FPS)
5010 ms: FlashFrames = 312 (62.3 FPS), FrameBufUpdates = 0 (0.0 FPS)
```

The first number on each line indicates the milliseconds that elapsed while the `--tracefps` option collected the statistical data indicated on the rest of the line. Each sample in the preceding example output is approximately 5 seconds long.

---

Updated 13 May 2009
The FlashFrames value indicates the number of frames of the SWF movie that the Flash Lite instance played during the monitored amount of time. Specifically, a Flash frame advances every time that the Flash Lite instance determines that a frame interval has elapsed. The Flash Lite instance then runs all the animations, actions, and ActionScript associated with the next Flash frame. This value is equivalent to the number of times an ActionScript onFrameEnter notifier on the root timeline would execute if it were present. This frame rate is set when the SWF movie is authored. The frame rate can be independent of the frame rate of video clips that can be running inside the Flash Lite instance.

The FrameBufUpdates value specifies the number of times the Flash Lite instance updated the render plane with an update to Flash content rendered into the plane. This frame rate can exceed the FlashFrames rate if an instance of the ActionScript Video class plays video at a higher frame rate than the SWF movie frame rate. The FrameBufUpdates rate is a good indicator of the frame rate of embedded video playback in a SWF movie: The playback rate of video in the SWF movie usually limits the rate at which the frame buffer updates.

The final two lines of output record zero frame buffer updates because the SWF movie stopped playing upon reaching its end. However the FlashFrames statistic shows that the Flash Lite instance continued to run at approximately 62 frames per second.

The --tracefps option is available only for platforms built in debug mode. Because debugging code adds some performance overhead, it is likely that a release build of your platform can render more frames per second than a debug build can.

For more detailed performance statistics, pass the --tracefpsfull option to the stagecraft command:

```
./stagecraft --tracefpsfull pathToSWF  ## debug mode only
```

Output from the --tracefpsfull option looks like the following example:

```
5010 ms:FlashFrames = 353 (70.5 FPS), FrameBufUpdates = 354 (70.7 FPS)
  DoPlays = 2057 (410.6 PS), TargetDoPlaysPS = 125.0, TargetFPS = 125.0
  SkippedFrames = 0, ActsPending = 2056, LongDoPlays = 215, ZeroTOs = 1918

5000 ms:FlashFrames = 351 (70.2 FPS), FrameBufUpdates = 351 (70.2 FPS)
  DoPlays = 2100 (420.0 PS), TargetDoPlaysPS = 125.0, TargetFPS = 125.0
  SkippedFrames = 0, ActsPending = 2100, LongDoPlays = 206, ZeroTOs = 1960
```

The values provided in the output have the following meanings:

- **DoPlays**
  
  The number of times Flash Lite for the digital home has called the FI_DoPlay() function in the Flash Lite instance. FI_DoPlay() provides the “heartbeat” of the Flash Lite instance and is used to process each frame in a SWF file.

- **TargetDoPlaysPS**
  
  The number of times the Flash Lite instance has requested the Flash Lite for the digital home to call FI_DoPlay() per second. When video is playing in the Flash instance, this rate typically exceeds the frame rate authored in the SWF movie.

- **TargetFPS**
  
  The frame rate authored in the SWF movie.

- **SkippedFrames**
  
  The number of Flash frames that the Flash Lite instance has skipped rendering to synchronize the Flash animation rate to audio playback.

- **ActsPending**
  
  Updated 13 May 2009
The number of times that the “actions pending” flag has been set in the return value of FI_DoPlay(). This number approaches the value of DoPlays in periods of intense CPU usage and when the Flash Lite instance determines that the actual frame rate is lagging behind the target frame rate.

- **LongDoPlays**
  The number of times that FI_DoPlay() takes longer than half the requested DoPlay interval to complete. As a fraction of total DoPlays, it is a rough indicator of CPU load of the Flash Lite instance.

- **ZeroTOs**
  The number of times Flash Lite for the digital home has called FI_DoPlay() twice without sleeping. If the actual frame rate isn’t reaching the target frame rate, this value is non-zero. The non-zero value indicates that Flash Lite for the digital home is attempting to catch up to the expected position in the timeline.